Collections

Java Standard Edition

Objectifs

- **Expliquer les** collections .
- **Enumeration dans les** collections.
- **Utilisation et** choix de la meilleure solution.

Plan du cours

Présentation collection?

utilité ? Utilisation ?

collection de taille invariable → les tableaux

1D, 2D, classe utilitaire **Arrays**

collection de taille variable → les collections

Taille dynamique classe utilitaire **Collections**

Généricité

Parcours Iterators Itération sur une collection.

Comparable & Comparator : Tri des éléments dans une collection.

Interfaces & implementations.

Plusieurs selon vos besoins

Collection, Map, List ArrayList Vector , Set hashSet

Utilisation et choix de la meilleure solution

Collections

PRESENTATION

Présentation

Collection :
- organisation logique contenant plusieurs informations.
- permet de globaliser, de simplifier et d'unifier les traitements réalisables sur les informations contenues.

Collection séquentielle:
- l'emplacement pour ranger les données n'est pas imposée.
- la nouvelle donnée dans la structure s'insère
  au début,
  à la fin,
  entre 2 données.

Une variable de type tableau référence un collection séquentielle

permet d’unifier les traitements

La généralisation est la possibilité de considérer un objet pour plus général qu'il n'est, le temps d'un traitement (voire cours polymorphisme)
Dans une collection les objets présents sont généralisés ce qui suffit pour certaines opérations comme :

- obtenir le nombre d' éléments de la collection,
- savoir si un objet s'y trouve,
- ajouter (sans contrainte sur la position) et enlever un objet

Tout objet particulier peut devenir membre d'une collection, et il ne « perd » rien en le devenant, mais la vue qu'on en a en tant que membre de la collection est la plus générale qui soit.

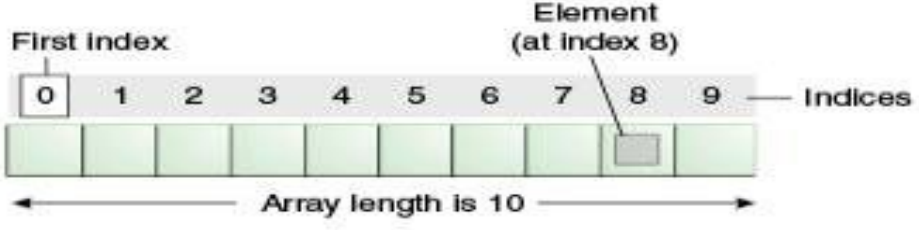
collection avec taille invariable

→ LES TABLEAUX

Définition

A une seule variable (référence à un tableau) correspond plusieurs informations de **même type**

Les valeurs se rangent dans des « cases » identifiées par un indice commençant à 0;



Taille Type

- **taille non dynamique**,

Un **tableau** a une **taille invariable**, qui est définie à l'instanciation de l'objet et qui ne peut plus être modifiée ensuite.

Comme la taille est fixe , attention à ne pas dépasser la dimension du tableau

- **tableau homogène**

Les éléments d'un tableau sont d'un type donné :

- primitif → `int myTab[] ;`

- objet → `Point [] tabPts ;`

Tableau

Déclaration/ initalisation → 0

Un **tableau** est une instance d'une classe créée dynamiquement

```
int arrayOfInteger[] = new int[10];
// Array of 10 items of int type initialize to zero.
```

L'opérateur **new** alloue l'emplacement nécessaire pour stocker les informations représentées par arrayOfInteger.

Les informations sont initialisées à 0 (type primitif int)

Déclaration/ initalisation → null

- Avec un tableau d'objet toute référence au sein du tableau est valorisée à *null*.

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

Déclaration & initialisation

- sans l'opérateur **new :** **int tab[ ] = { 0, 1, 2};**

- avec l'opérateur **new**

initialisation explicite différée permet de déterminer la taille du tableau pendant l'exécution du programme

```
System.out.println("taille du tableau?");
int TAILLE =sc.nextInt();
int tabDyn [] = new int[TAILLE];
```

initialisation

- **Exemple:**

```
- String[] names = new String[2] ;
    names[0]= "nicolas";
    names[1]= "steve";

- Point [] tabPts = new Point [3];
    tabPts [0] = new Point (0, 2) ;
    tabPts [1] = new Pixel (2, 8, Color.black) ;
    tabPts [2] = new Point3D (2, 8, 9)
```

lenght : taille du tableau

L' attribut lenght retourne la taille du tableau:

```
int[] tab = {1, 10, 12, 9};
System.out.println("Size: " + tab.length);
```

dernier élément indéxé: length – 1.

Lenght est final = non modifiable → tableau non redimensionnable une fois crée.

Accèder au tableau

- à un élément du tableau(get) :

```
int tab[] = {1, 10, 12, 9};
out.println("Fourth value: " + tab[3]);
```

- à un indice d'un tableau(set):

```
Tab[1] = 25; // Positionne la valeur pour le second item
```

Dans l'expression tab[i], i (démarrant à 0) ne doit pas dépasser taille -1 du tableau sinon message d'erreur à l'éxécution du programme **java.lang.ArrayIndexOutOfBoundsException**

Exemple: `System.out.println(tab[4]);`

Java gère les erreurs avec le mécanisme des exceptions.
Les exceptions sont détaillées dans le chapitre qui leur est consacré.

Lecture

```
String TabNames[] = {"Nicolas", "Steve", "John"};
```

avec une boucle for → 2 manières:

```
1) traditional loop → avec un index
for (int i = 0; i < TabNames.length; i++) {
     System.out.println(TabNames[i]);
}
2) for-each loop →  sans index
for (String name : TabNames) {
    System.out.println(name);
}
```

Type object

Le type Tableaux est un objet →

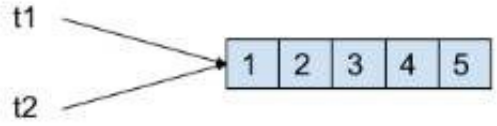
il hérite de la classe Object,

Cette classe fournit les méthodes :

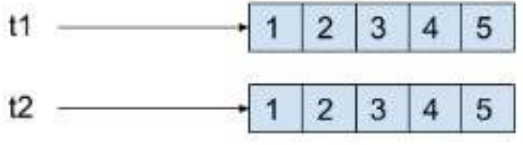
toString(), equals(), clone()…….

copie profonde : Clone

```
Copie superficielle
t1 = t2;
```




```
Copie profonde :
t2 = t1.clone();
t1[2]=33;
System.out.println(t1[2]);//affiche 33
System.out.println(t2[2]);//affiche 3
```

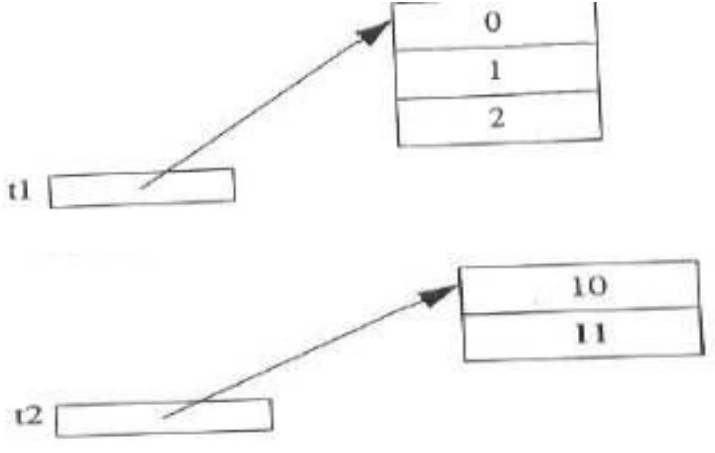


Semantique des référence

La variable de type tableau référence l'adresse en mémoire de cet objet.

Pour les variables t1, t2 c'est l'adresse en mémoire du premier élément du tableau

```
int t1 [] = new int[3];
t1 [0] = 0;
t1 [1] = 1;
t1 [2] = 2;

int t2 [] = new int[2];
t2 [0] = 10;
t2 [1] = 11;
```

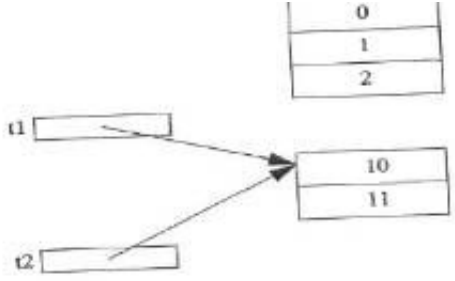


tableau

accès au tableau par référence

```
//acces au tableau par reférence
Avec l'instruction t1 = t2;
```

la référence contenue dans t2 est recopié dans t1 :




```
System.out.println(t1[0]);//affiche 10
t1[0]=3;
System.out.println(t2[0]);//affiche 3
```

immuabilité

Le contenu d'un tableau est toujours modifiable (**pas thread safe**)

```
...
int tab2Int [] = {10,20};
System.out.println ("avant appel:A=" + tab2Int[0]  + " B = "+ tab2Int[1]);
permut(tab2Int);
System.out.println ("apres appel:A = " + tab2Int[0]  + " B = "+ tab2Int[1]);
...
static void  permut (int tab[])
{
    int z ;
    z = tab[0] ;
    tab[0] = tab[1]  ;
    tab[1] = z ;
}
...
```

Tableau hétérogène

- variable de type tableau représente des éléments de même type
- possibilité de stocker dans un même tableaux des objets hétérogènes
- dans l'exemple, ils sont tous vues comme des points (polymorphisme, généricité...)

```
Point [] tabPts = new Point [3];
      tabPts [0] = new Point (0, 2) ;
      tabPts [1] = new Pixel (2, 8, Color.black) ;
      tabPts [2] = new Point3D (2, 8, 9)  ;
```

Collections

Tableau multidimensionnel

→ **n dim**ension

Présentation

les tableaux multi-dimensionnels sont des tableaux de tableau

→ les éléments du tableau sont eux même des tableaux

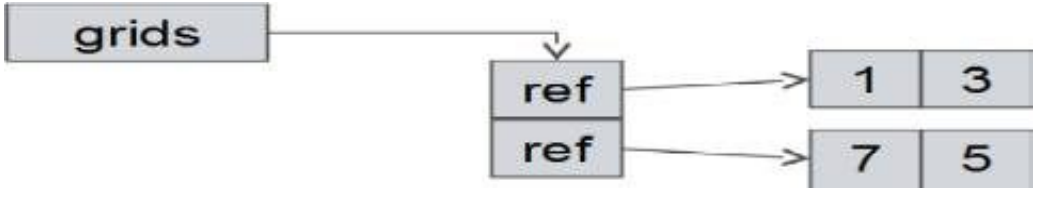
Syntaxe: ***type*** tab [ ][ ];

Tableau multidimensionnel

Le nombre de [ ] indique la dimension du tableau

Exemple tableau en 2 dimension:

```
-float[][] dimension = new float [10][10];
  // 10 × 10 tableau bidimensional de  float
-int[][] grids = { {1, 3}, {7, 5} };
```



Lire un élément d'un tableau 2D Exemple :

	Col 0	Col 1	Col 2	Col 3	Col dim2 -1
Ligne 0	8	4	6	-2	1
Ligne dim1-1	4	1	3	6	0

```
System.out.println( MonTab2D[1][1]);
```

Tableau à 2 dimensions

Lire tous les élements avec une boucle for

```
int MonTab2D[2][5]={ {8,4,6,-2,1 },{ 4,1,3,6,0 } };
int nblignes = 2;
int nbcol = 5;
 for (int i = 0; i < nblignes; i++)
     for (int j = 0; j < nbcol; j++)
     {
        printf("%d\n", MonTab2D[i][j]);
     }
```

Dimension hétérogène

dim1 n'est pas obligatoirement identique à ***dim2***

```
//String[][] Avengers = new String [2][3];
String[][] Avengers = {
                {"Mr. ", "Mrs. ", "Ms. "},
                {"Steed", "Peel"}
            };

System.out.println(Avengers[0][0]+Avengers[1][0]);
// Mr. Steed
System.out.println(Avengers[0][2] + Avengers[1][1]);
// Ms. Peel
```

Tableau de tableau

```
int tab[][] = { new int[4], new int[3], new int[5] } ;
```

la variable tab est une référence à un tableau qui contient

- ligne 1 une référence à un tableau de 4 entiers,
- ligne 2 une référence à un tableau de 3 entiers,
- ligne 3 une référence à un tableau de 5 entiers.

avec initialisation

automatique à 0 :

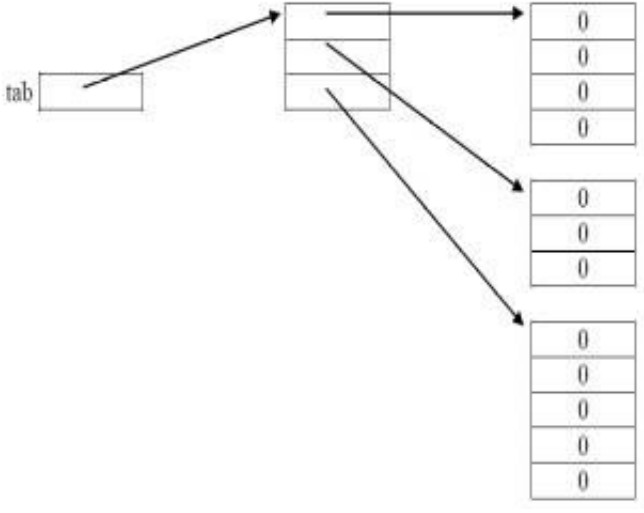
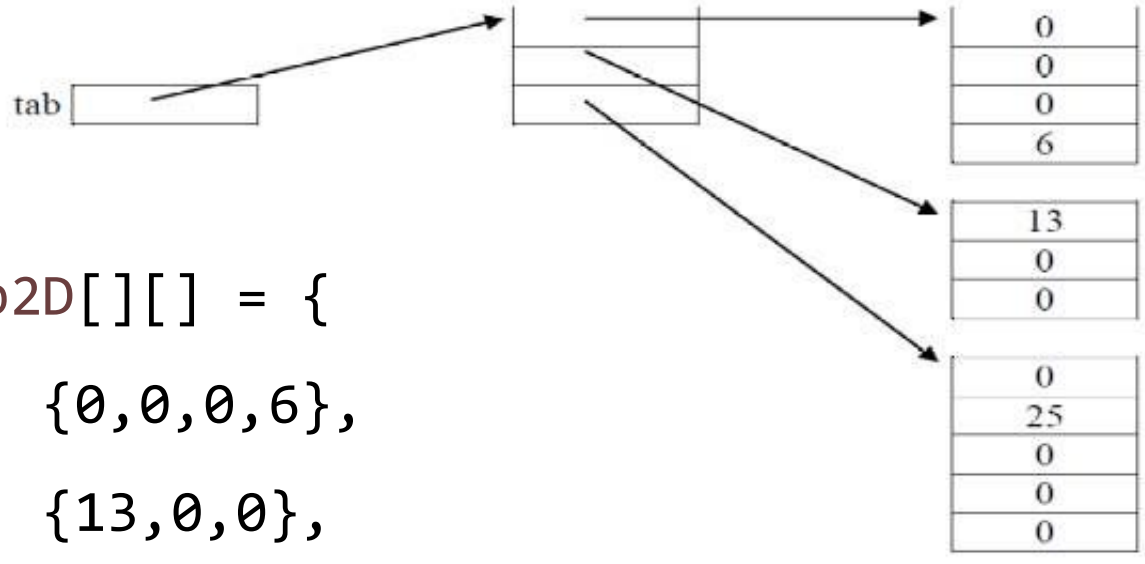


Tableau de tableau




```
int tab2D[][] = {
        {0,0,0,6},
        {13,0,0},
        {0,25,0,0,0 }
    };
```

Tableau & exception

La J.V.M surveille la validité des opérations et éventuellement déclenche des exceptions:

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 …

La gestion des exceptions sera traitée dans le support : **08_Exception**

En résumé

- 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 sont des entiers exclusivement.
- Les tableaux ne sont pas redimensionnables (taille déclarée une fois pour toutes).
- – exemple : le nombre de messages postés pour un sujet dans un forum n'étant pas limité, l'utilisation de tableau n'est pas une solution adaptée pour stocker ces messages

classe utilitaire pour les tableaux

- La classe **Arrays**

– fournit une gamme d'outils destinés à manipuler les tableaux
tri, recherche, copie, extraction, suppression, remplissage, transformation, ….

– sous forme de méthodes statiques exclusivement
exemple : sort, binarysearch, asList

– https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

Arrays common methods

La méthode sort:

static void sort(type[]) :

- pour un tri croissant des éléments dans un tableau

```
String TabFirstName[] = {"nicolas", "steve", "john","sophie"};
System.out.print("\n tableau de prenom: ");
for (String name : TabFirstName)
     System.out.print( name+ " ; ");
Arrays.sort(TabFirstName);
System.out.print("\n tri tableau prenom: ");
for (String name : TabFirstName)
    System.out.print( name+ " ; ");
```

```
tableau de prenom: nicolas ; steve ; john ; sophie ;
tri tableau prenom: john ; nicolas ; sophie ; steve ;
```

-A noter il faut que le type des éléments du tableau soient des ***Comparable***
Le type string implemente Comparable

tableau

Méthodes de classe Array

méthodes pour traiter les tableaux: recherche,conversion

- **static int binarySearch(Object[], Object) :**

- Recherche dans un tableau **trié** par rapport à une valeur donnée et retourne un index → de 0 à taille-1

```
String[] strArr = {"ABC","EFG","HIJ","KLM"};
System.out.println(Arrays.binarySearch(strArr,"ABC"));//0
System.out.println(Arrays.binarySearch(strArr,"KLM"));//3
```

- **static List asList(T[])** :

- Conversion d' un tableau en type collection

```
Integer SizeNonModif[]={1,2,3};
Collection<Integer> SizeModif = new ArrayList<Integer>(Arrays.asList(SizeNonModif));
SizeModif.add(4); //ajout d'éléments dans une collection
```

Collections

collection avec taille variable

→ LES COLLECTIONS

Caractéristiques des collections

- Classes destinées à la gestion d'objet
- Gestion automatique et dynamique
- Pas de limite de taille
- Permet de trier les éléments plus facilement
- Différents type de collections:
  - Adapté à vos besoins
  - Meilleures performances

    exemple type **list** ou **arraylist** (utilisé dans les diapos suivantes)

taille dynamique

- Add, remove, get, … elements

```
ArrayList<String> myString = new ArrayList<String>();
myString.add(" Hello ");
myString.add(" you ");
//a collection is never too small
myString.add("and you ");
```

```
ArrayList<Integer> myPhoneNumber = new ArrayList<Integer>();
myPhoneNumber.add(04);
myPhoneNumber.add(42);
myPhoneNumber.add(34);
myPhoneNumber.add(42);
```

classe utilitaire pour les collections

La classe **Collections**
(comme la classe **Arrays** mais pour des collections)
fournit une gamme d'outils destinés à manipuler les tableaux
tri, recherche, copie, extraction, suppression, ajout, transformation, ….
– Composées exclusivement de méthodes statiques

Classe utilitaire

Collections common methods

- static int binarySearch(List<T>, T) :

Recherche une valeur dans une List trié et retourne un index

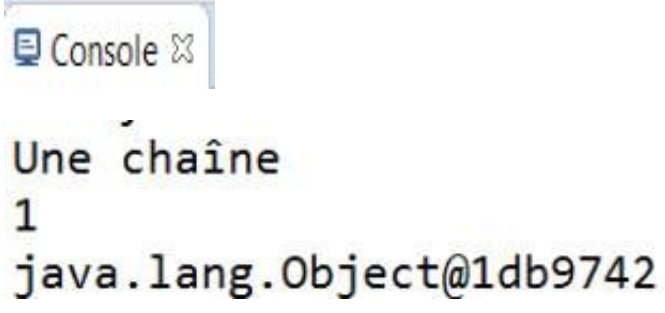
- static void reverse(List<T>) :
  - Renverse l'ordre des éléments dans une List
- static Comparator reverseOrder(Comparator) :
  - Retourne un Comparator qui ….→ (voire chapitre tri)
- static void sort(List<T>, Comparator) :
  - Tri une liste selon un comparator → (voire chapitre tri)

For more, look at the Javadoc :

http://download.oracle.com/javase/6/docs/api/java/util/Collections.html

Hétérogéneité→ généricité

```
ArrayList<Object> listHetero= new ArrayList<>();
listHetero.add("Une chaîne");
//boxing add(1)
listHetero.add(1);
listHetero.add(new Object());

for(Object obj : listHetero)
    System.out.println(obj);
```



Généricité

Collections

Présentation

- Apparue avec Java 5
- Semblable aux templates de C++.
- Plus besoin de cast
- Adapté aux collections
- Identique aux paramètres utilisés dans les méthodes génériques...

dans les collections

- parfaitement adaptée à l'usage de la généricité
- spécifie le type de donnée stocké dans les collections
- typable avec des paramètres qui ne sont pas des variables mais des « types classe »pour empêche de stocker des types hétérogènes
- La classe des éléments caractérisant les collections est spécifiée entre les symboles < et > qui suivent la classe de collection.

les classes génériques

La généricité 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** :

```
public static void main(String[] args)
{
 List list = new ArrayList();
 list.add(1);
 Integer integer =(Integer) list.get(0);

 List<Integer> listGenerik= new ArrayList<>();
 listGenerik.add(1);
 integer = listGenerik.get(0); // Plus besoin du cast

}
```

Méthodes génériques

Java permet de fournir des arguments **Type** dans les paramètres formels de méthode:

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

Ainsi que pour leur retour :

```
public static <T> T getFirst( T[] tableau)
{
     return tableau[0];
}
```

Méthodes génériques

```
character tabCharacter[] = { 'J', 'a','v','a' } ;
afficher(tabCharacter);
Integer tabInteger[] = { 0, 1, 2, 3  };
afficher(tabInteger);
```

Sans méthode générique, il faut écrire autant de méthodes qu'il y a de types:

```
public static  void afficher ( Integer tableau[])
{ //..}

public static  void afficher ( Character tableau[])
{ //..}
```

De type générique (en mentionnant dans l'entête de la méthode l'emploi d'un type de données<T>) une seule méthode suffit:

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

Spécifier le type de donnée à insérer

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

```
//insertion d'entier
ArrayList<Integer> monArrayList = new ArrayList<Integer>();
monArrayList.add(1);
monArrayList.add("Insertion impossible d'une string");

//insertion d'un objet de type point ou instance d'une classe dérivéee de point
ArrayList<Point> listePts =  new ArrayList<Point>();
Point Pt1 = new Point (3, 2);                listePts.add(Pt1);
Pixel Pt2 = new Pixel (2, 2, Color.black);   listePts.add(Pt2);
Pt3D  Pt3 = new Point3D (1, 5,3);            listePts.add(Pt3);

ArrayList<Pixel> listePixels =  new ArrayList<Pixel>();
Point Pt1 = new Point (3, 2);                listePixels.add(Pt1); → erreur compilation
Pixel Pt2 = new Pixel (2, 2, Color.black);   listePixels.add(Pt2);
Pt3D  Pt3 = new Point3D (1, 5,3);            listePixels.add(Pt3); → erreur compilation
```

Type générique contrôlé

Exemple: le tableau fournit à la méthode **getLastElmt** est de type **integer**

```
public static <T extends Integer> T getLastElmt( T[] tableau){
        return tableau[tableau.length - 1];
}
```

L'expression **<T extends Integer>** signifie que le type **T** doit être de type **Integer** ou type dérivé:

Toute tentative d'utiliser (appeler) *getLastElmt* autrement qu'en transmettant un ***Integer*** (ou dérivée) sera rejetée par Java :

```
Character tabCharacter[] = { 'J', 'a','v','a' } ;
Integer tabInteger[] = { 0, 1, 2, 3  };
getLastElmt(tabInteger)); //OK
getLastElmt(tabCharacter);//Erreur a la compilation
```

Collections

PARCOURS

Foreach

- Disponible depuis Java 5

```
String [] tab = {"one", "two", "three", "four" };
```

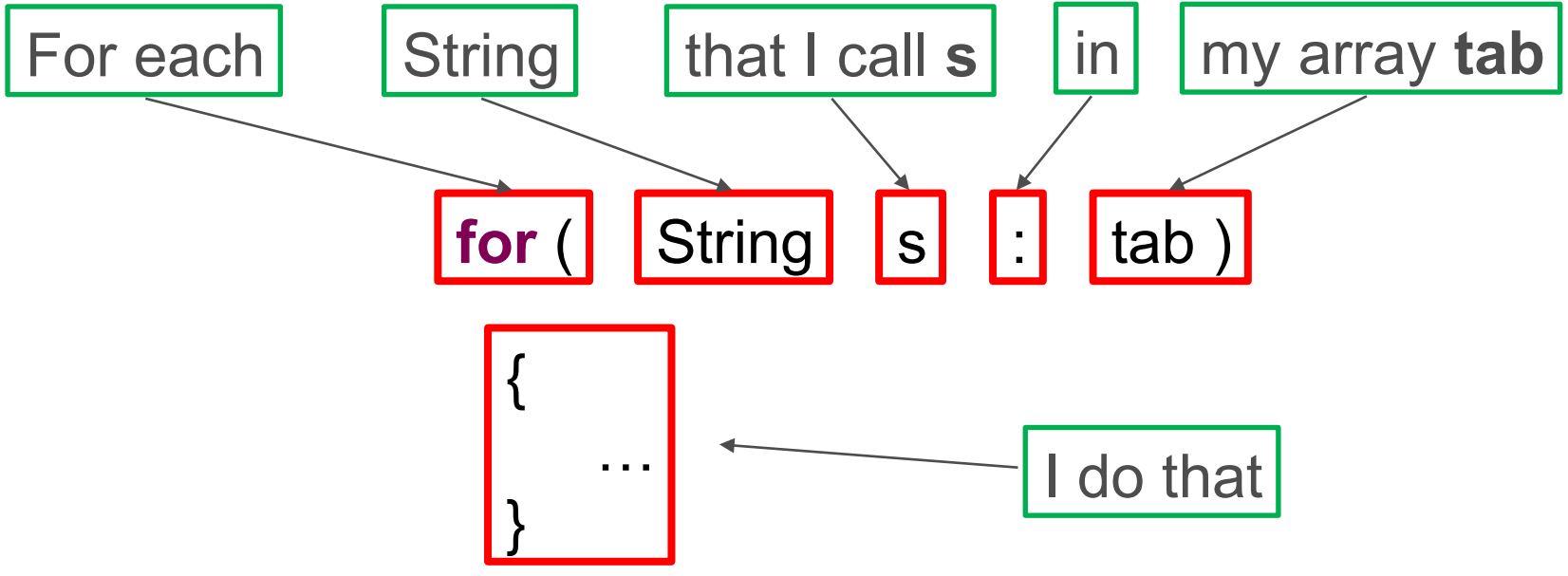
Avant :

```
for(int i = 0; i < 4; i+) {
    System.out.println(tab[i]);
    }
```

Après :

```
for(String s : tab) {
  System.out.println(s);
}
```

Foreach


For each
String
that I call s
in
my array tab
for (
String
s
:
tab)
{
…
}
I do that

Expression lambda JDK8

- **Exemple parcours d'une collection:**

```
List<Integer> myListInt = new ArrayList<Integer>();
myListInt.add(1);
myListInt.add(2);
...
myListInt.forEach(e -> System.out.println(e));
```

- **Exemple parcours d'un tableau avec filtre:**

```
String TabFirstName[] = {"nicolas", "steve", "john","sophie"};
System.out.println("initiale du nom  est s: ");
Arrays.stream( TabFirstName )
          .filter(i -> i.startsWith("s") )
          .forEach( i -> { System.out.println(i); } );
```

Avec un itérateur

Dans la hiérarchie des collections
l'interface *Collection* implémente
l'interface **iterable**,
qui met à disposition un itérateur.

Un Itérateur :

- objet générique , il itère sur les objets dans la collection
  - lit seulement dans un sens

Exemple : parcours collection avec un itérateur

```
Collection<String> myCollection = new ArrayList<String>();
// Add elements to the collection
myCollection.add("Elmt One");
myCollection.add("Elmt Two");

//positonne l'itérateur au debut de la collection
Iterator<String> it = myCollection.iterator();
//parcours la collection tant qu'il y a un elmt dans la
collection
while (it.hasNext()) {
  String myElement = it.next();
  System.out.println(myElement);
}
```

Interface ListIterator

Permet parcours de liste dans les 2 sens :

itérateur spécifique pour *List :*

```
ListIterator<String> listIt = myList.listIterator();
```

- Méthodes fournies par l'interface :

```
boolean hasNext(); //Check if there is a next element
boolean hasPrevious(); //Check if there is a previous element
E next(); //Get the next element
E previous(); //Get the previous element
void remove(); // Remove the actual element
```

voire code source CoursParcoursCollect2Sens

Collections

TRI : COMPARABLE/COMPARATOR

HTTPS://DOCS.ORACLE.COM/JAVASE/TUTORIAL/COLLECTIONS/INTERFACES/ORDER.HTML

Comparable <String>

String & Wrappers implémentent Comparable

La classe String contient la méthode **compareTo**

`compareTo(String anotherString)`

Compares two strings lexicographically.

```
ArrayList<String> course = new ArrayList<String>();
course.add("Lait");
course.add("Oeuf");
course.add("Sucre");
course.add(2, "Whisky");
System.out.println(course);
Collections.sort(course);
System.out.println(course);
```

```
[Lait, Oeuf, Whisky, Sucre]
[Lait, Oeuf, Sucre, Whisky]
```

https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

Comparable<xxx>

- la classe xxx implémentant l'interface Comparable<> pourra trier ses objets

-méthode à définir:

```
int compareTo(E e2);
```

- type de la valeur retournée est de type integer :
  - Negative if the current instance is inferior than the parameter
  - 0 if equal
  - Positive if the current instance is superior than the parameter

Comparable – Exemple la classe Point

```
//Classe implémentant l'interface Comparable  surcharge de la méthode compareTo(Obj) :
public class Point implements Comparable<Point> {
    int x,y;
    public Point(int x, int y) {  this.x = x;    }
    public int compareTo(Point pt){
        return  (this.x - pt.x);
    }
}
ArrayList<Point> listePts = new ArrayList<Point>();
listePts.add(new Point(13,2));
listePts.add(new Point(11,4));
listePts.add(new Point(34,45));
Collections.sort(listePts);
for(Point p : listePts) System.out.println(p.x);
```

Comparable – tri sur date embauche

```
class Salarie implements Comparable<Salarie> {
     protected String name;
     protected LocalDate hireDate;
     public User(String name , LocalDate hireDate ) {
        this.name = name;  this.hireDate = hireDate  }
     public int compareTo(User u2){
          return  name.compareTo(u2.name);
     }
}
List<Salarie> myListFirstName = new ArrayList<Salarie>();
          myListSalFirstName.add(new Salarie("John", dateHireJohn));
          myListSalFirstName.add(new Salarie("Michael",dateHireMich ));
          myListSalFirstName.add(new Salarie("Maria" ,dateHireMaria));
          Collections.sort(myListSalFirstName);
          for(Salarie s : myListSalFirstName) System.out.println(s.name);
//dans la classe Salarie pour une comparaison sur la  date embauche
//il est nécessaire de fournir  un Comparateur
Collections.sort(myListSalFirstName, Salarie.SENIORITY_ORDER);
```

```
Michael : 2020-01-02
Maria :   2020-01-12
John :    2020-02-15
```

Comparator

- Pour fournir un **comparateur**

La Classe implémente l'interface Comparator<E>

- Méthode à définir :

```
int compare(E e1, E e2);
```

La valeur retournée est un entier :

- Négative si e1 < e2
- 0 si égalité
- Positive si e1 > e2

Comparator – Exemple

- tri croissant sur la taille des strings

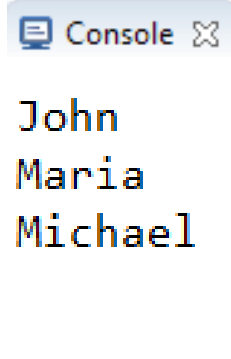
```
public class MyComparator implements Comparator<String> {
    public int compare(String s1, String s2){
        return (s1.length()- s2.length());
    }}
//utilisation de la classe MyComparator
List<String> myList= new ArrayList<String>();
myList.add("John");myList.add("Michael");
myList.add("Maria");
Collections.sort(myList, new MyComparator());

if (new MyComparator().compare("axx", "bx")== 1)
            System.out.println("a plus grand que b");
```

Console

```
John
Maria
Michael
```

Comparator – Exemple tri selon ancienneté

```
class Salarie implements Comparable<Salarie> {
     protected String name;
    protected LocalDate hireDate;
     public User(String name , LocalDate hireDate ) {
        this.name = name;  this.hireDate = hireDate }
}
 class CompareSal implements Comparator<Salarie>
{
     @Override
     public int compare(Salarie e1, Salarie e2) {
          return e1.hireDate.compareTo(e2.hireDate);
   }
}
List<Salarie> myListFirstName = new ArrayList<Salarie>();
          myListSalFirstName.add(new Salarie("John", dateHireJohn));
          myListSalFirstName.add(new Salarie("Michael",dateHireMich ));
          myListSalFirstName.add(new Salarie("Maria" ,dateHireMaria));
          Collections.sort(myListSalFirstName);
          for(Salarie s : myListSalFirstName) System.out.println(s.name);
//pour une comparaison sur la  date embauche nécessaire de fournir  un Comparateur
Collections.sort(myListSalFirstName, new CompareSal());
```



Comparable vs Comparator

- un Comparable représente un objet sur lequel est définie une relation d'ordre,
  - Je crée la classe de l'objet :

```
public class Salarie implements Comparable<Salarie>
```

contenant la méthode : `int compareTo(E e2);`

  - Fournit une séquence de tri unique sur un seul attribut
- un Comparator représente la relation d'ordre elle-même.
  - Je crée la classe spécialisée

```
public class CompareSal implements Comparator<Salarie>
```

contenant la méthode : `int compare(E e1, E e2);`

  - j'instancie un comparateur

```
Collections.sort(myListSal, new CompareSal());
```

  - Fournit plusieurs séquences de tri sur plusieurs attributs (plusieurs classes spécialisées)

Collections

INTERFACES & IMPLEMENTATIONS

Plusieurs type selon vos besoins

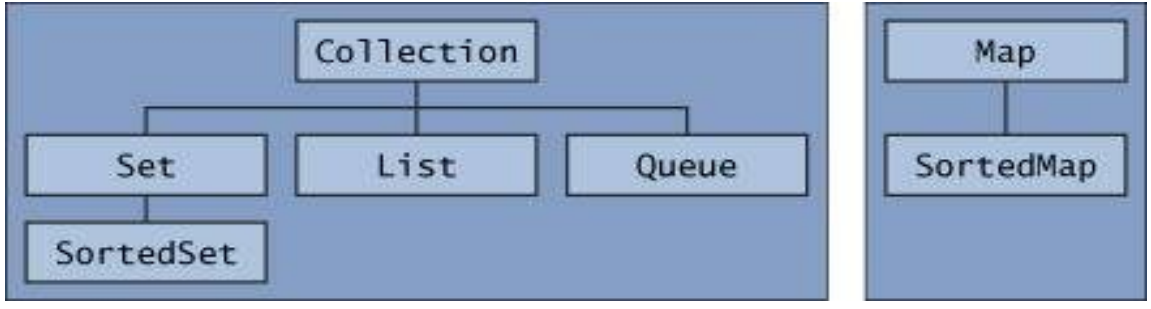
L’arborescence de l’A.P.I

Le package java.util propose un ensemble de collections, sous formes de classes utilitaires, divisé en 2 grandes familles chacune définie par une interface :

- java.util.Collection : pour gérer un groupe d'objets
- java.util.Map : pour gérer des éléments de type paires de clé/valeur

Chaque collection est une instance de classe implémentant l’interface Collection ou Map.

Arborescence



- présence d'interface (et les services associés) permet d'unifier l'utilisation des collections
- Tout interface utilise la généricité

Chaque interface a ses avantages et inconvénients

  - A vous de choisir la meilleure stratégie selon vos besoins

2 arborescences

- **Collection** : super classe implémentée par la plupart des objets qui gèrent des collections avec une généralisation maximum, fournit les services de base pour gérer une collection (add, remove)
Les sous interfaces sont fournies pour spécialiser les collections

- **Map** : définit des méthodes pour des objets gérant des collections
Principe : association entre une clé et des valeurs

```
Clé    : "David";
Valeur : int notesDavid[] = {2,4,1,17};
myMap.put("David" , notesDavid);
```

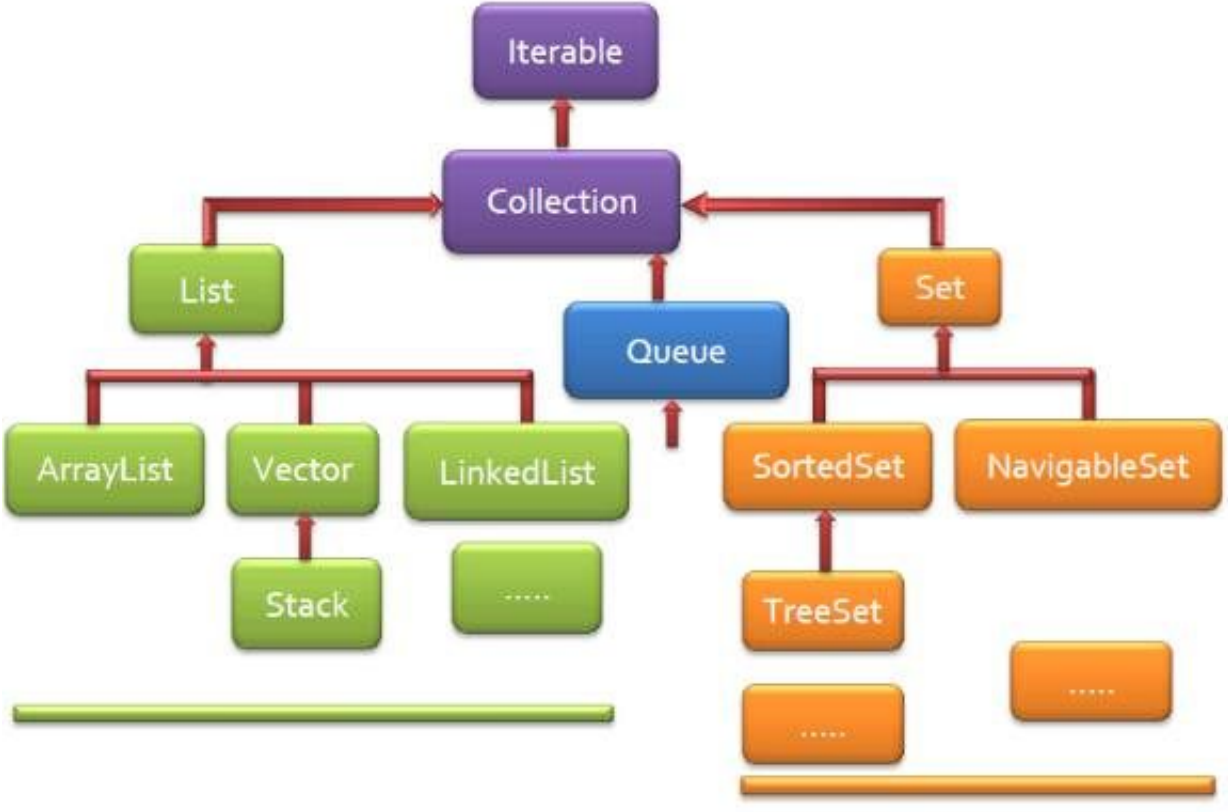
- Nous retrouvons les notes de David a partir de la clé "David"
  - `myMap.get("David");`

INTERFACES & IMPLEMENTATIONS

Interface Collection

Interface collection

arborescence java.util.Collection



sous interfaces

- Set :
  pour des objets qui n'autorisent pas de doublons
- List :
  pour des objets qui autorisent des doublons, un accès direct à un élément, relation d’ordre, insertion à l’endroit voulu.
- SortedSet :
  étend l'interface Set et permet d'ordonner l'ensemble

Définition interface List

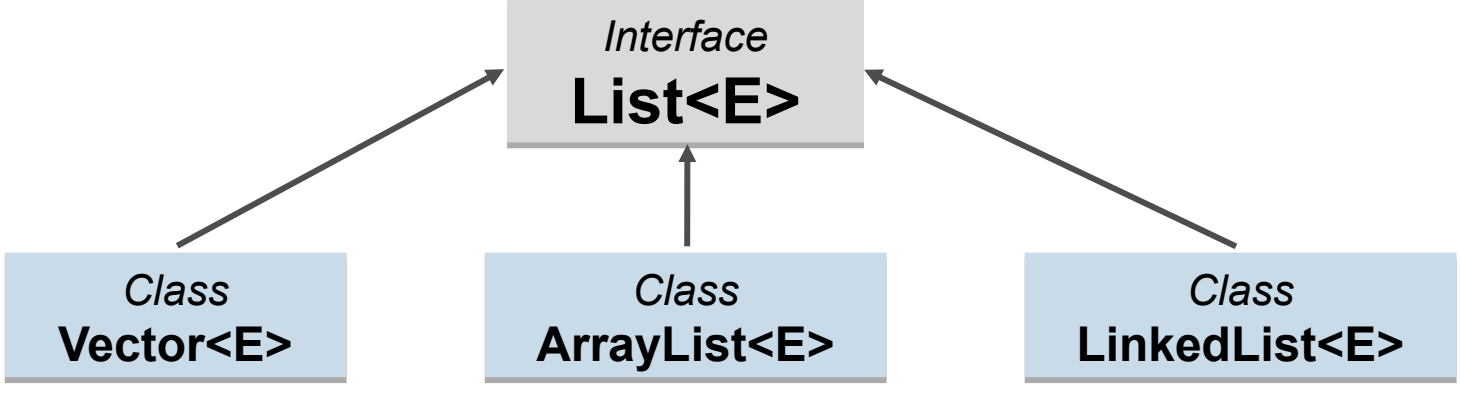
L'interface List, qui hérite de l'interface Collection, gère une liste d'objets indexée.

Une collection ordonnée (relation d'ordre→ séquence))

possibilité d'insérer des objets à l'endroit souhaité dans la liste.
A chaque objet déposé, la liste lui associe un index.
C'est grâce à cet index que l'on accède 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 (voire dans ce même support la partie parcours)

Interface List et descendance



Implémentations de l'interface list

- ArrayList : (modélise un tableau)
  - Bonne performance pour get et set
- LinkedList : (modélise une liste chaînée)
  - Meilleur performance pour add/remove
  - Pas performant pour get & set valeur
- Vector : (modélise un tableau)
  - Thread safe (toutes les méthodes sont synchronized)
  - Mauvaise performance

Les méthodes de *List* proposent des méthodes d'accès aux objets selon leur index et des méthodes de parcours de la collection.

List méthodes communes

- add(<any> element) :
  - Ajoute un élément à la fin de la liste
- <any> get(int index) :
  - Retourne l'élément à la position specifiée
- <any> remove(int index) :
  - Supprime l' élément à la position spécifiée
  - Retourne l'élément supprimé
- int size() : Retourne la taille de la Liste

Javadoc pour plus d'infos :

http://download.oracle.com/javase/6/docs/api/java/util/List.html

Exemple

```
List<String> myList = new ArrayList<String>();
myList.add("Monday");
myList.add("Tuesday");
myList.add("Wednesday");
myList.add("Thursday");
myList.add("Friday");
myList.add("Saturday");
myList.add("Sunday");

String day = myList.get(3);
System.out.println("The fourth day is " + day );
String removeDay = myList.remove(6);
System.out.println("I removed " + removeDay );
```

```
The fourth day is Thursday
I removed Sunday
```

Exemple

```
List<Integer> myListInt = new ArrayList<Integer>();

    myListInt.add(1);
    myListInt.add(2);
    myListInt.add(3);
    myListInt.add(4);
    myListInt.add(5);

    int UnEntierDansLaListe = myListInt.get(3);
    System.out.println("The fourth int is " +UnEntierDansLaListe );

    myListInt.set(3,34);
    System.out.println(myListInt.get(3) );
```

```
The fourth int is 4
34
```

Collections

INTERFACES & IMPLEMENTATIONS

La classe *Vector*

Présentation

-modélise la notion de tableau

-les éléments sont indexés avec des entiers, à partir de 0

-la classe *Vector* du package *java.util* permet de définir un vecteur :

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

-la collection se redimensionne pour accueillir les nouveaux éléments.

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

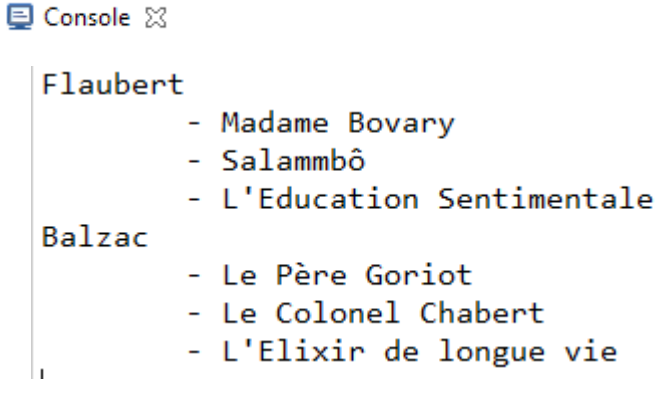
- *indexOf(Object)* // Renvoie l'indice d'un élément du vecteur, ou // -1 si la recherche a été infructueuse.
- *elementAt(int)* // Renvoie l'élément situé à l'index passé en paramètre.
- *size()* // Renvoie le nombre d'objets stockés.

Exemple (1/2)

```
class Auteur {
    private Vector<String> livres;
    private String nom;
    public Auteur ( String nom, Vector<String> livres ) {
        this.nom = nom;
        this.livres = livres;
 }
 public String toString() {
    // Itération sur le vecteur livres
    Enumeration<String> enumeration = livres.elements();
    while (enumeration.hasMoreElements()) {
        nom += "\n\t" + "- " + enumeration.nextElement();
    }
       return nom;
 }
 public void ajouterLivre( String unLivre ) {
    livres.addElement( unLivre );
 }
}
```

Exemple (2/2)

```
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>());
 // Les auteurs sont rangés dans un tableau d'auteurs
 Auteur tableauAuteurs[] = {auteurFlaubert, auteurBalzac};
 //Ajout des livres
 auteurFlaubert.ajouterLivre("Madame Bovary ");
 auteurFlaubert.ajouterLivre("Salammbô ");
 auteurFlaubert.ajouterLivre("L'Education Sentimentale ");
 auteurBalzac.ajouterLivre("Le Père Goriot ");
 auteurBalzac.ajouterLivre("Le Colonel Chabert");
 auteurBalzac.ajouterLivre("L'Elixir de longue vie");

 for (int i = 0; i < tableauAuteurs.length; i++) {
   System.out.println( tableauAuteurs[i] );
 }
}
```

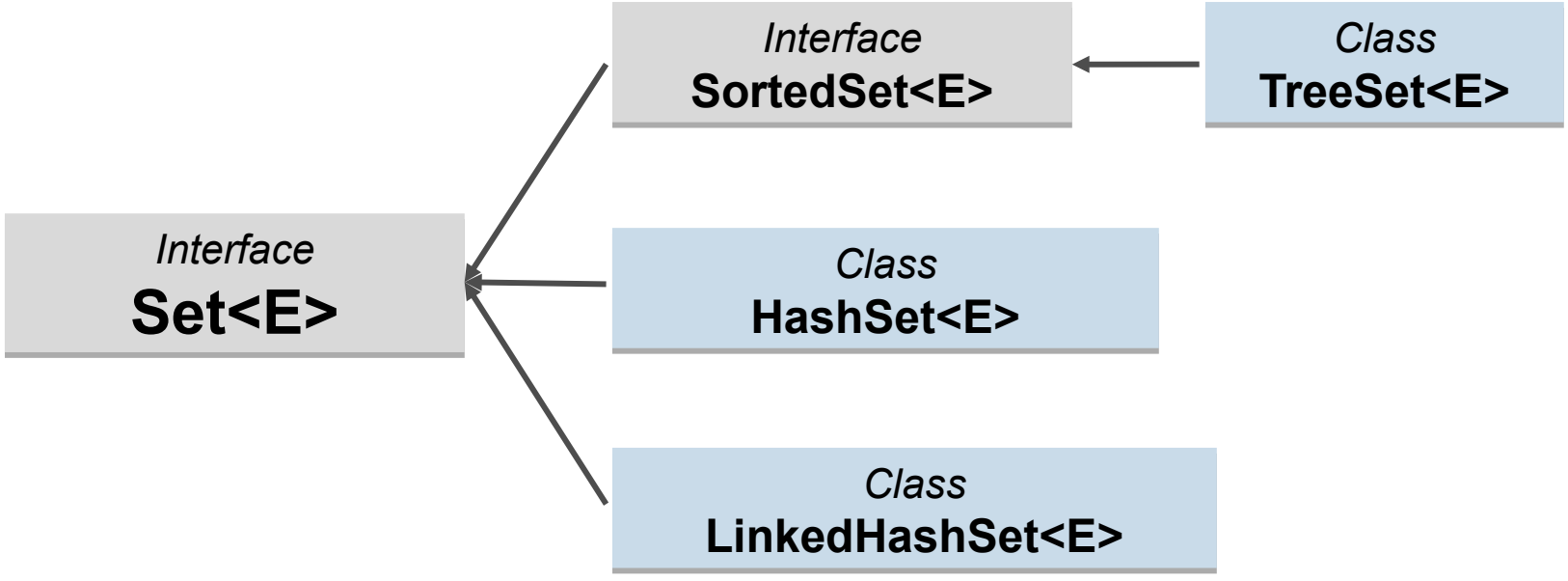


Collections

INTERFACES & IMPLEMENTATIONS

Interface Set

Arborescence



classes d'implémentation

Elles utilisent la notion de code de hachage. Les objets appartenant à une classe implémentant l'interface *Set* se stockent dans un hash table

garantissant la cohérence et les performances des classes

→ 2 instances égales retournant le même code de hachage ne peuvent pas coexister

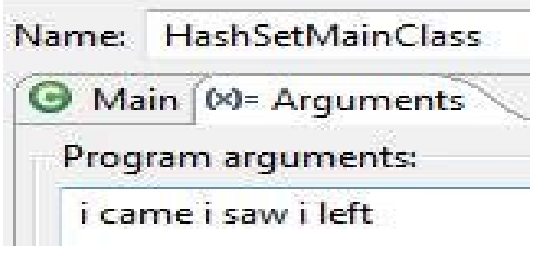
Conséquence : liste gérant un jeu d'objets unique (sans doublons)

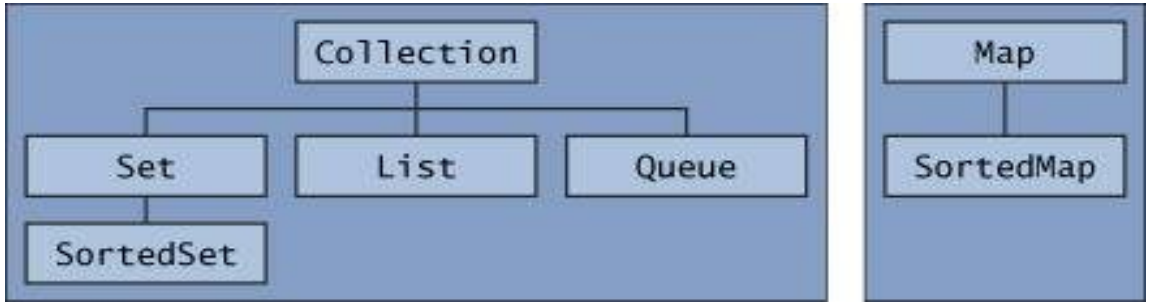
Exemple:hashset

HashSet :

- Manipule la valeur **hash** de la clé pour plus d'efficacité

```
Set<String> s = new HashSet<String>();
for (String a : args)
     s.add(a);
System.out.println(s.size() + " distinct words: " + s);
```





Interface Map

Présentation

Un jeu de clé/valeur (modélise table de hachage)

- Chaque clé est unique
- Une valeur peut avoir plus d’une clé
- Clés et Valeurs sont des Objets
- Similaire à un tableau associatif

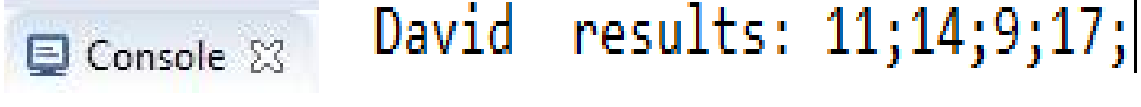
Définition d'un dictionnaire

Un dictionnaire est une collection d'éléments 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.

Exemple

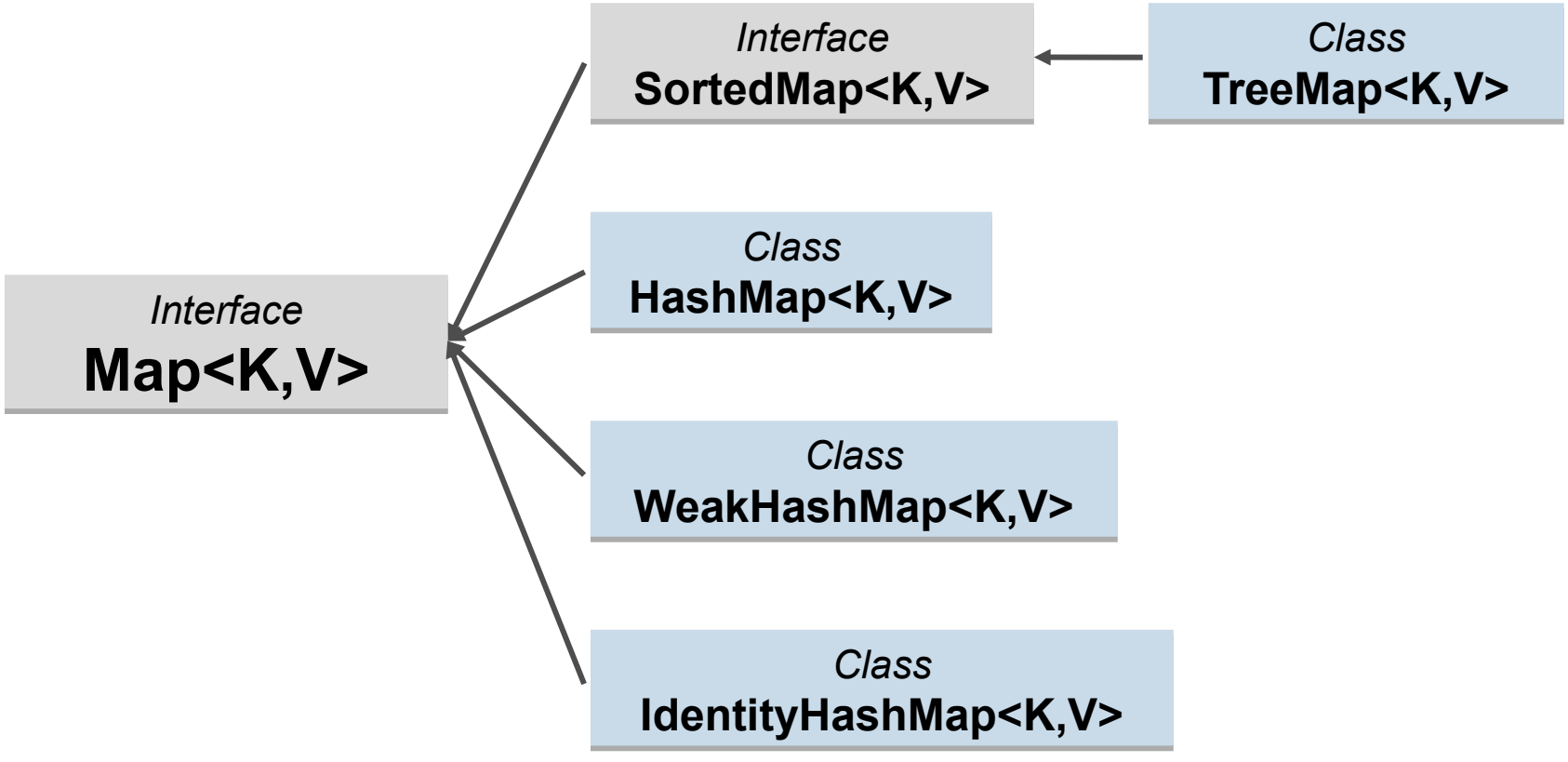
```
Map<String, int[]> myMap = new HashMap<String, int[]>();
int[] marksBob = {12,14,10,7};
myMap.put("Bob" , marksBob);
int[] marksDavid = {11,14,9,17};
myMap.put("David" , marksDavid);
int[] resultDavid =  myMap.get("David");
System.out.print("\nDavid  results: ");
for (int i:resultDavid)
  System.out.print(i + ";");
```

Le nom « David » joue le rôle de clé associé aux valeurs : 11, 14, 9, 17



Interface map

Arborescence



Implémentation

- HashMap :
  - Manipule la valeur hash de la clé pour plus d'efficacité
- SortedMap : tri les objets
- IdentityHashMap :
  - Comme HashMap

    Utilise l' operateur == pour vérifier si 2 keys sont égales
- TreeMap :

  Trie dans l'ordre croissant les tuple par clé

Méthodes communes

- put(<any> key, <any> value) :
  - Ajoute une paire key/value pair dans Map
- <any> get(<any> key) :
  - Retourne la valeur liée à la clé

- <any> remove(int index) :

  Supprime l'élément à la position spécifié

  Retourne l'élément supprimé

Méthodes communes

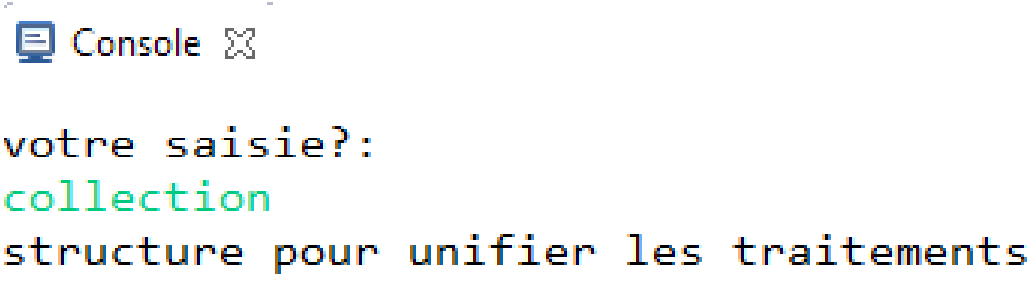
- boolean containsKey(<any> key) :
  - Vérifie si la key exist dans le Map
- boolean containsValue(<any> value) :
  - Vérifie si la valeur existe dans the Map

- For more, look at the Javadoc :

  http://download.oracle.com/javase/6/docs/api/java/util/List.html

Exemple

```
HashMap<String,String> glossaire = new HashMap<String,String >() ;
glossaire.put("collection","structure pour unifier les traitements");
glossaire.put("instance", "Objet créer à partir d'une classe");

String saisie = "collection";
if (glossaire.containsKey(saisie) )
        System.out.println(glossaire.get(saisie));
//affiche →  structure pour unifier les traitements
```



Parcours d'un HashMap

Avec une boucle for :

```
for (Map.Entry mapentry : map.entrySet())
{
    System.out.println("clé: "+mapentry.getKey()+ " | valeur: "+ mapentry.getValue());
}
```

Boucle while+iterator :

```
    Iterator iterator = map.entrySet().iterator();
     while (iterator.hasNext()) {
            Map.Entry mapentry = (Map.Entry) iterator.next();
           out.println("clé:"+mapentry.getKey()+"valeur:" + mapentry.getValue());
     }
```

Choix de la collection

A partir du sommet de la hiérarchie des collections, choisir un interface (Collection, List, Map). en se posant les questions

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 ?

Instancier la bonne collection et la désigner avec le type interface choisi.

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

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 pas de taille au départ.

Résumé Classifications des collections

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.

Résumé Classifications des collections

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.