



## - L'API des collections en Java -





# SOMMAIRE

✓ 1 - <a href="#">Les structures de données.</a>	1
✓ 2 - <a href="#">Les structures de données séquentielles.</a>	2
✓ 3 - <a href="#">Les tableaux.</a>	3
✓ 4 - <a href="#">Les listes.</a>	9
✓ 5 - <a href="#">Les piles et les files.</a>	13
✓ 6 - <a href="#">Java et les API de collections</a>	20
✓ 7 - <a href="#">La classe <b>Vector</b></a>	21
✓ 8 - <a href="#">Le dictionnaire ou HashTable</a>	29
✓ 9 - <a href="#">La hiérarchie de l'API <i>Collection</i></a>	36
✓ 10 - <a href="#">L'interface <b>List</b></a>	39
✓ 11 - <a href="#">Parcourir une collection</a>	41
✓ 12 - <a href="#">L'interface <b>Set</b></a>	50
✓ 13 - <a href="#">L'interface <b>Map</b></a>	55
✓ 14 - <a href="#">Une interface dérivée de <i>Map</i> : <b>SortedMap</b></a>	58
✓ 15 - <a href="#">L'interface <b>Map.Entry</b></a>	60
✓ 16 - <a href="#">La généricité</a>	61
✓ 17 - <a href="#">Les classes <b>Arrays</b> et <b>Collections</b></a>	67
✓ 18 - <a href="#">Stratégie pour utiliser une collection</a>	68
✓ 19 - <a href="#">Résumé ... Classifications des collections</a>	69



# - 1 - Les structures de données

## Définition

Une **structure de données** est une organisation logique, construite et destinée à contenir des informations.

Cette structure, organisée donc en tant que **conteneur**, permet de globaliser, de simplifier et d'**unifier** les traitements réalisables sur les informations contenues.

## Exemples

Parmi les structures de données les plus usitées, que l'on appelle également des **collections**, on trouve :

- Les tableaux,
- Les listes chaînées,
- Les piles, les files,
- Les graphes,
- Les arbres,
- ...etc...

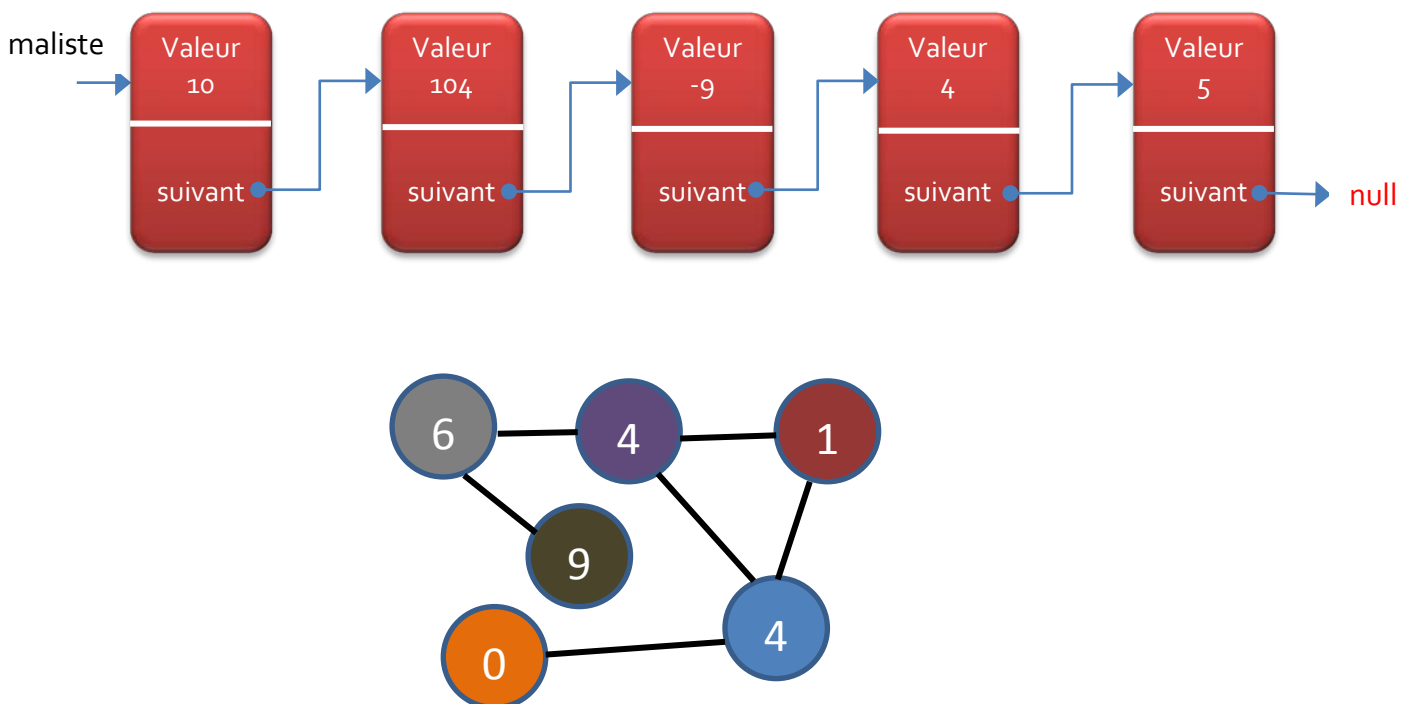


Figure 1 : Quelques structures de données

## - 2 - Les structures de données séquentielles

- ✓ Dans ce type de collections, l'emplacement où sont rangées les données n'est pas imposé.
- ✓ La nouvelle donnée peut être mise n'importe où dans la structure : au début, à la fin, entre les deux.
- ✓ Les collections séquentielles principales sont :
  - Les tableaux.
  - Les listes.

## 3 - Les tableaux

### Définition

Un **tableau** est une collection indexée d'éléments dont le **nombre** est fixé à la **création**. Les éléments d'un tableau sont repérés par des entiers, généralement à partir de 0.

Un tableau représente un ensemble d'emplacements - ou cases - contigus en mémoire.

### Représentation

Dans l'exemple ci-dessous, on représente un tableau d'entiers.

8	0	-4	33	17
---	---	----	----	----

Figure 2 : Un tableau de 5 entiers.

### Quelques propriétés concernant les tableaux en Java

- Un **tableau** est un **objet** qui peut être référencé par une **variable** qui le nomme.
- Un **tableau** a une **taille invariable**, qui est définie à l'instanciation de l'objet et qui ne peut plus être modifiée ensuite.
- Les éléments d'un tableau sont d'un **type donné**. Ce type peut être un **type primitif** ou un **type objet**.
- Un **tableau** est une instance d'une classe créée dynamiquement. Cette classe hérite directement **d'Object**.
- Chaque **tableau** définit une variable **length** qui renseigne sur le nombre d'éléments qu'il contient.
- Un **tableau** peut avoir plusieurs dimensions, traduites par le nombre de paires de crochets à la déclaration.
- On **accède** à un élément particulier d'un tableau grâce à un **index**, ou plusieurs si le tableau est au moins de deux dimensions.
- L'indice du premier élément d'un tableau est 0 et celui du dernier élément le **nombre d'éléments du tableau - 1**.

## Exemple de déclarations de variables référençant un tableau

```
// Première forme de déclaration
int [] tableau-entiers ;      // Pour déclarer le tableau d'entiers de la page précédente

char tableau-caracteres [] ; // Deuxième forme possible de déclaration

Vehicule [ ] parcVehicules ; // Déclaration d'un tableau d'objets de la classe Vehicule
```

## Déclaration d'un tableau

- Dimension déterminée par le nombre d'éléments à la déclaration.

```
String [ ] unGroupe = {"Pierre", "Paul", "Jacques"} ;
```

- Dimension déterminée avec l'opérateur *new*.

```
String [ ] unAutreGroupe = new String [5] ; // 5 éléments à null
```

## Accès aux éléments d'un tableau

- L'accès aux éléments d'un tableau se fait grâce à la notation « crochet ». Exemple :

```
int tableau[] = new int [10];
for ( int i = 0 ; i < 10 ; i++ ) {
    tableau[i] = 2 * i;
}
// Affichage de la table de multiplication par 2
for ( int i : tableau ) {
    System.out.print(i+"-");
}
```

- Le **nombre de paires de crochets** traduit la dimension du tableau. Exemple :

```
short [ ] [ ] matrice;      // Déclaration d'un tableau à deux dimensions
```

- La création d'un tableau dont les éléments sont de **type objet** n'instancie pas d'objet pour chacun de ses éléments : toutes les références au sein du tableau sont valorisées à *null*.



► Exemple :

```
// Déclaration d'un tableau de 5 objets de classe Vehicule  
Vehicule [ ] parcVehicules = new Vehicule[5] ;
```



Figure 3 : Un tableau de 5 *Vehicule*.

parcVehicules

- Un tableau est un objet en **Java**, il hérite à ce titre de la métaclasse *Object* et dispose ainsi de **toutes ses méthodes**.
- Lors de l'utilisation ultérieure d'un tableau, la **JVM** surveille la validité des opérations et peut être amenée à déclencher des exceptions, parmi lesquelles :
  - ✓ *ArrayIndexOutOfBoundsException* : l'accès d'un élément se fait en dehors de la plage  $[0, n - 1]$ ,  $n$  étant la taille du tableau.
  - ✓ *ArrayStoreException* : insertion d'un élément dans le tableau dont le type est incorrect, relativement au type du tableau.
  - ✓ *NegativeArraySizeException* : la taille du tableau est négative ...

## Exemple de tableau avec ses contraintes

```
package com.michel.entites;

/**
 * @author Michel
 */
public class Auteur {

    private String[] livres;
    private String nom;

    public Auteur(String nom, String[] livres) {
        this.nom = nom;
        this.livres = livres;
    }

    // Renvoie une chaîne formatée
    @Override
    public String toString() {

        String description = nom.toUpperCase();

        // Itération sur le tableau livres[]
        for (String livre : livres) {
            description += "\n\t----- " + livre;
        }

        return description;
    }

    public void ajouterLivre(String unLivre) {

        // Déclaration d'un tableau temporaire
        String[] livresAux = new String[livres.length + 1];

        // Copie élément par élément du tableau initial → tableau temporaire
        for (int i = 0; i < livres.length; i++) {
            livresAux[i] = livres[i];
        }

        livresAux[livresAux.length - 1] = unLivre;
        livres = livresAux;
    }
}
```

La classe Auteur

```
package com.michel.application;
import com.michel.entites.Auteur;
import static javax.swing.JOptionPane.*;
```

```
/**
 * @author Michel
 */
```

```
public class Principale {
```

La classe Principale instanciant des *Auteur*

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

```
        // Déclaration de deux objets tableau de String
```

```
        String livresFlaubert[] = {"Madame Bovary", "Salammbô"};
```

```
        String livresBalzac[] = {"Le Père Goriot", "Eugénie Grandet", "Le Colonel Chabert"};
```

```
        // Déclaration de deux objets Auteur
```

```
        Auteur auteurFlaubert = new Auteur ("Flaubert", livresFlaubert);
```

```
        Auteur auteurBalzac = new Auteur ("Balzac", livresBalzac);
```

```
        // Déclaration d'un tableau de deux auteurs
```

```
        Auteur tableauAuteurs[] = {auteurFlaubert, auteurBalzac};
```

```
        // Modification d'un objet Auteur
```

```
        auteurFlaubert.ajouterLivre("L'Education Sentimentale ");
```

```
        String resultat="";
```

```
        // Itération sur les auteurs
```

```
        for (Auteur auteur : tableauAuteurs) {
```

```
            System.out.println(auteur);
```

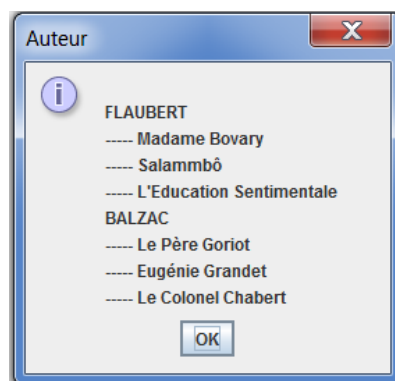
```
            resultat += "\n" + auteur;
```

```
        }
```

```
        showMessageDialog(null, resultat, "Auteur", INFORMATION_MESSAGE);
```

```
    }
```

```
}
```



**Figure 4** : Exploitation de tableaux.



## Constat de l'exemple précédent

- Les **tableaux** sont certes simples d'utilisation mais sont **inadaptés** pour gérer une quantité importante d'informations du même type quand leur nombre est indéterminé.
- Par exemple, le nombre de messages postés pour un sujet dans un forum n'étant pas limité, il existe des solutions plus efficaces que les tableaux pour stocker ces messages.
- On constate notamment les défauts suivants liés à l'emploi des tableaux :
  - Les tableaux ne sont pas redimensionnables (taille déclarée une fois pour toutes).
  - L'insertion d'un élément au milieu d'un tableau oblige à décaler tous les éléments qui suivent.
  - La recherche d'un élément particulier dans un grand tableau est peu performante s'il n'est pas trié.
  - Les indices pour accéder aux éléments d'un tableau **ne sont que des entiers**.

## - 4 - Les listes

### Définition

- Une **liste** est aussi une **collection séquentielle d'éléments** mais, à la différence des tableaux, ces **éléments ne sont pas contigus** en mémoire.
- On dit qu'ils sont **chaînés**.
- Une liste chaînée que l'on ne peut parcourir que dans un seul sens est dite « **simplement chaînée** ». Lorsque la liste peut être parcourue dans les deux sens, elle est « **doublement chaînée** ».

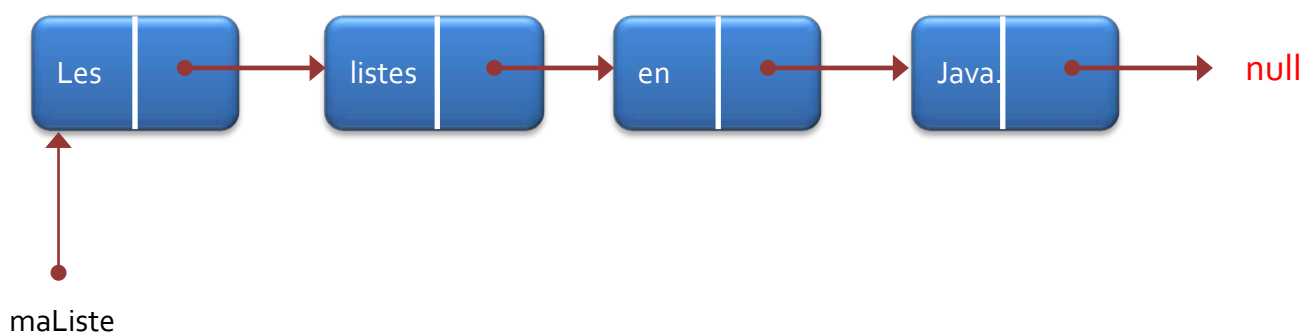
### Avantages

- **Plus de limite de taille** comme avec les tableaux.
- La valeur que l'on insère dans un maillon peut être de n'importe quel type.
- Les ajouts d'éléments se font **dynamiquement**.
- Les insertions, suppressions et mises à jour d'éléments se font par simples **affectations** : il n'est plus nécessaire de décaler les autres éléments.

### Inconvénients

- Le code à écrire est **plus complexe** que la simple indexation de tableaux.
- Certaines opérations **sont plus lentes** qu'avec les tableaux : il faut par exemple parcourir la liste à partir de son début (dont on doit conserver la **référence**) pour rechercher un élément par exemple.

### Représentation logique d'une liste ( ici simplement chaînée )



**Figure 5** : Exemple de liste chaînant des *String*.

- Les éléments constituant une liste se nomment les « **maillons** ».
- L'organisation de ces maillons est toujours la même :
  - Si la liste est simplement chaînée, on trouve dans le maillon :
    - ✓ **La valeur** que l'on souhaite stocker dans le maillon (tout type convient).
    - ✓ **La référence** qui contient l'endroit où est rangé le maillon suivant.
  - Si la liste est doublement chaînée, on trouve dans le maillon :
    - ✓ **La valeur** que l'on souhaite stocker dans le maillon (tout type convient).
    - ✓ **La référence** qui désigne l'endroit où est rangé le maillon suivant.
    - ✓ **La référence** qui désigne l'endroit où est rangé le maillon précédent.
- La **fin de la liste** est matérialisée avec une référence sur **null**.
- Le **début de la liste** est désigné avec une référence.

- Codons maintenant en **Java** la classe correspondant à la liste de la figure 5 précédente :

```
package com.michel.entites;

/**
 *
 * @author Michel-HP
 */
public class Maillon {

    private String mot;        // La valeur utile dans le maillon
    private Maillon suivant;    // La référence désignant le maillon suivant

    public Maillon(String mot, Maillon suivant) {
        this.mot = mot;
        this.suivant = suivant;
    }

    public String getMot() {
        return mot;
    }

    public void setMot(String mot) {
        this.mot = mot;
    }

    public Maillon getSuivant() {
        return suivant;
    }

    public void setSuivant(Maillon suivant) {
        this.suivant = suivant;
    }
}
```

Figure 6 : Code de la classe modélisant un maillon de la liste.

Dans un exemple, que vous complèterez dans le TP correspondant, mettons en œuvre les fonctionnalités suivantes, relatives à la classe **Maillon** précédente.

- Les mots constituant la valeur de chaque maillon se trouvent dans un tableau :

```
String phrase[] = { "Les" , "listes" , "en" , "Java" } ;
```

- Seul vous est proposé maintenant le code de la méthode permettant l’affichage de la liste chaînée. Il vous appartiendra dans la *série d’exercices N°3* dédiée aux collections de coder la construction de la liste à partir du tableau de *String* présenté ci-dessus.

```
public static void afficherListe(Maillon maListe) {  
    // Le paramètre maListe représente le début de la liste  
    // soit l'adresse où est rangé le premier maillon.  
    Maillon parcours = maListe;  
  
    for (parcours = maListe; parcours != null; parcours = parcours.getSuivant()) {  
        System.out.print(parcours.getMot() + "-");  
    }  
}
```

Figure 7 : Parcours itératif de la liste et affichage des maillons.

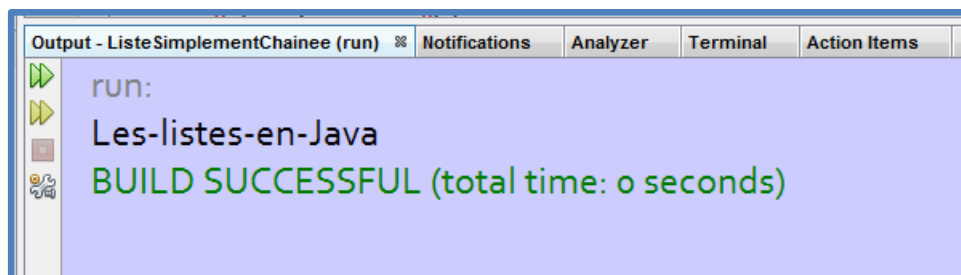


Figure 8 : Affichage de la liste.



## - 5 - Les piles et les files

- Contrairement aux structures de données séquentielles étudiées précédemment, les **piles** et les **files** n'autorisent pas les opérations d'ajout, de suppression ou de lecture à se faire n'importe où.
- Ces deux types de conteneurs **imposent une politique particulière** concernant ces opérations.

### La pile

- La **notion de pile** en programmation est très répandue et totalement assimilable à une pile d'assiette dans l'évier d'une cuisine : on ne peut poser **une nouvelle assiette** qu'au **sommet** de la pile déjà constituée et on ne peut retirer, de cette pile, que l'assiette se **situant au sommet**.
- Ces deux opérations en informatique se nomment respectivement **empiler** (pour le dépôt) et **dépiler** (pour le retrait).
- Ces deux opérations liées à cette notion de pile ont pour effet de nommer cette structure : **LIFO**. Pour **Last Input First Output**. Dernier entré, premier sorti.

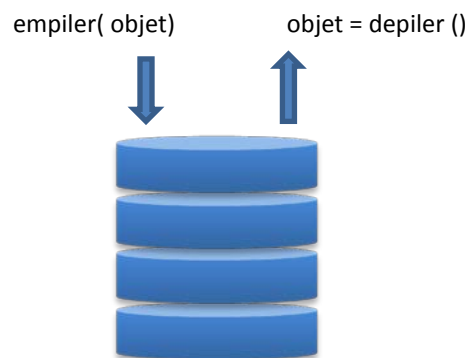


Figure 9 : Dépôt au sommet de la pile et retrait.

- L'**implémentation physique** et **interne** d'une pile va mettre en œuvre la **notion de tableau** ou de **liste** pour empiler/dépiler les éléments.
- C'est grâce à l'un de ces conteneurs que les opérations **empiler** et **depiler** se réaliseront.
- Pour contraindre l'utilisateur de la pile à n'utiliser que les méthodes **empiler(...)** et **depiler()**, on utilise les principes de l'encapsulation pour masquer les détails d'implémentation et pour offrir l'interface public souhaitée.

- Codons maintenant une classe *PileEntier* destinée à gérer ... des entiers.
- La structure accueillant ces entiers, destinés à être empilés/dépilés, sera implémentée en interne dans la classe, **par un tableau** mais, dans la série d'exercices « *Les collections* », vous remplacerez le tableau par une liste chaînée.
- L'**interface** de la pile ne changeant pas, l'opération précédente, consistant à changer la structure physique interne de la pile par une **liste**, sera complètement transparente pour l'utilisateur : il continuera à utiliser les mêmes méthodes de la classe pour empiler et dépiler ses objets.
- On voit bien ici l'importance d'offrir seulement au monde extérieur l'**interface publique** d'une classe et de **s'abstraire de son implémentation**. Cette notion fondamentale en objet garantit l'évolutivité du code. Génial.

Bon : codons la pile :

```

package com.michel.entites;
/**
 * @author Michel-HP
 */
public class PileEntier {

    private int[] pile; // Le tableau où sont déposés les entiers

    // L'index dans le tableau matérialisant le sommet de la pile
    private int index_sommet = 0;

    // La taille de la pile et donc le nbr d'entiers maxi empilables.
    private int taille_pile ;

    public PileEntier(int taille_pile) {
        this.taille_pile = taille_pile;
        pile = new int[taille_pile];
    }

    public void empiler(int valeur) {
        if (index_sommet < taille_pile) { // Il reste de la place dans la pile
            // Dépôt au sommet et incrémentation de l'index le désignant.
            pile[index_sommet++] = valeur;
        }
    }

    public int depiler() { // Il reste au moins un élt. dans la pile
        if (index_sommet != 0) { // Il reste au moins un élt.
            return pile[--index_sommet]; // Mise à jour de l'index du sommet
        } else {
            return 0; // Arbitrairement : gestion d'exception mieux ....
        }
    }

    public void reinitialiser() { // La pile est "vidée" .
        index_sommet = 0;
    }

    // Test de la pile vide <=> l'index du sommet = 0
    public boolean pileVide() {
        return (index_sommet == 0);
    }

    // Test de la pile pleine <=> l'index du sommet = taille de la pile
    public boolean pilePleine() {
        return (index_sommet == taille_pile);
    }
}

```

**Figure 10** : La classe *PileEntier*.

```

package com.michel.application;

import com.michel.entites.PileEntier;
import static javax.swing.JOptionPane.*;
/**
 * @author Michel-HP
 */
public class Principale {

    public static void main(String[] args) {

        PileEntier maPile = new PileEntier(17);

        System.out.println("Etat de la pile ---> " + (maPile.pileVide() == true ? " vide" : " pas vide"));
        System.out.println("Etat de la pile ---> " + (maPile.pilePleine() == true ? "pleine" : " pas
            pleine"));

        for (int i = 0; i < 10; i++) { // Remplissage de 10 entiers aléatoires
            maPile.empiler((int) (Math.random() * 15));
        }

        maPile.reinitialiser();

        int valeur = 1;
        while (! maPile.pilePleine()) { // On empile jusqu'à ce que la pile soit pleine.
            maPile.empiler(valeur++);
        }
        System.out.println("Après les empilements, état de la pile ---> " + (maPile.pileVide() == true
            ? " vide" : " pas vide"));
        System.out.println("Après les empilements, état de la pile ---> " + (maPile.pilePleine() ==
            true ? "pleine" : " pas pleine"));

        String vueMaPile = "";
        while (! maPile.pileVide()) { // On dépile jusqu'à ce que la pile soit vide.
            String message;
            message = "|" + maPile.depiler() + "|\n";
            System.out.print(message);
            vueMaPile += message;
        }
        showMessageDialog(null, "Voici le contenu de la pile : \n" + vueMaPile, "maPile",
            INFORMATION_MESSAGE);
    }
}

```

**Figure 11** : Utilisation de la classe *PileEntier*.

► L'exécution du code précédent produit l'affichage de la boîte de dialogue :

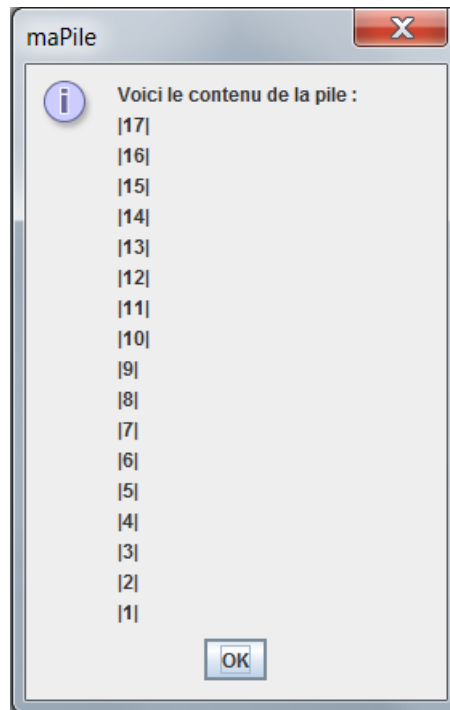


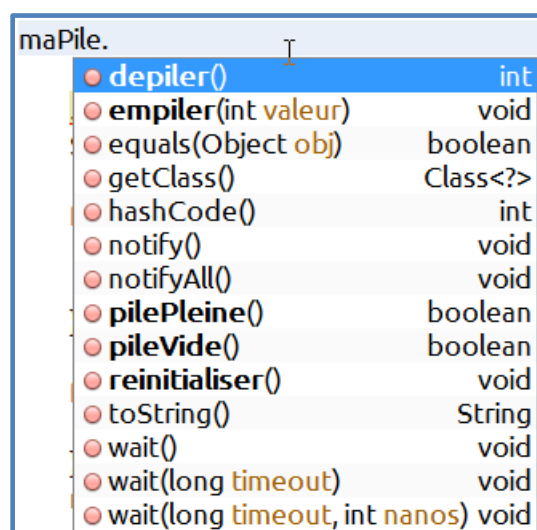
Figure 12 : La classe *PileEntier*.



Il faut remarquer que **les seules opérations possibles** réalisables sur la pile sont celles faisant partie de la **spécification publique** de la classe : toute tentative visant à corrompre l'objet, en accédant à ses variables d'instance, est naturellement rejetée :

```
maPile.pile[6] = 1515 ; // Impossible : on contraint l'utilisateur à passer par empiler(...)
```

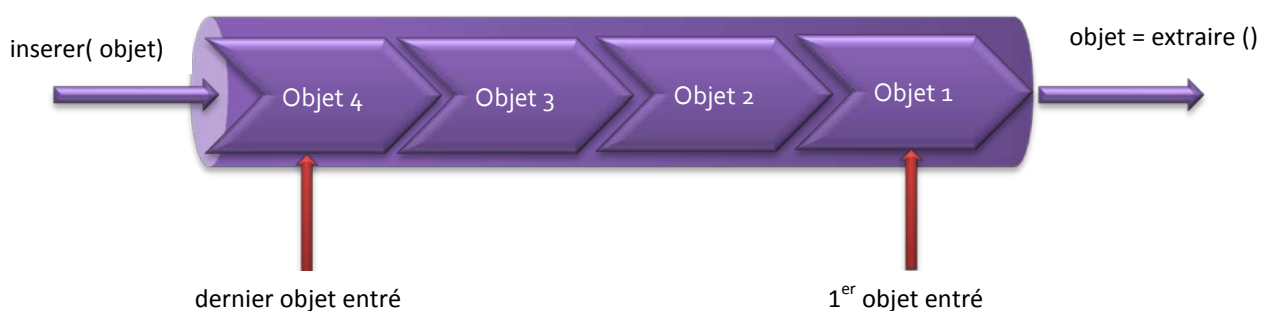
► Seules les propriétés **public** de la classe sont atteignables.



- Ceci atteste qu'il est parfaitement possible de **changer l'implémentation** d'une classe **sans en impacter** le monde extérieur qui l'utilise.
- C'est ainsi que dans le TP que vous mettrez en œuvre, vous pourrez changer, au sein de la classe, le support des entiers empilés - en passant d'un **tableau à une liste** - sans même **toucher au code** de la méthode *main* de la classe *Principale* !! C'est beau.

## La file

- La **notion de file** en programmation est également très répandue. Elle est assimilable à une **file d'attente** devant un guichet à la Poste : premier arrivé, premier servi !!
- Ou bien encore : une imprimante partageable sur le réseau. Les jobs d'impression y sont déposés à la queue leu leu dans la file et cette imprimante extrait de la file chacun des travaux d'impression.
- Ce mode de fonctionnement est à l'origine du nom que l'on donne à cette structure : **F.I.F.O.** : **F**irst **I**nput **F**irst **O**utput.
- Les deux opérations en informatique consistant à mettre une donnée dans la file d'un côté et la sortir de l'autre se nomment respectivement **insérer** (pour le dépôt) et **extraire** (pour le retrait).



**Figure 13** : Insertion dans la file à gauche et retrait à droite.

- Un peu de **vocabulaire** : on parle de « **tête de file** » pour désigner le premier objet disponible.
- La programmation d'une **file** et son exploitation ressemblent fort à celles d'une pile. L'évoquer ici ne représente pas un immense intérêt.
- C'est la raison pour laquelle, à des fins pédagogiques, c'est vous qui, dans l'un des TP de la série dédiée aux collections, aurez le plaisir de coder cette notion de file.

## - 6 - Java et les API de collections

- Pour éviter les désagréments liés aux structures séquentielles types tableaux et pour offrir une **large gamme de services** prêts à l'emploi au développeur, le package *java.util* de **Java** propose tout un ensemble de collections, sous formes de **classes utilitaires**.
- Au sein de ce package, **toutes les classes de collections** sont utilisées pour gérer un ensemble d'éléments **de type objet**. Chaque classe est prévue pour gérer un ensemble avec des caractéristiques différentes.
- En l'occurrence, aucune des classes de collection **n'est limitée en taille**. Chacune résout de façon optimale certaines limitations liées aux tableaux.

### Caractéristiques des collections

- Toute **collection** en **Java** est un objet utilitaire destiné donc à **contenir des objets**.
- Chaque **collection** est une **instance de classe** implémentant l'interface **Collection** ou **Map**.
- Toute collection peut être **parcourue** pour atteindre les objets qu'elle contient grâce à un **énumérateur** ou un **itérateur**.
- Chaque type particulier de collection possède des propriétés particulières et offre des **services spécifiques**.
  - Certaines collections sont disponibles depuis **Java 1.0** :  
*java.util.Vector* et *java.util.Hashtable*.
  - ... d'autres depuis **Java 1.2** :  
*java.util.ArrayList* et *java.util.HashMap*
- Nous allons découvrir, par la pratique, deux collections : **Vector** et **Hashtable**.
- Puis nous étudierons plus formellement les hiérarchies de l'API.



## - 7 - La classe *Vector*

- Etudions d'abord une classe de l'A.P.I. Java 1.0 : *Vector*, qui modélise la notion de tableaux mais qui a la particularité de ne pas limiter en taille le nombre d'objets déposés.
- On peut ajouter des objets en nombre : la collection se **redimensionne** pour accueillir les nouveaux éléments.
- Nous allons détailler la classe *Vector* puis reprendre l'exemple précédent sur les tableaux de livres et d'auteurs en intégrant **la notion de vecteur**. On aura ainsi une bonne représentation des apports des collections.

### Définition d'un vecteur

- Un vecteur est une collection ordonnée d'objets qui s'agrandit **dynamiquement** en fonction des besoins.
- Les éléments d'un vecteur sont indexés avec des entiers, à partir de 0, comme les tableaux.
- La classe *Vector* du package *java.util* permet de définir un vecteur :

```
Vector vecteur = new Vector() ;
```

### Ajout et suppression d'éléments dans un *Vector*

- *addElement( Object )* // Ajoute l'élément à la fin d'un vecteur.
- *insertElementAt( Object, int )* // Insère un nouvel élément à l'indice indiqué.
- *setElementAt( Object, int )* // Remplace l'élément situé à l'indice indiqué par un nouvel élément.
- *removeElement( Object )* // Supprime la 1ère occurrence de l'élément
- *removeElementAt( int )* // Supprime l'élément qui se trouve à l'indice indiqué.

## Accès aux éléments d'un vecteur

- `indexOf( Object )` // Renvoie l'indice d'un élément du vecteur, ou  
// -1 si la recherche a été infructueuse.
- `elementAt( int )` // Renvoie l'élément situé à l'index passé en paramètre.
- `size()` // Renvoie le nombre d'objets stockés.

## Vecteurs et énumérations

- ✓ L'interface *Enumeration* du package *java.util* définit un type abstrait qui déclare des services de contrôle d'itération pour parcourir le contenu d'une collection d'objets.

```
public interface Enumeration {  
    boolean hasMoreElements ( ); // Renvoie true s'il reste des éléments à parcourir  
    Object nextElement( );      // Renvoie le prochain élément de la collection  
}
```

## Itération sur le contenu d'un vecteur

- ✓ La méthode *elements()* de *Vector* renvoie un objet dont le type implémente l'interface *Enumeration*.
- ✓ Reprenons l'exemple des livres et auteurs et remplaçons les tableaux par des instances de la classe *Vector*.

```

import java.util.* ;

public class Auteur {
    private Vector<String> livres; // la v.i. livres devient un Vector<String>
    private String nom;

    public Auteur ( String nom, Vector livres ) {
        this.nom = nom;
        this.livres = livres;
    }

    public String toString() {
        String chaine = nom;

        // Itération sur le vecteur livres
        Enumeration<String> enumeration = livres.elements();

        while (enumeration.hasMoreElements()) {
            chaine += "\n\t" + "- " + enumeration.nextElement();
        }
        return chaine;
    }
    public void ajouterLivre( String unLivre ) {
        livres.addElement( unLivre ); // Très utile pour ajouter un nouveau livre
    }
}

```

**Figure 14** : La nouvelle classe *Auteur*.



On remarquera en '1', la ligne de code suivante dans la classe *Auteur* :

```
private Vector<String> livres; // la v.i. livres devient un Vector<String>
```

La variable d'instance *livres* est déclarée comme étant une instance de type *Vector<String>* : on utilise ici la notion de **généricité**. Cela consiste à typer une classe particulière – ici *Vector* – avec un ou plusieurs autres types - ici *String* -

En conséquence, le type *Vector<String>* signifie : une collection de type *Vector* capable de collectionner des *String*.

Le type *Vector*, qui est une collection, disons « générique », est donc plus précise, plus spécifique avec l'ajout de *<String>* accolé : elle devient une collection de ... *String*.

Sans cette généricité, elle est une simple collection d'*Object* et est donc apte à recevoir potentiellement toute instance de n'importe quel type.

Si l'on avait utilisé un simple *Vector* pour *livres*, il aurait alors été possible de déposer une instance de tout type comme par exemple :

```
livres.addElement(new Date()); // Ajout d'une date dans la collection des œuvres !!
```

Ce qui évidemment n'est pas un comportement souhaitable, c'est pourtant possible. Nous vous invitons à expérimenter ce fait.



De même, l'instruction en '2' ...

```
// Itération sur le vecteur livres
```

```
Enumeration<String> enumeration = livres.elements();
```

... précise que l'instance *enumeration* est un objet capable de se déplacer dans la collection de l'instance à laquelle la méthode *elements()* est appliquée : ici *livres*, une collection de *String*.

Le type d'*enumeration* étant *Enumeration<String>*, il est inutile de transtyper le retour de *nextElement()* puisque l'on est certain d'y trouver des *String*.

Sans l'emploi de cette généricité, il aurait fallu écrire :

```
(String) enumeration.nextElement();
```

... puisque par défaut la méthode *nextElement()* renvoie un *Object*. Là aussi, je vous invite à expérimenter cette particularité :

Remplacez :

```
Enumeration<String> enumeration = livres.elements();
```

par :

```
Enumeration enumeration = livres.elements();
```

Et constatez les effets ....

➤ Explorons maintenant la méthode *main* qui exploite la classe *Auteur*.

```

package com.michel.application;

import com.michel.entites.Auteur;
import java.util.Date;
import java.util.Vector;
import static javax.swing.JOptionPane.*;

/**
 * @author Michel-HP
 */
public class Principale {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // Définition de deux objets Auteur
        Auteur auteurFlaubert = new Auteur("Flaubert", new Vector<String>());
        Auteur auteurBalzac = new Auteur("Balzac", new Vector<String>());

        auteurFlaubert.ajouterLivre("Madame Bovary ");
        auteurFlaubert.ajouterLivre("Salammbô ");

        auteurBalzac.ajouterLivre("Le Père Goriot ");
        auteurFlaubert.ajouterLivre("L'Education Sentimentale ");

        // Définition d'un tableau de deux auteurs
        Auteur tableauAuteurs[] = {auteurFlaubert, auteurBalzac};

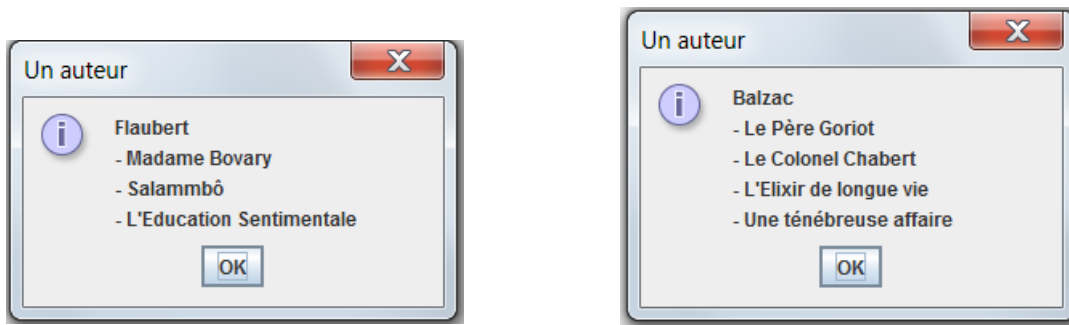
        auteurBalzac.ajouterLivre("Le Colonel Chabert");
        auteurBalzac.ajouterLivre("L'Elixir de longue vie");
        auteurBalzac.ajouterLivre("Une ténébreuse affaire");

        System.out.println("-----");

        // Itération
        for (int i = 0; i < tableauAuteurs.length; i++) {
            System.out.println( tableauAuteurs[i] );
            showMessageDialog(null, tableauAuteurs[i], "Un auteur",
                INFORMATION_MESSAGE );
        }
        System.out.println("-----\n");
    } // public static void main
}

```

## Affichage de l'exécution précédente :



**Figure 15** : Les affichages de la nouvelle classe *Auteur*.

On vient de mettre en évidence que les collections simplifient significativement le code : on s'affranchit totalement de la taille à allouer.

➤ Revenons sur la méthode *elements()* de la classe *Vector* :

**elements**

```
public Enumeration<E> elements()
```

Returns an enumeration of the components of this vector. The returned `Enumeration` object will generate all items in this vector. The first item generated is the item at index 0, then the item at index 1, and so on.

**Returns:**

an enumeration of the components of this vector

**See Also:**

`Iterator`

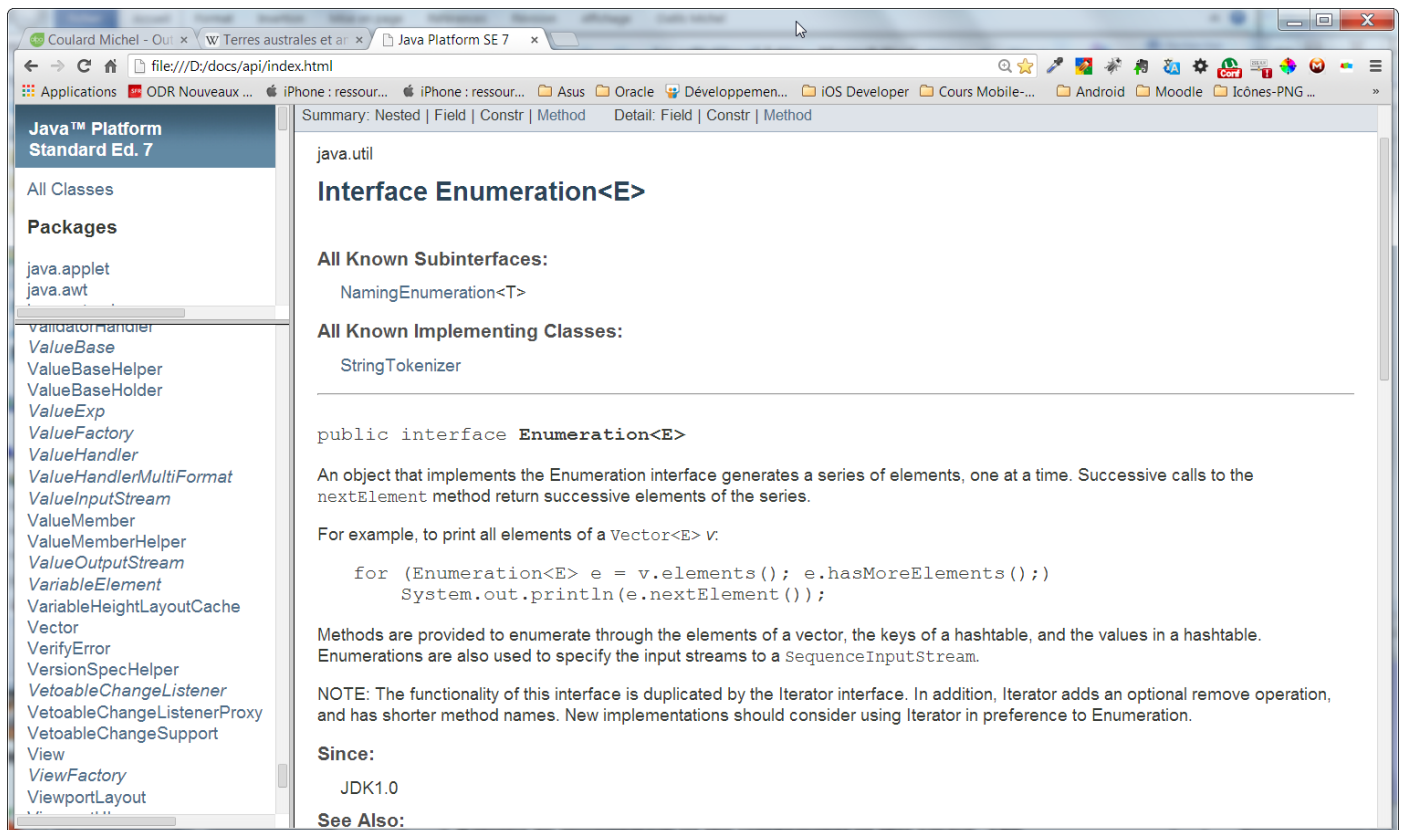
La documentation précise qu'elle renvoie une instance issue d'une classe qui implémente l'interface *Enumeration*.

La méthode *elements()* que l'on applique sur *livres* – instance de la classe *Vector<String>* - va ainsi nous permettre d'itérer sur la collection de *String* ( qui représente des livres ).

➤ Revenons maintenant quelques instants sur l'interface *Enumeration* dont voici le code :

```
public interface Enumeration {  
    boolean hasMoreElements () ; // Renvoie true s'il reste des éléments à parcourir  
    Object nextElement() ;       // Renvoie le prochain élément de la collection  
}
```

► Explorons la documentation de **J2 SE** liée à cette interface :



Que dit cette documentation ? : elle précise que tout objet qui implémente cette interface *Enumeration* renvoie un élément, un à un, par l'intermédiaire de la méthode *nextElement()*.

L'itération sur la collection se fait en testant s'il reste des éléments dans la collection grâce à la méthode *hasMoreElements()*.

► Pour terminer, examinons le code de la méthode *elements()* de cette interface *Enumeration* :

```

/**
 * Returns an enumeration of the components of this vector. The
 * returned Enumeration object will generate all items in
 * this vector. The first item generated is the item at index 0,
 * then the item at index 1, and so on.
 *
 * @return an enumeration of the components of this vector.
 * @see Enumeration
 * @see Iterator
 */
public Enumeration elements() {
    return new Enumeration() {
        int count = 0;
        public boolean hasMoreElements() {
            return count < elementCount;
        }
        public Object nextElement() {
            synchronized (Vector.this) {
                if (count < elementCount) {
                    return elementData[count++];
                }
            }
            throw new NoSuchElementException("Vector Enumeration");
        }
    };
}

```

**Figure 16** : Extrait de la méthode *elements()* de *Enumeration*.

On y voit que la méthode gère un compteur ( *count* ), le compare à un nombre d'éléments ( *elementCount* ) sur un tableau d'éléments représentant la collection ( *elementData* ).



## - 8 - Le dictionnaire ou *HashTable*

- Etudions maintenant un autre ancienne classe : *HashTable*, qui représente la notion de dictionnaire ou table de hachage ( qui fera l'objet d'un chapitre dédié ) mais qui a la particularité, comme la classe *Vector*, de ne pas limiter en taille le nombre d'objets déposés.
- On peut ajouter des objets en nombre : la collection se **redimensionne** pour accueillir les nouveaux éléments.
- Nous allons détailler la classe *HashTable* avec la même démarche : vous aurez ainsi une bonne représentation des apports des collections avant d'étudier plus en profondeur l'API.

### Définition d'un dictionnaire

- Un dictionnaire est une **collection d'éléments** qui s'agrandit dynamiquement en fonction des besoins. Les éléments d'un dictionnaire sont des **associations** entre une **clé** ( de type *Object* ) et une **valeur** ( elle aussi de type *Object* ). Comme un vrai dictionnaire papier, on peut faire un parallèle entre la **clé** et le **mot** puis la **valeur** et la **définition**.



Figure 17 : Grâce à la **clé**, on accède à la **valeur**.

- La classe *Hashtable* du package *java.util* permet de déclarer un dictionnaire :

```
Hashtable dictionnaire = new Hashtable() ;
```

- Etudions maintenant les opérations réalisables sur un *HashTable* :

### Ajout et suppression d'éléments dans un *HashTable*

*/\* Ajoute une association clé-valeur. Si la clé était déjà présente dans le dictionnaire, l'ancienne valeur est remplacée par la nouvelle.\*/*

- *put ( Object clé, Object valeur )*

*// Supprime l'association dont la clé est clé*

- *remove ( Object clé )*

### Accès aux éléments

- *get( Object clé ) // Renvoie la valeur de l'association dont la clé est indiquée par // le paramètre.*
- *size() // Renvoie le nombre d'associations déposées.*
- *isEmpty() // Renvoie true si l'objet ne contient aucune association.*

### Dictionnaires et énumérations

- **Itération** sur le contenu d'un dictionnaire :

Deux méthodes de *HashTable* renvoient un objet dont le type implémente l'interface *Enumeration* vue précédemment :

- *elements() // pour itérer sur la collection des **valeurs** d'un dictionnaire*
- *keys() // pour itérer sur la collection des **clés** d'un dictionnaire.*

## Un exemple d'utilisation

- Nous allons mettre en œuvre la notion de dictionnaire en établissant des couples dont la **clé** sera le **nom de l'auteur** et la **valeur** correspondante, **l'auteur** en lui-même en tant qu'instance ( et donc avec les ouvrages qu'il a écrits ).

```
package com.michel.entites;

import java.util.Enumeration;
import java.util.Vector;

/**
 * @author Michel-HP
 */
public class Auteur {

    private Vector<String> livres;
    private String nom;

    public Auteur(String nom, Vector<String> livres) {
        this.nom = nom;
        this.livres = livres;
    }

    public String getNom() {
        return nom;
    }

    public Vector<String> getLivres() {
        return livres;
    }

    public String toString() { // Renvoie une chaîne formatée
        String chaine = "-----\n" + nom;

        // Itération sur le vecteur livres
        Enumeration<String> enumeration = livres.elements();

        while (enumeration.hasMoreElements()) {
            chaine += "\n\t" + enumeration.nextElement();
        }

        return chaine;
    }

    public void ajouterLivre(String unLivre) {
        livres.addElement(unLivre);
    }
}
```

La classe Auteur

```
package com.michel.application;
```

```
import com.michel.entites.Auteur;
```

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

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

```
import java.util.Vector;
```

```
import static javax.swing.JOptionPane.*;
```

```
/**
```

```
 * @author Michel-HP
```

```
 */
```

```
public class Principale {
```

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

```
        // Définition de trois objets Auteur
```

```
        Auteur auteurFlaubert = new Auteur("Flaubert", new Vector<String>());
```

```
        Auteur auteurBalzac = new Auteur("Balzac", new Vector<String>());
```

```
        Auteur auteurHugo = new Auteur("Hugo", new Vector<String>());
```

```
        // Définition d'un dictionnaire (clé = Nom de l'auteur, valeur = l'auteur )
```

```
        Hashtable dicoAuteurs = new Hashtable();
```

```
        // Ajout de paires ( clé, valeur ) dans dicoAuteurs
```

```
        dicoAuteurs.put(auteurFlaubert.getNom(), auteurFlaubert);
```

```
        dicoAuteurs.put(auteurBalzac.getNom(), auteurBalzac);
```

```
        dicoAuteurs.put(auteurHugo.getNom(), auteurHugo);
```

```
        auteurFlaubert.ajouterLivre("Madame Bovary");
```

```
        auteurFlaubert.ajouterLivre("Salammbô");
```

```
        auteurFlaubert.ajouterLivre("L'Education Sentimentale");
```

```
        System.out.println("Le type de \"livres\" est : " +
```

```
            auteurFlaubert.getLivres().getClass().getCanonicalName());
```

```
        auteurHugo.ajouterLivre("Les misérables");
```

```
        auteurHugo.ajouterLivre("Notre-Dame de Paris");
```

```
        auteurHugo.ajouterLivre("La légende des siècles");
```

```
        auteurBalzac.ajouterLivre("Le Père Goriot ");
```

```
        auteurBalzac.ajouterLivre("Le colonel Chabert");
```

```
        auteurBalzac.ajouterLivre("La fille aux yeux d'or");
```

```
        auteurBalzac.ajouterLivre("L'Elixir de longue vie");
```

```
        // Déclaration d'un énumérateur sur les clés du dictionnaire
```

```
        Enumeration<String> enumeration = dicoAuteurs.keys();
```

```
        System.out.println("-----");
```

La classe manipulant un dictionnaire

```

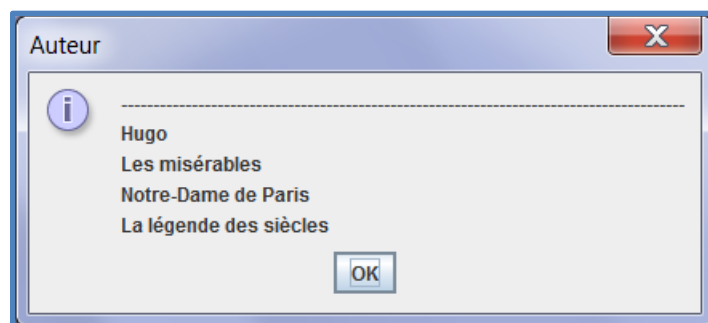
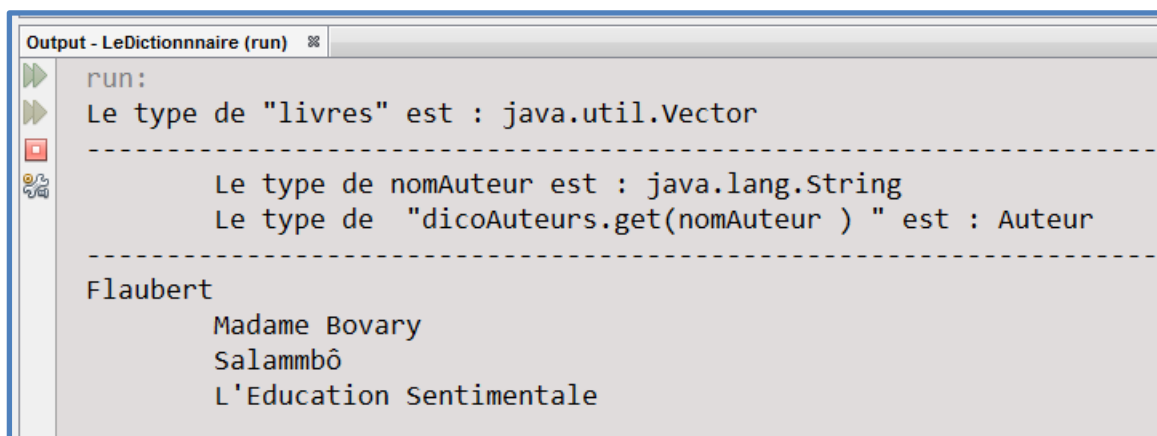
System.out.println("Le type de \"livres\" est : " +
    auteurFlaubert.getLivres().getClass().getCanonicalName());

while (enumeration.hasMoreElements()) {
    String nomAuteur = enumeration.nextElement();
    System.out.println("\tLe type de nomAuteur est : " + nomAuteur.getClass().getName());
    System.out.println("\tLe type de \"dicoAuteurs.get(nomAuteur)\" est : " +
        dicoAuteurs.get(nomAuteur).getClass().getSimpleName());

    System.out.println(dicoAuteurs.get(nomAuteur));
    showMessageDialog(null, dicoAuteurs.get(nomAuteur), "Auteur",
        INFORMATION_MESSAGE);
} // while
System.out.println("-----\n");
}
}

```

- Les affichages produits par *System.out.println(...)* dans l'extrait ci-dessus sont riches d'enseignements et montrent bien le type de la **clé** dans le dictionnaire : **String** et le type de la **valeur** en regard de la clé : **Auteur**.



**Figure 18** : Exploitation du dictionnaire.

- Dans l'état actuel des choses, le dictionnaire ayant été ainsi déclaré :

```
Hashtable dicoAuteurs = new Hashtable();
```

.... rien ne nous empêche de déposer un couple comme ceci :

```
dicoAuteurs.put(34, new java.util.Date());
```

Evidemment, lors de l'exploitation du dictionnaire par :

```
String nomAuteur = enumeration.nextElement();
```

... « Y'en a qui ont essayé : ils ont des problèmes » : à un moment, dans l'itération, le type retourné ne sera pas un *String* mais un *Integer*.

D'où problème. D'où également l'usage de la **généricité** qui nous amène à déclarer le dictionnaire de cette façon :

```
Hashtable<String,Auteur> dicoAuteurs = new Hashtable();
```

Dès lors, on a l'obligation de faire des `put ( String , Auteur )` sinon le compilateur ne va pas aimer.

Maintenant, le *HashTable* est typé et l'on ne pourra déposer **que** des couples de type ( *String*, *Auteur* ) : tout autre dépôt sera refusé par **Java** .

Nous vous invitons donc à le faire et à constater que ...

```
dicoAuteurs.put(34, new java.util.Date());
```

... est désormais interdit.

## Premier bilan

Les classes que nous avons découvertes jusqu'ici n'ont été étudiées qu'à **des fins pédagogiques** : bien qu'elles soient toujours disponibles, elles sont remplacées par d'autres collections, plus riches.

Ceci à partir de **Java 5**.

L'API des collections s'est considérablement **modernisée**, étoffée et offre de nombreux services, aussi pratiques que performants.

Nous allons les découvrir et constater que l'étude menée jusqu'à maintenant va vous aider à les comprendre.

Cette **nouvelle API** dédiée aux collections a été repensée, notamment en permettant la **généricité** et est extrêmement utilisée par les développeurs **Java**.

Voici un extrait de l'API des collections :

## - 9 - La hiérarchie de l'API *Collection*

### Extrait de la hiérarchie des collections en Java

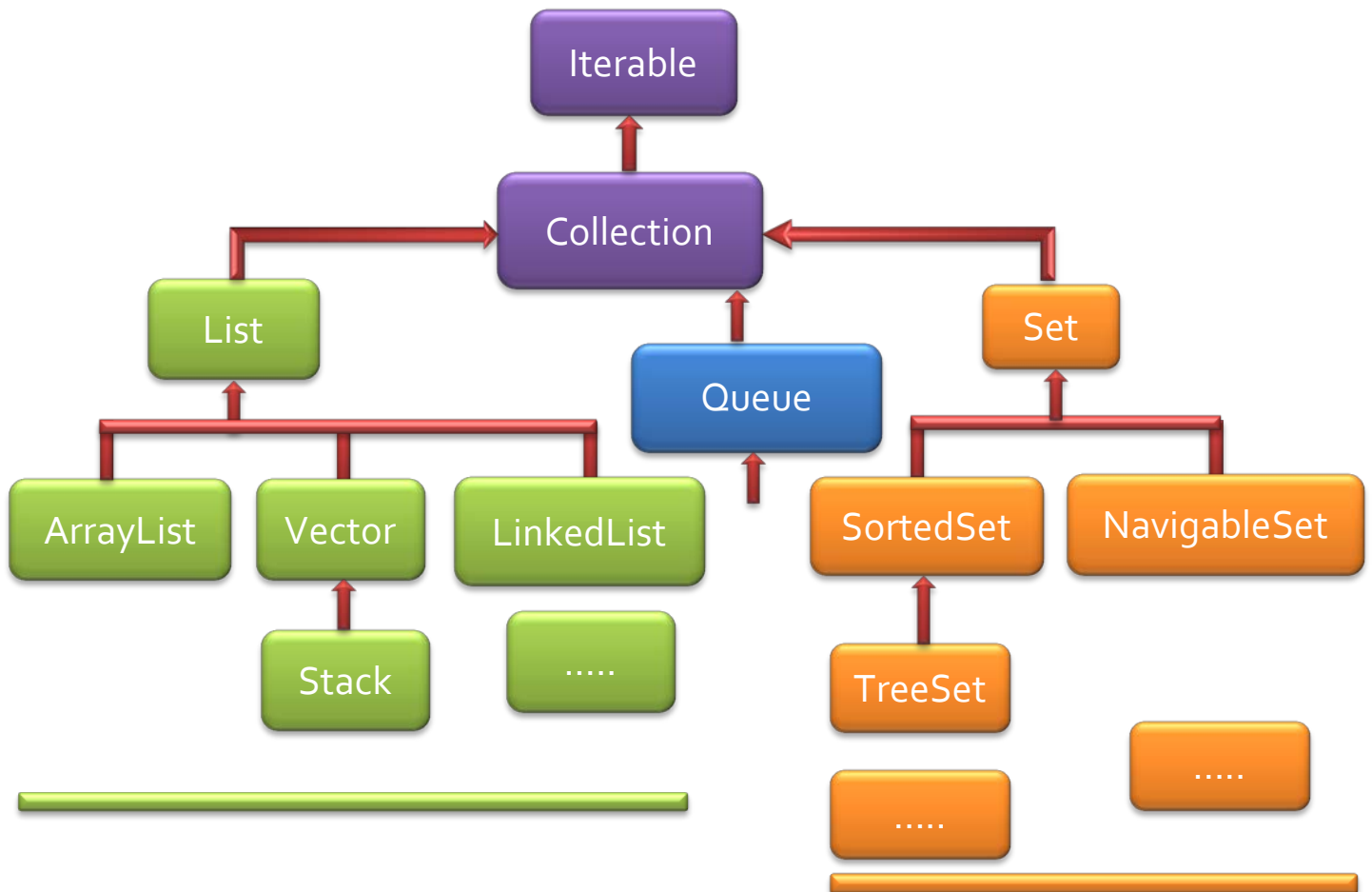


Figure 19 : Extrait de la hiérarchie de l'API *Collection*.

Cette hiérarchie (très incomplète) représente l'arborescence des collections. A sa base, on trouve l'interface *Iterable*.

Cette interface précise que tout objet issu d'une classe qui l'implémente peut être parcouru. D'où son nom. Il est nécessaire de parcourir la documentation pour bien appréhender l'API.



Il faut remarquer que :



- les 3 conteneurs figurant au sommet de la hiérarchie ci-dessus : *Collection*, *List* et *Set* sont des interfaces et donc non instanciables.
- Les classes *PileEntier*, *Vector*, *ListeChaine* que nous avons étudiées au début de ce tutoriel existent bien avec les classes respectives suivantes :

### Stack

- *push(E item)* pour empiler.
- *E pop()* pour dépiler.
- *boolean empty()* pour tester si la pile est vide.
- ....

### Vector

- déjà étudiée ...

### LinkedList

- *addFirst ( E e )*
- *addLast ( E e )*
- *add ( E e )*
- *add ( int index, E item )*

Là aussi, la documentation est très complète et est à consulter impérativement ....

## Les 4 interfaces principales

Le but recherché par les auteurs de l'API consiste à unifier l'utilisation des collections et à rendre leurs services associés **les plus universels possible**, d'où la présence massive des interfaces.

### Collection

Héritant de l'interface *Iterable*, vient l'interface *Collection* dont les services déclarés offrent, dans les classes concrètes d'implémentation, toutes les fonctionnalités inhérentes aux tableaux, listes, tables de hachage, ensembles, ....

### List

Dans la branche de gauche, l'interface *List* et ses descendantes proposent des classes destinées à collectionner des objets **selon une relation d'ordre** avec, pour le développeur, la possibilité d'insérer des objets où il le souhaite dans sa collection.

### Set

L'autre branche à partir de l'interface *Set* propose de collectionner des objets en rendant impossible le dépôt d'un même objet plus d'une fois.

Apparaît également dans la [figure 19](#) ci-dessus la classe *Stack*, qui représente la structure de pile **LIFO** que nous avons étudiée précédemment.

Il existe aussi maintenant l'interface *Queue* permettant de modéliser une file **FIFO**.

### Queue

L'interface *Queue*, héritant de *Collection*, propose grâce à ses classes d'implémentation de modéliser le concept de file.

## - 10 - L'interface *List*

### Définition

- L'interface *List*, qui hérite de l'interface *Collection*, gère une liste d'objets indexée. A chaque objet déposé, la liste lui associe un index.
- C'est grâce à cet index que l'on s'intéresse à chacun des objets.
- Les méthodes de *List* proposent des *méthodes d'accès* aux objets selon leur index et des *méthodes de parcours de la collection*.

### Méthodes de l'interface *List*

Type retour	Méthodes	Description
boolean	add ( E element )	ajoute l'élément en fin de liste.
void	add( int index, E element )	ajoute l'élément spécifié à la position indiquée par <i>index</i> .
boolean	addAll( Collection<? extends E> c )	ajoute les éléments de la collection spécifiée en fin de liste.
boolean	addAll( int index ,Collection<? extends E> c )	ajoute les éléments de la collection à l'emplacement spécifié par <i>index</i> .
void	clear()	supprime tous les objets de la liste.
boolean	contains( Object o )	retourne <b>true</b> si la liste contient l'élément spécifié.
boolean	containsAll ( Collection<?> c )	retourne <b>true</b> si la liste contient la collection spécifiée.
boolean	equals( Object o )	compare l'objet spécifié avec la liste.
E	get( int index )	retourne l'élément de la liste à la position spécifiée.
int	indexOf( Object o )	retourne l'index de la 1ere occurrence de l'élément spécifié ou -1 si la liste ne le contient pas.
boolean	isEmpty()	retourne <b>true</b> si la liste est vide.
Iterator<E>	iterator()	retourne un itérateur sur les éléments de la liste.
int	lastIndexOf ( Object o )	retourne l'index de la dernière occurrence de l'élément spécifié ou -1 si la liste ne le contient pas.
ListIterator<E>	listIterator()	retourne un itérateur de liste sur les éléments de la liste.
ListIterator<E>	listIterator( int index )	retourne un itérateur de liste sur les éléments de la liste à partir de la position spécifiée.
E	remove( int index )	supprime l'élément à la position spécifiée.
boolean	remove ( Object o )	supprime la 1ere occurrence dans la liste de l'élément spécifié.
boolean	removeAll( Collection<?> c )	supprime de la liste tous les éléments spécifiés dans la collection.
boolean	retainAll( Collection<?> c )	ne retient que les éléments de la liste qui sont contenus dans la collection spécifiée.
E	set ( int index, E element )	remplace l'élément à la position spécifiée dans la liste avec l'élément spécifié.
int	size()	retourne le nombre d'éléments dans la liste.
List<E>	subList( int fromIndex, int toIndex )	renvoie la partie de la liste entre l'index <i>fromIndex</i> spécifié inclus, et l'index <i>toIndex</i> exclus.
Object[]	toArray()	retourne un tableau contenant tous les éléments de la liste (du premier au dernier élément).
<T> T[]	toArray( T[] a )	Retourne un tableau contenant tous les éléments de la liste (du premier au dernier élément). Le type du tableau est celui du tableau spécifié.

## - 11 - Parcourir une collection

Si l'on regarde la hiérarchie des collections et la documentation particulière de l'interface *Collection*, on voit qu'elle déclare une méthode *iterator()* qui renvoie un **Iterator<E>**.

- Les classes implémentant cette interface disposent donc toutes de cet outil.
- Que dit la documentation ?

The screenshot shows the Java Platform Standard Ed. 7 documentation for the `java.util.Iterator<E>` interface. The page includes navigation tabs (Overview, Package, Class, Use, Tree, Deprecated, Index, Help) and a sub-navigation bar (Prev Class, Next Class, Frames, No Frames). The main content area displays the following information:

- Interface `Iterator<E>`**
- Type Parameters:**
  - `E` - the type of elements returned by this iterator
- All Known Subinterfaces:**
  - `ListIterator<E>`, `XMLEventReader`
- All Known Implementing Classes:**
  - `BeanContextSupport`, `BCSIterator`, `EventReaderDelegate`, `Scanner`

The interface definition is shown as:

```
public interface Iterator<E>
```

A description follows: "An iterator over a collection. `Iterator` takes the place of `Enumeration` in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

**Since:** 1.2

**See Also:** `Collection`, `ListIterator`, `Iterable`

... qu'*Iterator* est une interface typée ( `<E>` ) ...

... et qu'elle remplace l'interface *Enumeration* que nous avons étudiée au début de ce tutoriel. Cet outil a donc pour objectif, comme l'interface *Enumeration*, d'itérer sur les objets dans la collection.

Regardons maintenant les méthodes que propose cette interface : *hasNext()*, *next()*, *remove()*.

## Method Summary

### Methods

Modifier and Type	Method and Description
boolean	<code>hasNext()</code> Returns <code>true</code> if the iteration has more elements.
E	<code>next()</code> Returns the next element in the iteration.
void	<code>remove()</code> Removes from the underlying collection the last element returned by this iterator (optional operation).

Cela ne vous rappelle rien ? C'est exactement dans le même esprit qu'avec *Enumeration* que l'on va utiliser *Iterator* :

Enumeration	Iterator
<code>hasMoreElements()</code>	<code>hasNext()</code>
<code>nextElement()</code>	<code>next()</code>
	<code>remove()</code>

**Figure 20** : Comparaison *Enumeration* et *Iterator*

On remarque que *Iterator* propose un service supplémentaire : `remove()`. Cette méthode, comme son nom le laisse suggérer, permet de supprimer l'objet courant de la collection que l'itérateur désigne. Pratique.



Toute classe concrète issue de l'interface *Collection* est consultable par un *Iterator* parce qu'elle implémente *Iterable*.

Je vous invite à vérifier ceci en parcourant la documentation.

Le *scénario programmatique* de consultation d'une collection quelconque est donc :

Déclarer une instance de type *Iterator* et l'affecter avec le retour de la méthode `iterator()` dont toute classe, issue de *Collection*, dispose.

Exemple, en supposant que `maListe` référence un `ArrayList<String>` :

```
Iterator<String> itereur = maListe.iterator();
String element ;
while (itereur.hasNext()) {
    element = monIterateur.next();
    ....
}
```

## Un itérateur spécifique pour *List*

Comme indiqué dans le tableau précédent, l'interface *List* propose, en plus d'un *Iterator*, un itérateur spécifique de parcours : *ListIterator*. Ce *ListIterator* est obtenu par un appel à la méthode ... *listIterator()*.

Cette interface *ListIterator*, dérivée de *Iterator*, propose des services enrichis de parcours de liste dans les deux sens et des outils de **traitements des éléments** de la liste en cours de parcours.

Voici la liste des méthodes disponibles dans *ListIterator* :

Type retour	Méthodes	Description
void	add ( E element )	ajoute l'élément spécifié.
boolean	hasNext()	renvoie <b>true</b> s'il y a encore des éléments dans la liste dans le sens aller.
boolean	hasPrevious()	renvoie <b>true</b> s'il y a encore des éléments dans la liste dans le sens retour.
E	next()	retourne le prochain élément de la liste.
void	nextIndex()	Retourne l'index de l'élément suivant.
boolean	previous()	retourne l'élément précédent.
int	previousIndex()	retourne l'index de l'élément précédent.
void	remove()	supprime l'élément retourné par le dernier <i>next()</i> ou <i>previous()</i> .
void	set( E e )	remplace l'élément retourné par le dernier <i>next()</i> ou <i>previous()</i> par l'élément spécifié.

**Figure 21** : Les méthodes de l'interface *ListIterator*.

Les outils de traitements sur l'élément courant désigné par l'objet *ListIterator* sont particulièrement pratiques pour le développeur. Précisons-les :

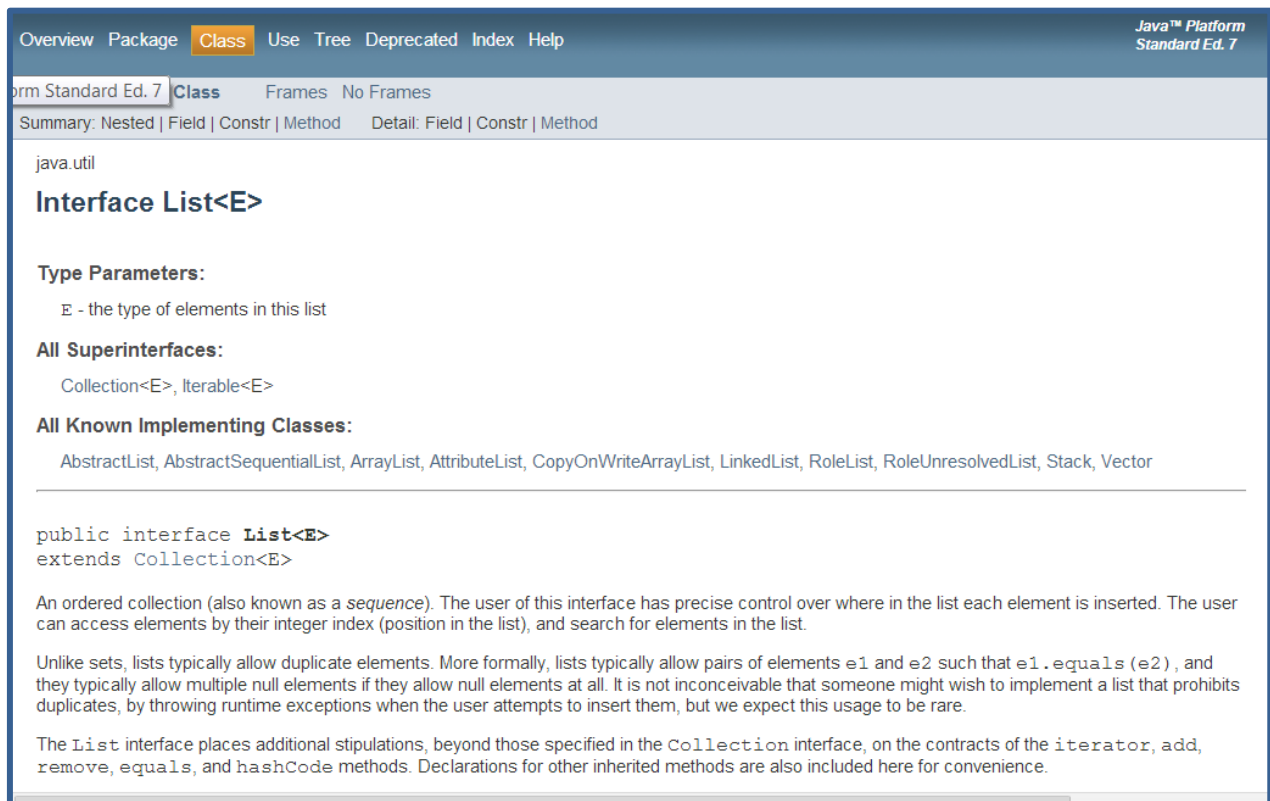
Méthodes	Description
add( E element )	<b>insère</b> l'élément à l'endroit désigné par le <i>ListIterator</i> , juste avant l'élément potentiellement renvoyé par <i>next()</i> . Un appel à <i>next()</i> à l'issue de ce <i>add(...)</i> renvoie non pas l'élément fraîchement déposé mais celui qui le suit. Méfiance.
remove ( E element )	<b>supprime</b> de la liste l'élément précédemment retourné par un <i>next()</i> ou un <i>previous()</i> .  Deux séquences d'activités interdites : <ol style="list-style-type: none"> <li>1. Effectuer deux appels consécutifs à <i>remove(...)</i> sans <i>next()</i> ou <i>previous()</i> entre les deux.</li> <li>2. Appeler <i>remove(...)</i> après avoir inséré ( <i>add</i> ) ou supprimé ( <i>remove</i> ) un élément.</li> </ol>
set ( E element )	<b>remplace</b> l'élément précédemment retourné par <i>next()</i> ou <i>previous()</i> par celui correspondant à l'argument.  Une séquence d'activités interdite : <ol style="list-style-type: none"> <li>1. Effectuer un appel à <i>set(...)</i> aussitôt après un appel à <i>add(...)</i> ou <i>remove (...)</i>.</li> </ol>

Figure 22 : Les méthodes sur l'élément courant de l'interface *ListIterator*.



## Les principales classes concrètes de type *List*

En examinant la documentation de l'interface *List*,



**Figure 23 :** L'interface *List* et ses classes d'implémentation.

... on voit toutes les classes qui implémentent *List*, notamment :

- *ArrayList* ( modélise un tableau ).
- *LinkedList* ( modélise une liste chaînée ).
- *Stack* ( modélise une pile ).
- *Vector*( modélise un tableau , déjà étudiée ).

## Caractéristiques communes de ces classes concrètes

- Les collections implémentées par les classes de type *List* peuvent contenir des objets identiques.
- Chaque élément mémorisé ayant une position déterminée - un index -, ces classes ressemblent aux tableaux Java mais **n'imposent aucune limite en taille**.

## Critères de choix de collections à instancier selon ses besoins

- Les tableaux (*ArrayList*, *Vector*, ...) permettent un accès très rapide à ses éléments : il suffit d'indiquer l'index de l'élément souhaité. En revanche, l'action consistant à ajouter un nouvel élément dans un tableau déjà constitué peut prendre beaucoup de temps : il faut décaler tous les éléments à partir de l'élément inséré. De surcroît, redimensionner un tableau parce qu'il a été sous-dimensionné à sa création est une activité coûteuse en temps.
- Concernant les *listes chaînées*, on ne peut pas aller directement à un élément particulier : il faut commencer le parcours par le début de la liste. Cela peut être long.
- Par contre : les éléments d'une liste *n'étant pas contigus* mais dispersés en mémoire, l'ajout d'un nouvel élément se fait rapidement : il suffit d'instancier un nouveau maillon et, par le biais d'affectation de références, l'insérer au début, à la fin ou entre deux éléments de la liste, selon ses besoins. C'est très rapide.
- Une liste chaînée n'a pas de taille a priori puisque il suffit de créer un nouveau maillon pour l'enrichir. Le problème de redimensionnement évoqué ci-dessus n'existe donc pas avec les listes chaînées.

Réalisons maintenant un exemple de mise en œuvre de l'interface *List* et des objets *Iterator* et *ListIterator*.

Vous reprendrez cet exemple dans un TP de la série les collections.

```

package com.michel.application;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

/**
 * @author Michel-HP
 */
public class Principale {

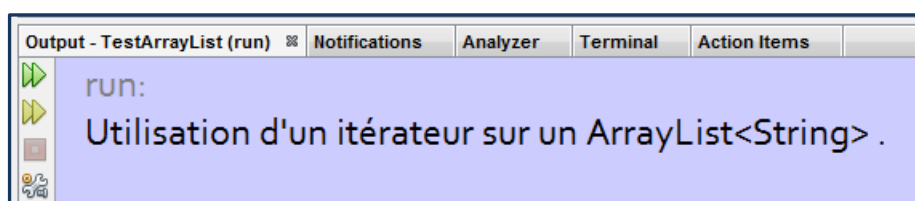
    public static void main(String[] args) {
        // Instanciation d'un ArrayList typé <String> et référencé sur List
        List maListe = new ArrayList<String>();

        // Déclaration et initialisation d'un tableau de String servant à remplir
        // l'ArrayList
        String[] tableau = {"Utilisation", "d'un", "itérateur", "sur", "un",
                           "ArrayList<String>"};

        Iterator iterator = maListe.iterator(); // Un simple itérateur

        for ( String mot : tableau ) {
            maListe.add(mot); // Chaque mot du tableau est inséré dans l'ArrayList
        }
        // Itération et affichage du contenu de l'ArrayList
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
        System.out.println(".\n");
        ...
    }
}

```



....

// Une autre ArrayList

List<String> listeMots = new ArrayList<String>();

// ... son remplissage ...

listeMots.add("L'interface List en ");

listeMots.add("Les ArrayList en ");

listeMots.add("Les Iterator en ");

listeMots.add("Les ListIterator en ");

afficherListe( listeMots ); // Affichage de l'ArrayList

modifierListe( listeMots ); // Sa modification

afficherListe( listeMots ); // De nouveau son affichage après modification.

}

public static void afficherListe(List<String> liste) {

System.out.println("Voici le contenu de l'ArrayList<String> contenant " +  
liste.size() + " éléments." : ");

Iterator iterator = liste.iterator();

while (iterator.hasNext()) {

System.out.println("\t- " + iterator.next() + " ");

}

System.out.println("");

}

public static void modifierListe(List<String> liste) {

// Le ListIterator permet de modifier la liste

ListIterator<String> monIterateur = liste.listIterator();

while (monIterateur.hasNext()) {

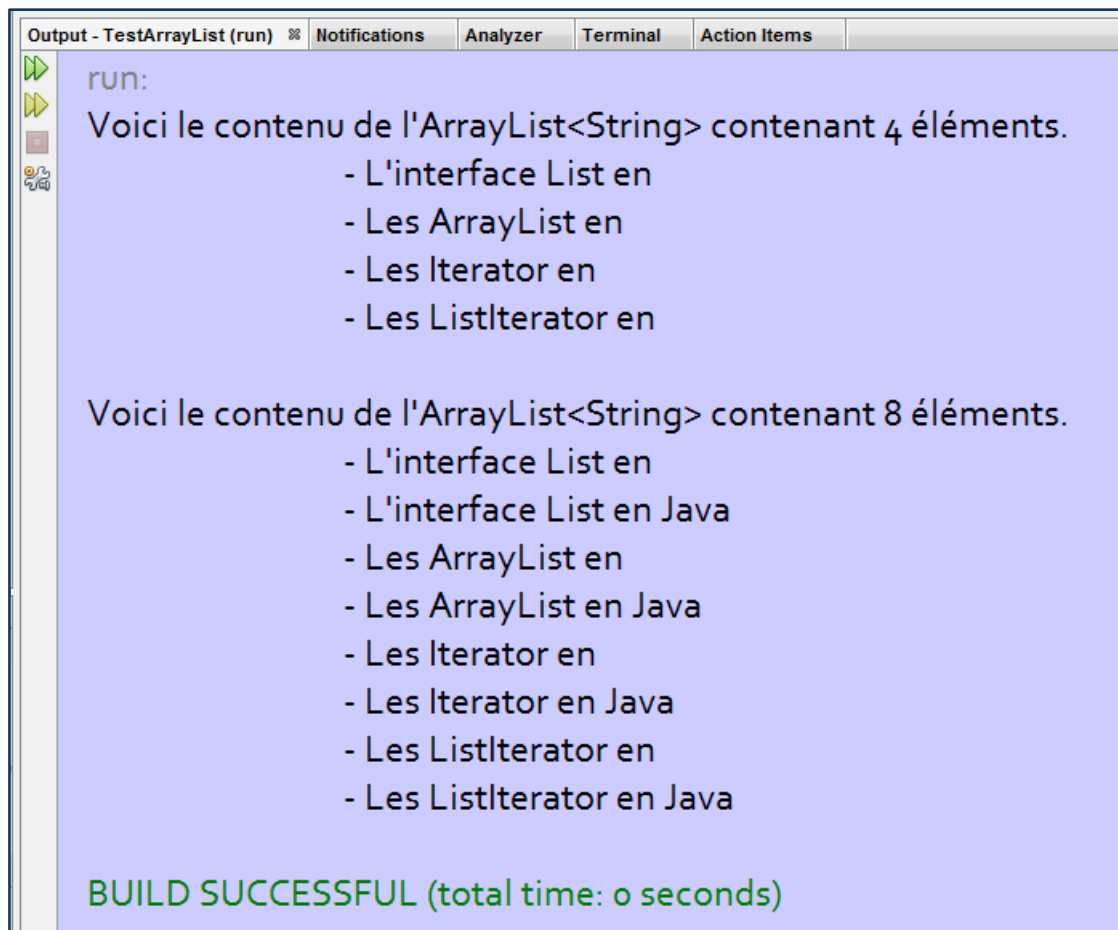
String element = monIterateur.next();

monIterateur.add(element + "Java"); // Ajout d'une chaîne à chaque tour

}

}

}



```
Output - TestArrayList (run)  Notifications  Analyzer  Terminal  Action Items
run:
Voici le contenu de l'ArrayList<String> contenant 4 éléments.
- L'interface List en
- Les ArrayList en
- Les Iterator en
- Les ListIterator en

Voici le contenu de l'ArrayList<String> contenant 8 éléments.
- L'interface List en
- L'interface List en Java
- Les ArrayList en
- Les ArrayList en Java
- Les Iterator en
- Les Iterator en Java
- Les ListIterator en
- Les ListIterator en Java

BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 24 : L'*ArrayList* avant et après modification.

On voit bien dans cet exemple l'intérêt du *ListIterator* : la possibilité d'agir sur la liste, ici en ajout, pendant la phase d'itération.

## - 12 - L'interface Set

### Définition

L'interface *Set*, qui hérite comme *List* de l'interface *Collection*, gère un ensemble d'objets **uniques**.

En d'autres termes : deux objets identiques ne peuvent pas coexister dans une classe implémentant l'interface *Set*.

Se pose maintenant la question légitime suivante : quand considère-t-on que deux objets sont identiques ? **Quel est le critère** ?

La réponse est la suivante et traduit une **propriété bien connue en Java** : deux objets sont égaux lorsque leur comparaison, réalisée avec la méthode *equals( Object o )*, renvoie **true**.



Il est important de noter que :

- le développeur peut établir le critère d'égalité sur la classe dont il est auteur ( redéfinition de *equals()* ).
- tout objet dispose inévitablement de la méthode *equals(Object o)* puisqu'elle est définie dans *Object*.

Cette propriété, spécifique donc à *Set*, implique que les méthodes consistant à déposer un nouvel objet dans une collection implémentant *Set* peuvent ne pas aboutir si cet objet existe déjà dans le conteneur.

### Les principales classes d'implémentation de Set

- Parmi les classes d'implémentation les plus usitées de *Set*, on trouve : *HashSet*, *LinkedHashSet* et *TreeSet*.
- Ces classes utilisent la notion de **code de hachage** pour les objets destinés à y être déposés.
- Ceci garantit la **cohérence** et les **performances** des classes issues de *Set*.
- Et ne fait que confirmer la propriété citée ci-dessus : deux instances égales doivent retourner le même code de hachage.

- La classe *HashSet* modélise un conteneur d'objets uniques basés donc sur leur code de hachage.
- La classe *LinkedHashSet* modélise une liste doublement chaînée.
- La classe *TreeSet* modélise un arbre binaire trié.

## Méthodes de l'interface *Set*

Type retour	Méthodes	Description
boolean	add ( E element )	ajoute l'élément dans l'ensemble s'il ne l'est pas déjà.
boolean	addAll( Collection<? extends E> c)	ajoute les éléments de l'ensemble s'ils ne le sont pas déjà.
void	clear()	supprime tous les objets de l'ensemble.
boolean	contains( Object o)	retourne <b>true</b> si l'ensemble contient l'élément spécifié.
boolean	containsAll ( Collection< ?> c)	retourne <b>true</b> si l'ensemble contient la collection spécifiée.
boolean	equals( Object o)	compare l'objet spécifié avec l'ensemble.
int	hashCode()	retourne le hashcode de l'ensemble.
boolean	isEmpty()	retourne <b>true</b> si l'ensemble est vide.
Iterator<E>	iterator()	retourne un itérateur sur les éléments de l'ensemble.
boolean	remove ( Object o)	supprime l'élément spécifié de l'ensemble s'il existe.
boolean	removeAll( Collection< ?> c)	supprime de l'ensemble tous les éléments spécifiés dans la collection.
boolean	retainAll(Collection< ?> c)	ne retient que les éléments de l'ensemble qui sont contenus dans la collection spécifiée.
int	size()	retourne le nombre d'éléments dans l'ensemble.
Object[]	toArray()	retourne un tableau contenant tous les éléments de l'ensemble.
<T> T[]	toArray( T[] a)	retourne un tableau contenant tous les éléments de l'ensemble. Le type du tableau est celui du tableau spécifié.



- Voyons un exemple d'ensemble *Set* mis en œuvre par *HashSet* avec des *String*.



Rappel : La classe *String* redéfinit bien les méthodes *equals()* et *hashCode()*. Le dépôt d'instances de *String* dans un *Set* est donc parfaitement envisageable.

Exemple de mise en œuvre d'un *HashSet* : dépôts et affichages.

```
public class Principale {

    public static void main(String[] args) {

        // Création d'un HashSet de String
        Set<String> lesChaines = new HashSet<String>();

        String[] tableau = {"Instanciation", "d'un", "HashSet", "et", "exploitation",
            "de", "ses", "propriétés", "et", "de", "ses", "méthodes"};

        String resultat = "";

        for (String mot : tableau ) {
            if ( lesChaines.add(mot)) {
                resultat+= "\nLe mot \"" + mot + "\" a été ajouté dans la collection
                    HashSet d'éléments uniques" ;
                System.out.println("Le mot " + mot + " a été ajouté dans la collection
                    HashSet d'éléments uniques");
            }
            else {
                System.out.println("Le mot \"" + mot + "\" est déjà présent dans la
                    collection HashSet d'éléments uniques");
                resultat+= "\n<b>Le mot \"" + mot + "\" est déjà présent</b> dans la
                    collection HashSet d'éléments uniques" ;
            }
            System.out.println("Il y a " + lesChaines.size() + " mots dans l'ensemble");
        }

        JOptionPane.showMessageDialog(null, resultat, "HashSet",
            JOptionPane.INFORMATION_MESSAGE);
        lesChaines.clear();
        System.out.println("Il y a " + lesChaines.size() + " mots dans l'ensemble");
    }
}
```

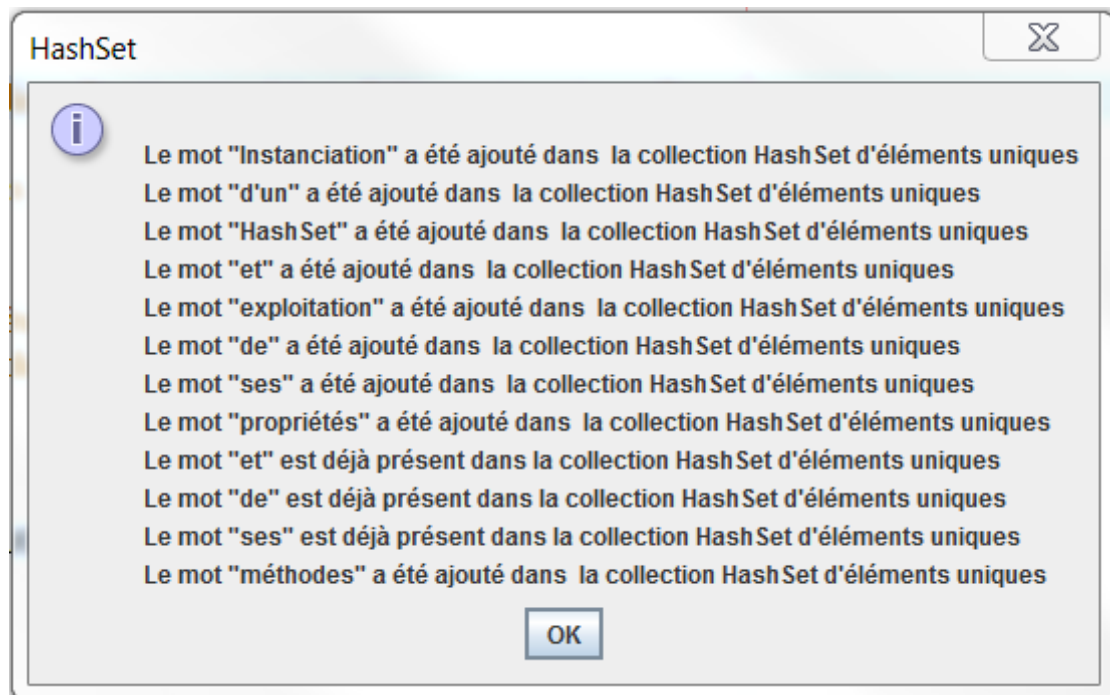


Figure 25 : Affichage des résultats suite aux dépôts.

## - 13 - L'interface *Map*

### Définition de la notion de table de hachage

Nous avons découvert par la pratique, au début de ce support, la notion de dictionnaire mise en œuvre par la classe *Hashtable*.

Cette classe *Hashtable* met en œuvre la notion de table de hachage qui, nous l'avons vu, représente un couple déposé sous la forme (**clé, valeur**).



Les méthodes permettant la gestion de cette structure de données sont les suivantes :

### Les méthodes d'une table de hachage

Méthodes	Description
put ( cle, valeur )	dépose l'association dans le conteneur.
get ( cle )	renvoie la valeur associée à la clé si elle existe.
remove ( cle )	supprime l'association si la clé existe.

### Caractéristiques d'une collection modélisant une table de hachage

- **Clé** et **valeur** sont des **objets**.
- Chaque clé est **unique** dans la collection.
- Une valeur peut exister plusieurs fois, avec des clés différentes.
- Il existe un **ensemble** pour :
  - les clés.
  - les valeurs.
  - les associations déposées.
- Il appartient, à chaque classe concrète, de fournir dans son interface des **méthodes d'accès à ces trois ensembles**.

## Extrait de la hiérarchie liée aux tables de hachage

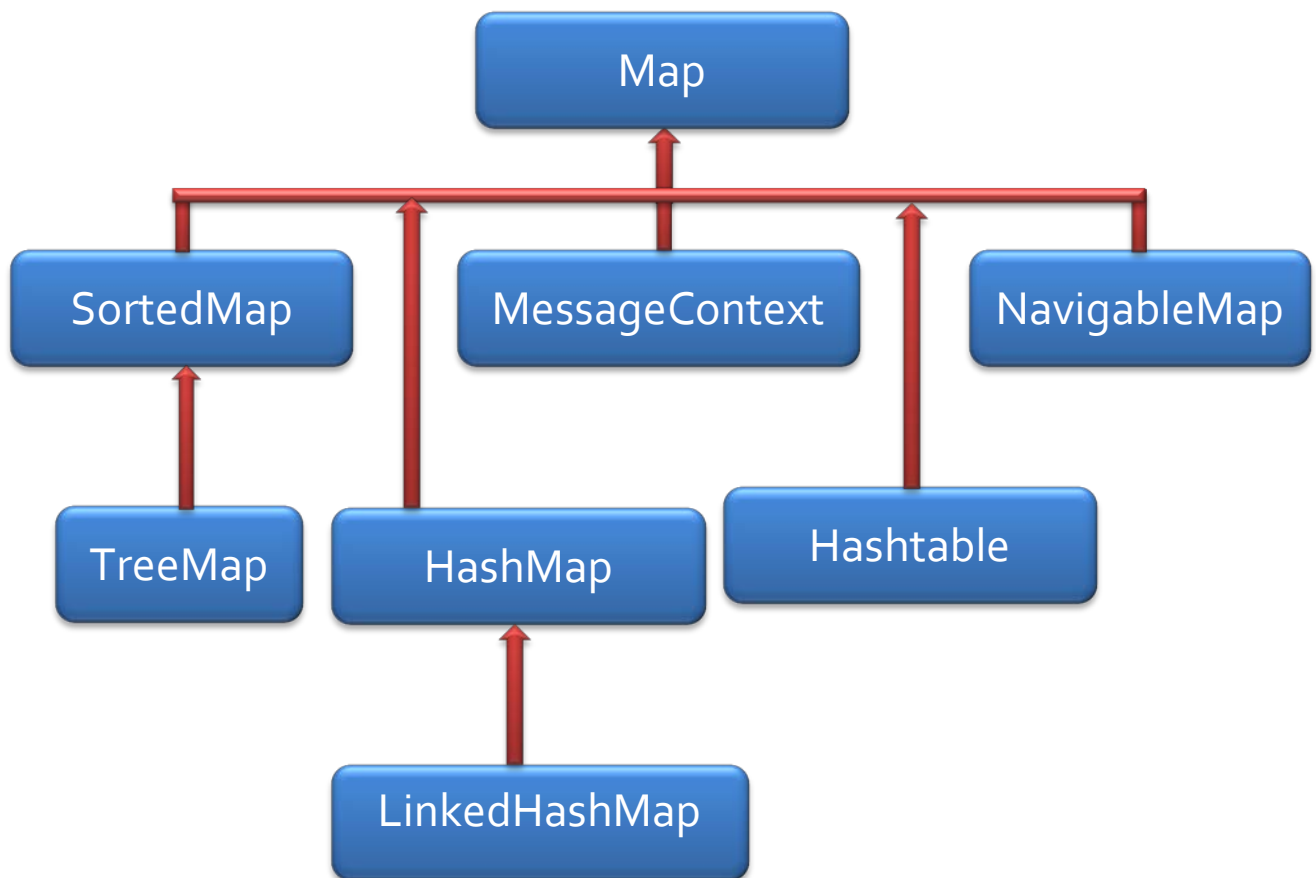


Figure 26 : Extrait de la hiérarchie des *Map*.

L'interface *Map* est à la base de cette hiérarchie. Elle propose les méthodes suivantes :

Type retour	Méthodes	Description
boolean	clear()	supprime toutes les associations de la <b>Map</b> .
boolean	containsKey( Object key )	retourne <b>true</b> si la <b>Map</b> contient une association avec <i>key</i> .
boolean	containsValue ( Object valeur )	retourne <b>true</b> si la <b>Map</b> contient au moins une association avec la <i>valeur</i> .
Set<Map.Entry<K,V>>	entrySet()	retourne un <b>Set</b> des associations.
boolean	equals(Object o)	compare la <b>Map</b> avec <i>o</i> .
V	get( Object key )	retourne la valeur de l'association avec la clé spécifiée ou <b>null</b> si elle n'existe pas.
int	hashCode()	retourne la valeur de hash code de la <b>Map</b> .
boolean	isEmpty()	retourne <b>true</b> si la <b>Map</b> est vide.
Set<K>	keySet()	retourne un <b>Set</b> des clés de la <b>Map</b> .
<V>	put( K key, V value)	dépose une association dans la <b>Map</b> .
ListIterator<E>	putAll( Map< ? extends K, ? extends V> m ), V value)	copie toutes les associations de la <b>Map</b> spécifiée.
V	remove( Object key )	supprime l'association liée à <i>key</i> si elle existe.
int	size()	retourne le nombre d'associations dans la <b>Map</b> .
Collection<V>	values()	retourne une <b>Collection</b> des valeurs de la <b>Map</b> .

Figure 27 : Les méthodes de *Map*.

## - 14 - Une interface dérivée de *Map* : *SortedMap*

Dans la hiérarchie de la [figure 26](#), on peut voir que *SortedMap* est une interface dérivée de *Map*. Que propose-t-elle en plus ?

Comme son nom l'indique, elle collectionne les associations selon une relation d'ordre. En d'autres termes, les associations déposées **sont triées** selon les clés.

### Les moyens techniques pour établir le tri sur les clés

Comment établir cette notion de relation d'ordre sur les clés ? Deux moyens programmatiques sont envisageables :

- Les clés des associations sont des implémentations de *Comparable*, et, à ce titre, le code de la méthode *compareTo (...)* décrit le scénario de tri.
- Lors de l'instanciation de la *SortedMap*, on fournit une instance d'un *Comparator* ( un constructeur dans les classes d'implémentation le permettent ).

## Les méthodes de l'interface *SortedMap*

Type retour	Méthodes	Description
Comparator<? super K>	comparator()	retourne l'éventuel comparateur transmis à la <b>SortedMap</b> .
Set<Map.Entry<K,V>>	entrySet()	retourne un <b>Set</b> des associations.
K	firstKey()	retourne la plus petite clé de la <b>SortedMap</b> .
SortedMap<K,V>	headMap ( K toKey )	retourne le sous-ensemble de la <b>SortedMap</b> dont les clés sont inférieures à <i>toKey</i> .
Set<K>	keySet()	retourne un <b>Set</b> des clés de la <b>SortedMap</b> .
K	lastKey()	retourne la plus grande clé de la <b>SortedMap</b> .
SortedMap<K,V>	subMap( K fromKey, K toKey )	retourne le sous-ensemble de la <b>SortedMap</b> dont les clés sont comprises entre <i>fromKey</i> et <i>toKey</i> .
SortedMap<K,V>	tailMap ( K fromKey )	retourne le sous-ensemble de la <b>SortedMap</b> dont les clés sont supérieures à <i>fromKey</i> .
Collection<V>	values()	retourne une <i>Collection</i> des valeurs de la <b>SortedMap</b> .

**Figure 28** : Les méthodes de *SortedMap*.

## - 15 - L'interface *Map.Entry*

Dans les tableaux de description des méthodes précédents, vous avez remarqué que l'on fait référence au type *Map.Entry*. A quoi correspond-t-il ?

C'est tout simplement la représentation objet d'une association dans la *Map*. C'est ce que l'on appelle une entrée dans un *Map*.

- Une entrée ( *Map.Entry* ) est donc un couple (K,V ).

Type retour	Méthodes	Description
boolean	equals(Object o)	compare l'entrée avec o.
K	getKey()	retourne la <b>clé</b> de l'entrée.
V	getValue()	retourne la <b>valeur</b> de l'entrée.
int	hashCode	retourne le <i>hashCode</i> de l'entrée.

- La méthode *entrySet()* de l'interface *Map* renvoie un *Set* de *Map.Entry* : il ne reste plus qu'à exploiter ce *Set* pour s'intéresser aux entrées.
- Vous allez dans un des exercices associés exploiter ces principes de façon à construire une *HashMap* des auteurs ( **clé** = Nom de l'auteur, **valeur** = auteur lui-même )

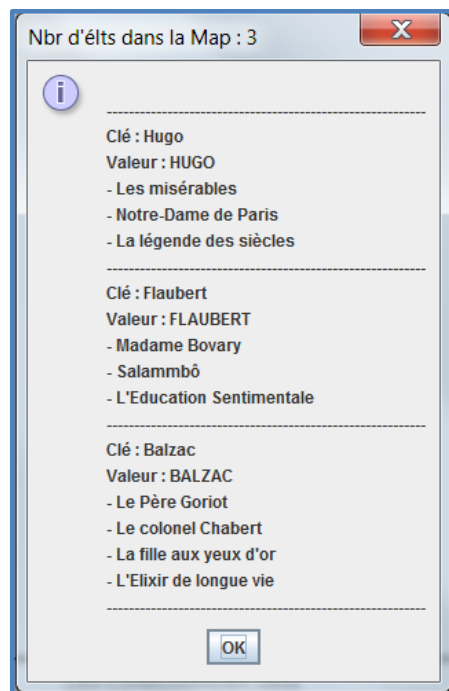


Figure 29 : Le contenu de la *HashMap* que vous construirez.



## - 16 - La généricité

### Historique

La **généricité** est le mécanisme le plus demandé par les développeurs **Java** depuis son origine. Elle est apparue avec Java 5 et est semblable aux templates de C++.

### Définition

Intégrée donc depuis Java 5.0 et utilisée par les classes de collection, elle permet au développeur de spécifier une classe différente de *java.lang.Object* comme type des éléments stockés. Son but essentiel est d'éviter l'utilisation de l'opérateur de **cast**.

Une classe **générique** est donc une classe pouvant être utilisée pour des objets de différentes classes. En quelque sorte, une classe **générique** est typée avec des paramètres qui ne sont pas des variables mais des « **types classe** ».

Comme nous l'avons rencontré précédemment, il s'avère que les collections sont parfaitement adaptées à l'usage de cette **généricité**.

La classe des éléments caractérisant les collections est spécifiée entre les symboles **<** et **>** qui suivent la classe de collection.

### Exemples

1. **ArrayList<Integer>** représente une collection de classe **java.util.ArrayList** dans laquelle seuls des objets de classe **java.lang.Integer** pourront être ajoutés. La généricité simplifie alors la consultation des éléments d'une collection en évitant de faire appel à l'opérateur de **cast**.

La généricité empêche également le dépôt d'objet d'un type autre que le type déclaré lors de l'instanciation de la collection.

2. **HashMap<String,Auteur>** lesAuteurs = new **HashMap<String,Auteur>**();

Les collections génériques peuvent aussi être utilisées dans une **structure itérative** pour énumérer un à un les éléments d'un ensemble.

## Méthode générique

Java autorise non seulement de typer une classe avec un ou plusieurs types classes pour les rendre « génériques » mais permet aussi de fournir des arguments **Type** dans les paramètres formels de méthodes, ainsi que pour leur retour.

Exemple : Développons une classe d'utilitaires pour les tableaux dotée de 3 méthodes :

- `getPremier ( E[] elt )` : renvoie le **premier élément** d'un tableau quel que soit le type de ses éléments.
- `getDernier ( E[] elt )` : renvoie le **dernier élément** d'un tableau quel que soit le type de ses éléments.
- Et enfin `getType ( E[] tableau )` : renvoie une chaîne de caractères correspondant au type des éléments du tableau reçu.

Ces méthodes étant des utilitaires : elles apparaissent comme **méthodes de classes**.

```
package com.michel.ouutils;

/**
 * @author Michel-HP
 */
public class OutilsTableaux {

    public static <E> E getPremier( E[] tableau) {
        return tableau[0];
    }

    public static <E> E getDernier( E[] tableau) {
        return tableau[tableau.length - 1];
    }

    public static <E> String getType ( E[] tableau) {
        return tableau.getClass().getSimpleName();
    }

    public static <E> void afficher ( E[] tableau) {
        for ( int i = 0 ; i< tableau.length ;i++){
            System.out.print("-" + tableau[i] );
        }
        System.out.println("");
    }
    ....
}
```

- On constate que syntaxiquement, il suffit de mentionner dans l'entête de la méthode l'emploi d'un type de données. Ici : **<E>**
- Ensuite, on utilise ce type **E** comme n'importe quel type de données.

Exemple :

```
public static <E> E getPremier( E[] tableau) { ... }
```

`getPremier(...)` reçoit un tableau d'une dimension de type **E** et renvoie une variable également de type **E**.

- Regardons son emploi dans la méthode *main* : ( page suivante )



Vous pouvez voir des appels à d'autres méthodes utilitaires que *getPremier(...)* dans cet exemple : c'est vous qui les coderez dans les exercices.

Vous aurez ainsi à écrire les méthodes *getDernier()*, *getType()*, *trierTabEntier()* ....

// Déclaration d'un tableau de caractères initialisé.

```
Character tableauCaracteres[] = { 'J', 'a', 'v', 'a' } ;
```

// Déclaration d'un tableau d'Integer.

```
Integer tableauInteger[] = new Integer[10];
```

// Remplissage aléatoire du tableau d'Integer.

```
for ( int i = 0 ; i < tableauInteger.length ; i++ ) {  
    tableauInteger[i] = new Integer( (int)(Math.random() * 10));  
}
```

```
OutilsTableaux.afficher(tableauCaracteres);
```

```
OutilsTableaux.afficher(tableauInteger);
```

```
System.out.println("Le premier élément dans le tableau de caractères est : " +  
    OutilsTableaux.<Character>getPremier(tableauCaracteres));
```

```
System.out.println("Le dernier élément dans le tableau de caractères est : " +  
    OutilsTableaux.<Character>getDernier(tableauCaracteres));
```

```
System.out.println("Le premier élément dans le tableau d'entiers est : " +  
    OutilsTableaux.<Integer>getPremier(tableauInteger));
```

```
System.out.println("Le dernier élément dans le tableau d'entiers est : " +  
    OutilsTableaux.<Integer>getDernier(tableauInteger));
```

```
System.out.println("Le type du tableau \"tableauCaractères\" est : " +  
    OutilsTableaux.<Character>getType(tableauCaracteres));
```

```
System.out.println("Le type du tableau \"tableauInteger\" est : " +  
    OutilsTableaux.<Integer>getType(tableauInteger));
```

```
System.out.println("\nTableau d'entiers non trié");
```

```
OutilsTableaux.afficher(tableauInteger);
```

// OutilsTableaux.<Integer>trierTabEntier(tableauCaracteres);

```
OutilsTableaux.<Integer>trierTabEntier(tableauInteger);
```

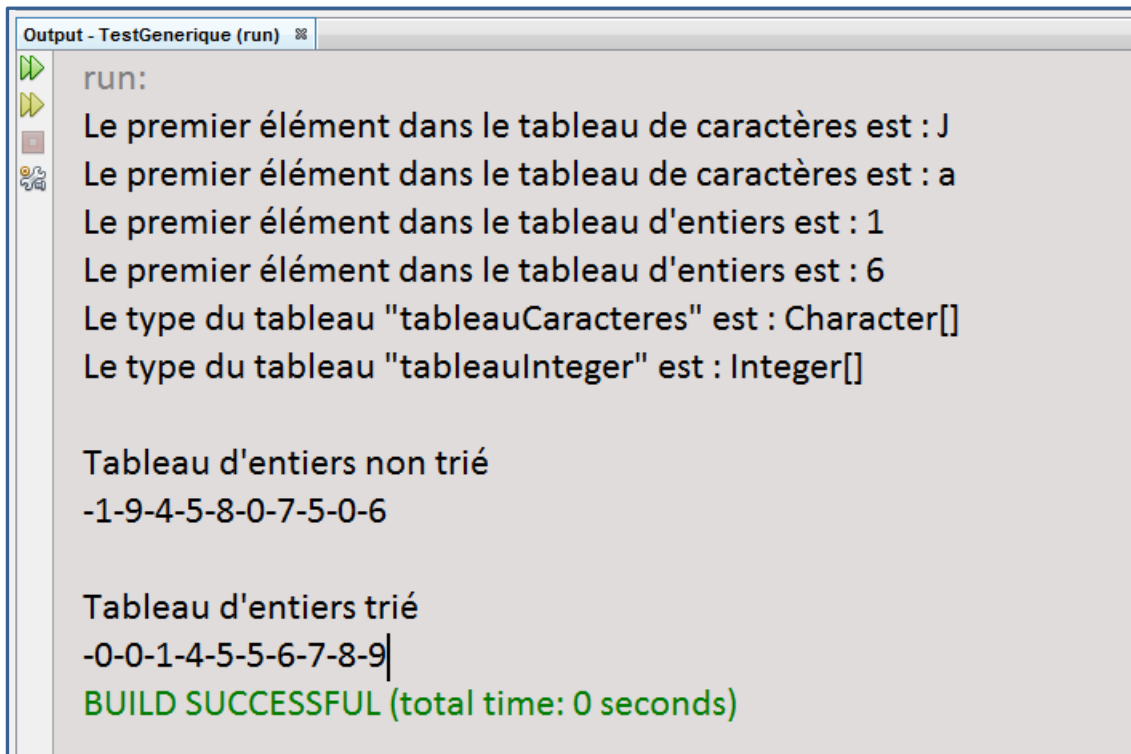
```
System.out.println("\nTableau d'entiers trié");
```

```
OutilsTableaux.afficher(tableauInteger);
```

On constate qu'il suffit de préciser le type souhaité lors de l'appel de la méthode :

```
OutilsTableaux.<Character>getType(tableauCaracteres));
```

L'exécution du code de *main* produit les affichages suivants :



```
run:
Le premier élément dans le tableau de caractères est : J
Le premier élément dans le tableau de caractères est : a
Le premier élément dans le tableau d'entiers est : 1
Le premier élément dans le tableau d'entiers est : 6
Le type du tableau "tableauCaracteres" est : Character[]
Le type du tableau "tableauInteger" est : Integer[]

Tableau d'entiers non trié
-1-9-4-5-8-0-7-5-0-6

Tableau d'entiers trié
-0-0-1-4-5-5-6-7-8-9|
BUILD SUCCESSFUL (total time: 0 seconds)
```

Successivement on trouve :

1. Affichage du tableau de caractères 1 à 1
2. Affichage du tableau d'entiers 1 à 1
3. Affichage du **premier** élément du tableau de caractères : 'J'
4. Affichage du **dernier** élément du tableau de caractères : 'a'
5. Affichage du **premier** élément du tableau d'entiers : '1'
6. Affichage du **dernier** élément du tableau d'entiers: '6'
7. Affichage du type du tableau de caractères : Character[]
8. Affichage du type du tableau d'entiers : Integer[]
9. Affichage du tableau d'entiers 1 à 1 avant le tri.
10. Affichage du tableau d'entiers 1 à 1 après le tri.

Nos méthodes génériques de la classe *OutilsTableaux* fonctionnent parfaitement.

## Type générique contrôlé

Il est possible en **Java** d'obliger un type générique à être **un type classe particulier** (exactement ou une de ses descendances).

Reprenons l'exemple des utilitaires dédiés aux tableaux.

- Ajoutons une méthode permettant de trier un tableau d'**Integer** : *trierTabEntiers*.
- Dans un premier temps, on se contente d'afficher une chaîne avant de trier.
- On veut s'assurer que le tableau est bien est tableau d'**Integer**.

```
public static <E extends Integer> void trierTabEntier ( E[] tableauInteger) {  
    System.out.println("trierTabEntier");  
}
```

- L'expression **<E extends Integer>** ci-dessus ne traduit pas ici la notion d'héritage : elle signifie que le type **E** doit être de type **Integer** ou toute sous-classe d'**Integer**.

Toute tentative d'utiliser ( appeler ) *trierTabEntier* autrement qu'en transmettant un **Integer** ( ou dérivée , qui n'existe pas ) sera rejetée par **Java** .

```
OutilsTableaux.<Integer>trierTabEntier(tableauInteger); // OK
```

```
OutilsTableaux.<Character>trierTabEntier(tableauCaracteres); // Interdit
```

Il vous appartiendra dans le T.P. d'écrire le code de la fonction *trierTabEntier* :

```
public static <E extends Integer> void trierTabEntier ( E[] tableauInteger) {  
    ..... // A vous de jouer  
}
```

# - 16 - Les classes **Arrays** et **Collections**

## Présentation

Les classes **Arrays** et **Collections** sont des classes à vocation d'utilitaires. Elles fournissent toutes les deux une gamme d'outils destinés à manipuler les tableaux et les collections avec toutes les activités que l'on peut imaginer : tri, recherche, copie, extraction, suppression, remplissage, transformation, ....



## La classe **Arrays**

Comme on peut le supposer, cette classe offre tous les services décrits ci-dessus pour tous les tableaux, quels qu'ils soient. Ces services sont évidemment fournis sous la forme de méthodes de classes.

Ainsi, pour trier un tableau d'objets de tout type, on fera :

```
Arrays.sort ( tableauVoitures ) ;
```

Naturellement, il faut que le type des éléments du tableau soient des *Comparable*.

## La classe **Collections**

De même, cette classe offre tous les services décrits ci-dessus pour les collections, quelles qu'elles soient. Ces services sont évidemment fournis sous la forme de méthodes de classes.

Il est important de noter que l'on manipule là des collections **au sens objet**, et donc : tous les outils annexes aux collections que nous avons étudiés ( itérateur , tri, ... ) sont présents et prennent tous leurs sens.

## Conseils

Tout développeur **Java** ( **SE** ou **EE** ) a inévitablement besoin de manipuler des collections. La connaissance des outils offerts par ces classes passe un examen et une étude approfondis de ces deux classes en exploitant la **Javadoc** fournie par le **JDK**.

C'est une activité incontournable qui vous fera gagner un temps précieux par la suite dans vos développements.

## - 18 - Stratégie pour utiliser une collection

Dans cette forêt de classes, une **question légitime** qu'un développeur se pose est la suivante :

« Quelle collection dois-je utiliser pour mon programme ? »

La réponse la plus juste que l'on peut donner à cette interrogation est la suivante :

« **Ça dépend ....** »

Voici quelques critères qu'un développeur doit évaluer chronologiquement pour déterminer et motiver ses choix concernant la collection adaptée :

- A partir du sommet de la hiérarchie des collections, partir sur la bonne branche, c'est-à-dire choisir l'interface initiale ( **Collection**, **List**, **Map** ).
- Dois-je gérer des objets par **indexation**, les objets doivent-ils être **uniques**, doivent-ils être **triés** dans la collection ? Est-ce que les objets collectionnés sont **atomiques** ou doivent-ils être **associés à une clé** pour les atteindre ?
- En fonction des réponses liées aux questions ci-dessus, choisir la bonne classe d'implémentation : « j'ai besoin d'une liste chaînée, d'un simple tableau, d'une pile, d'un ensemble trié ... »
- **Instancier la bonne collection** et la désigner avec le **type interface choisi**. Pour quelles raisons ?

Tout simplement parce que si l'on change ultérieurement **le type de la classe d'implémentation** concernée, le code qui utilise cette collection n'est pas impacté. Par compatibilité, toute classe concrète, quelle qu'elle soit, est de **type de l'interface** qu'elle implémente.

Exemple :

```
List maListe = new ArrayList<String>();  
.....  
// Changement de classe d'implémentation :  
maListe = new LinkedList<String>();  
.....
```



## - 19 - Résumé ....

# Classifications des collections

- Les collections implémentées par les classes `java.util.ArrayList` et `java.util.LinkedList` sont conçues pour ajouter des éléments les uns à la suite des autres dans un ensemble ordonné. `LinkedList` permet aussi d'ajouter des éléments n'importe où dans la collection. Elles peuvent contenir des éléments identiques. Chaque élément mémorisé ayant une position déterminée, ces classes ressemblent aux tableaux `Java` mais n'imposent aucune limite en taille.
- Les ensembles implémentés par les classes `java.util.HashSet` et `java.util.TreeSet` ne peuvent pas contenir des éléments identiques. La classe `java.util.TreeSet` stocke les éléments de son ensemble dans l'ordre ascendant.
- Les ensembles implémentés par les classes `java.util.HashMap` et `java.util.TreeMap` utilisent une clé pour accéder aux éléments au lieu d'un indice entier. La classe `java.util.TreeMap` stocke les éléments de son ensemble dans l'ordre ascendant des clés.
- Les ensembles implémentés par `java.util.Stack` modélisent le concept de pile.

NB : Les tableaux étant typés ( grâce au type précisé lors de leur déclaration) sont plus simples à programmer et moins gourmands en mémoire, ils restent très souvent utilisés notamment pour les ensembles dont la taille est connue à l'avance.

# Copyright

► **Chef de projet (responsable du produit de formation)**

PERRACHON Chantal, DIIP Neuilly-sur-Marne

► **Ont participé à la conception**

COULARD Michel, CFPA Evry Ris-Orangis

► **Réalisation technique**

COULARD Michel, CFPA Evry Ris-Orangis

► **Crédit photographique/illustration**

Sans objet

► **Reproduction interdite / Edition 2014**

**AFPA Février 2014**

**Association nationale pour la Formation Professionnelle des Adultes**

13 place du Général de Gaulle – 93108 Montreuil Cedex

[www.afpa.fr](http://www.afpa.fr)