Programmation générique

JAVA

Part 1

COLLECTIONS

Structures de données

- C'est l'organisation efficace d'un ensemble de données, sous la forme :
  - Tableaux
  - Listes
  - Piles
  - …

=> Cette efficacité réside dans la quantité mémoire utilisée pour stocker les données,
 le temps nécessaire pour réaliser des opérations sur ces données.

Définition Collection

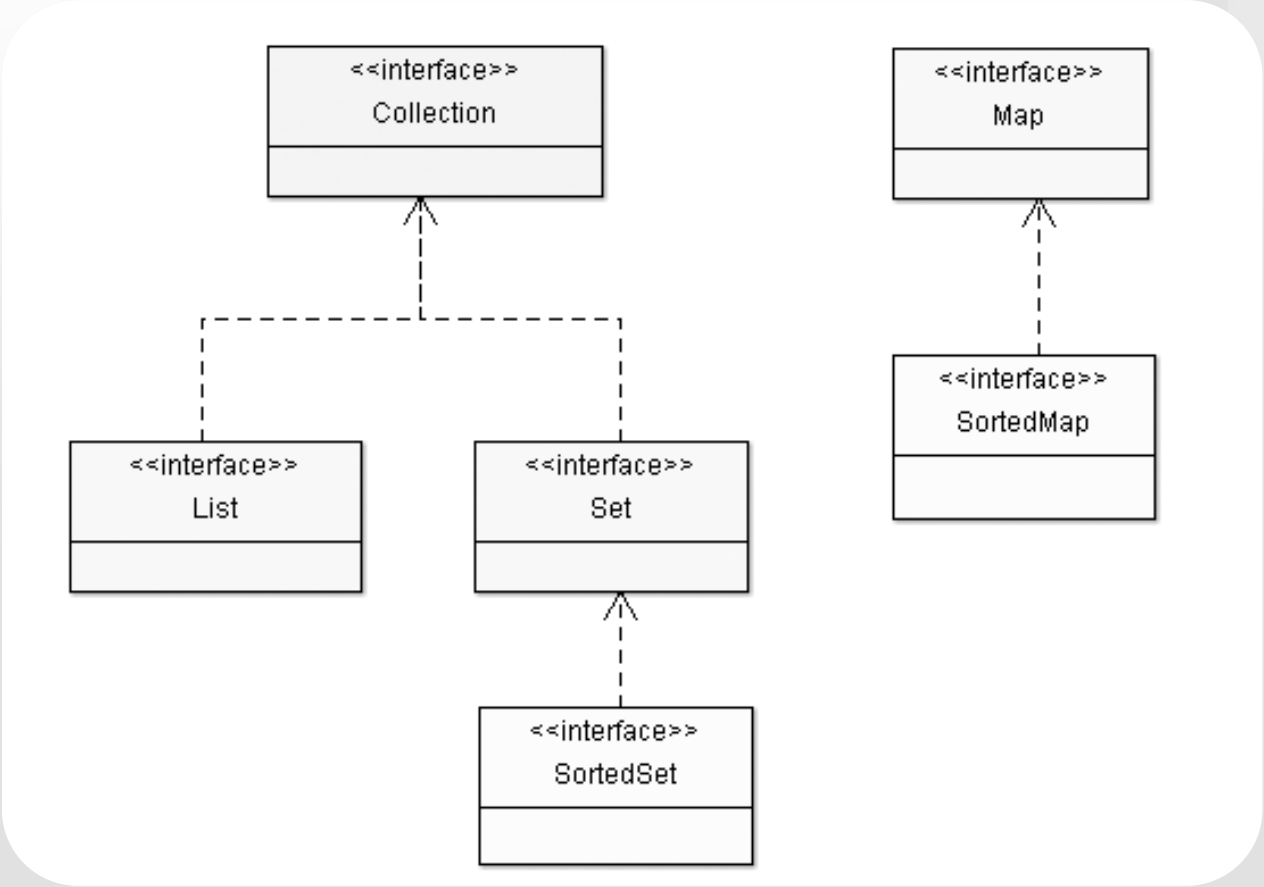
- Une collection est une structure de données permettant de regrouper un ensemble d'objets
- Comme un Tableau mais en mieux
- La plus part des opérations sur un tableau déjà implémentées
- On parle de Java Collections Framework

Introduction

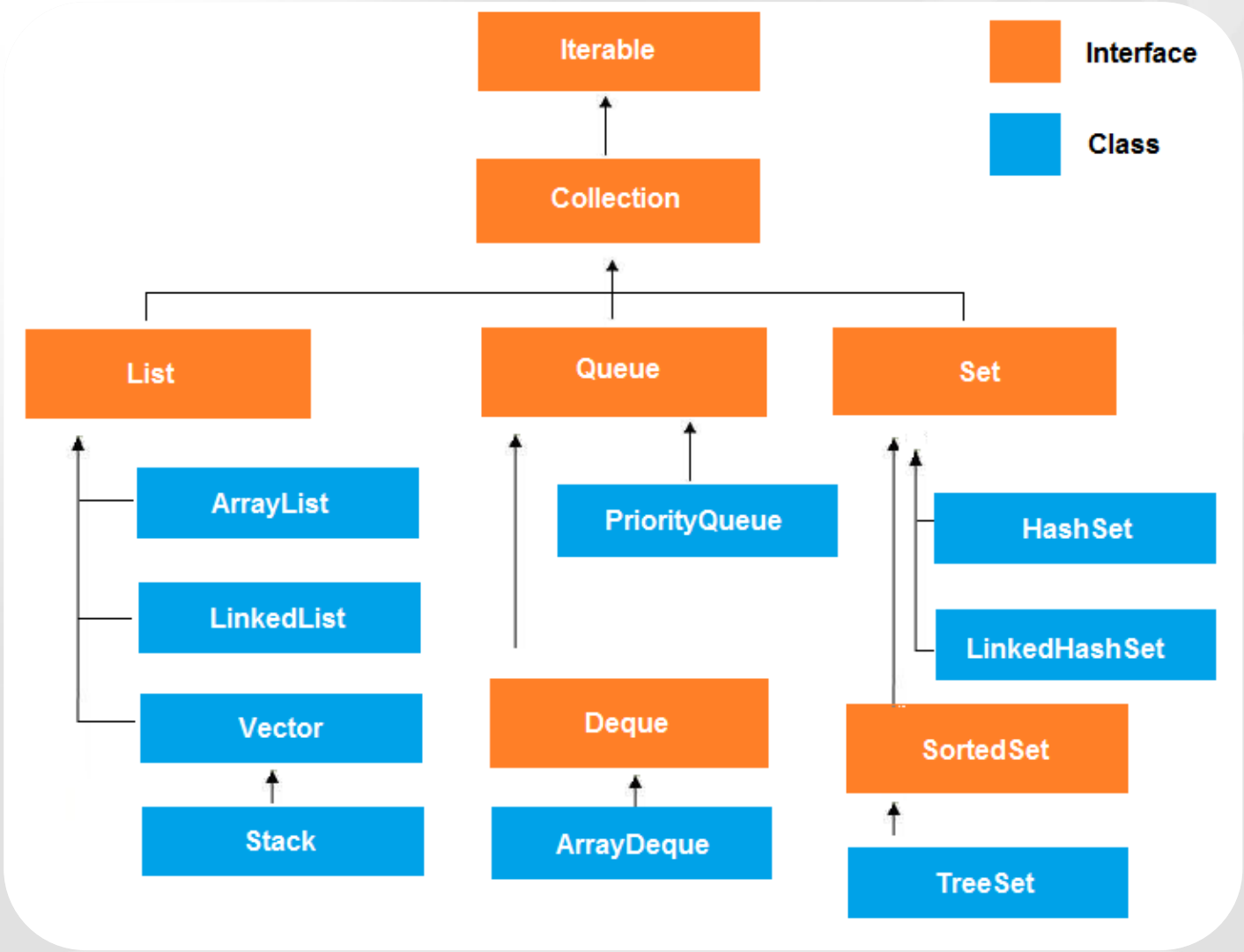
- Il n’est pas obligé de stocker les données dans des tableaux car les collections d'objets sont dynamiques
- Les collections n'ont pas de taille prédéfinie. Il est donc impossible de dépasser leur capacité !
- Il existe un grand nombre de types et objets Collection , mais nous verrons les principaux d'entre eux. Les objets que nous allons aborder sont tous dans le *package* **java.util**.
- Les collections sont primordiales dans les programmes Java.

Introduction

- La hiérarchie d'interfaces composant ce qu'on appelle les collections.
- il s'agit bien d'interfaces : celles-ci encapsulent la majeure partie des méthodes utilisables avec toutes les implémentations concrètes.



Introduction



Introduction

- il existe plusieurs types de collections, que les interfaces **List** et **Set** implémentent directement l'interface Collection
- l'interface **Map** gravite autour de cette hiérarchie, tout en faisant partie des collections Java.
- Ces interfaces ont des particularités correspondant à des besoins spécifiques.
  - Les objets de type List servent à stocker des objets sans condition particulière sur la façon de les stocker. Ils acceptent toutes les valeurs, même les valeurs null.
  - Les types Set sont un peu plus restrictifs, car ils n'autorisent pas deux fois la même valeur (le même objet), ce qui est pratique pour une liste d'éléments uniques, par exemple.
  - Les Map sont particulières, car elles fonctionnent avec un système clé - valeur pour ranger et retrouver les objets qu'elles contiennent.

Les objets List

- Les objets appartenant à la catégorie List sont, pour simplifier, des tableaux extensibles à volonté. On y trouve les objets Vector, LinkedList et ArrayList.
- Vous pouvez y insérer autant d'éléments que vous le souhaitez sans craindre de dépasser la taille de votre tableau.
- Ils fonctionnent tous de la même manière : vous pouvez récupérer les éléments de la liste via leurs indices..

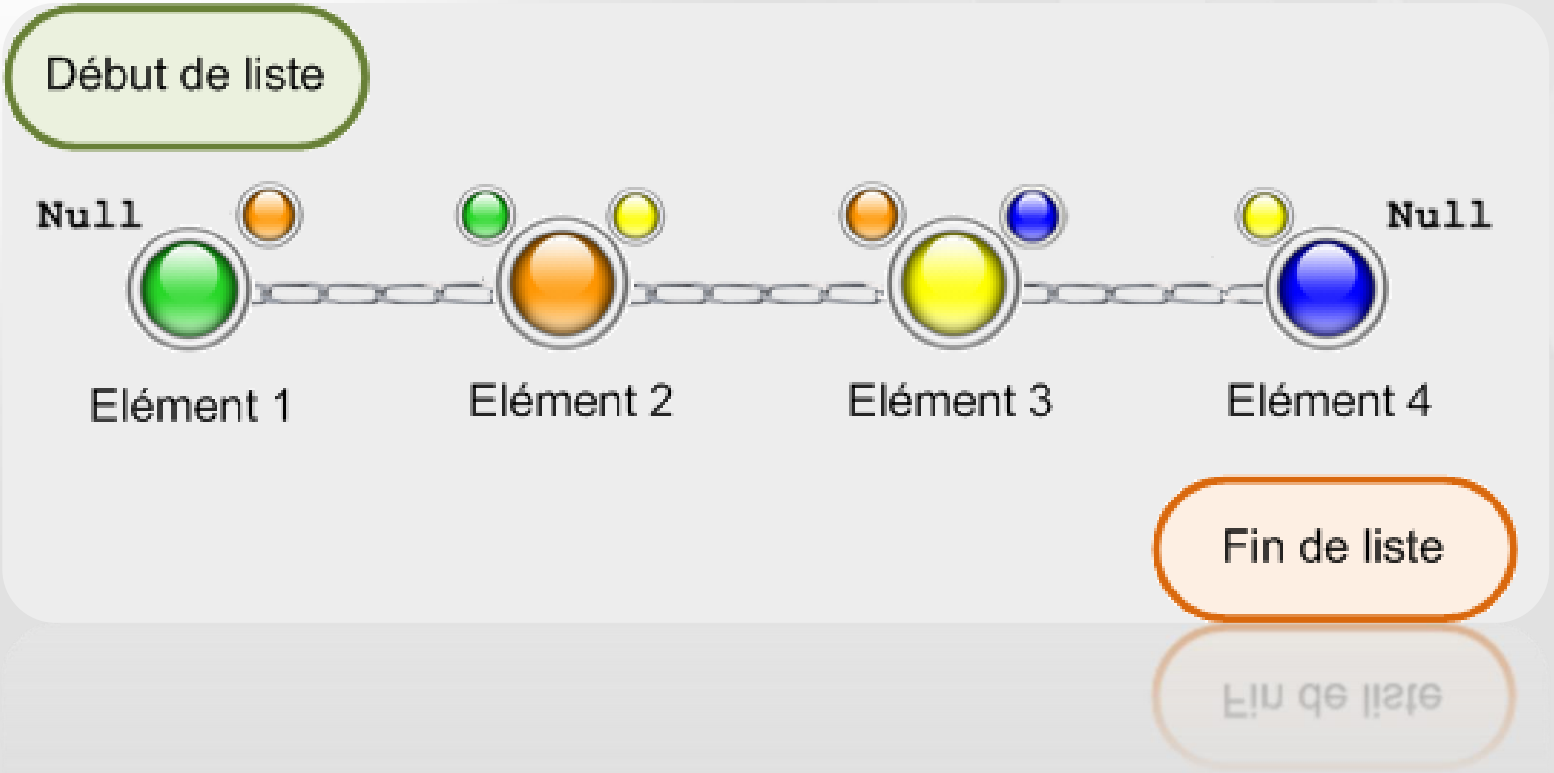
Les objets List

LinkedList

- Une liste chaînée (LinkedList en anglais) est une liste dont chaque élément est lié aux éléments adjacents par une référence à ces derniers. Chaque élément contient une référence à l'élément précédent et à l'élément suivant, exceptés le premier, dont l'élément précédent vaut null, et le dernier, dont l'élément suivant vaut également null.

Les objets List

LinkedList



Les objets List

LinkedList / Exemple

```
import java.util.LinkedList;
import java.util.List;

public class Test {

  public static void main(String[] args) {
    List l = new LinkedList();
    l.add(12);
    l.add("DSI2");
    l.add(12.20f);
    l.add(150.6);

    for(int i = 0; i < l.size(); i++)
      System.out.println("Élément à l'index " + i + " = " + l.get(i));
  }
}
```

Les objets List

LinkedList/ Iterator

- Cette classe implémente l'interface **Iterator**. Ainsi, nous pouvons utiliser cette interface pour lister notre LinkedList.
- Un itérateur est un objet qui a pour rôle de parcourir une collection.

Les objets List

LinkedList/ Iterator

- Cette interface est très simple, et n'expose que trois méthodes :
  - hasNext() : retourne true si la collection possède encore des éléments à itérer ;
  - next() : retourne l'élément suivant ;
  - remove() : permet de retirer de la collection l'élément courant. Cette opération est optionnelle, et peut ne pas être supportée par toutes les implémentations. Si une implémentation ne supporte pas cette méthode, alors elle doit jeter l'exception UnsupportedOperationException.

Les objets List

LinkedList / Iterator / Exemple

```
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
public class TestIterator {
  public static void main(String[] args) {
    List l = new LinkedList();
    l.add(12);
    l.add("Bonjour DSI2");
    l.add(12.20f);

    for(int i = 0; i < l.size(); i++)
      System.out.println("Élément à l'index " + i + " = " + l.get(i));

      System.out.println("\n \tParcours avec un itérateur ");
      System.out.println("-----------------------------------");
      ListIterator li = l.listIterator();

      while(li.hasNext())
        System.out.println(li.next());}}
```

Les objets List

LinkedList

- Vu que tous les éléments contiennent une référence à l'élément suivant, de telles listes risquent de devenir particulièrement lourdes en grandissant ! Cependant, elles sont adaptées lorsqu'il faut beaucoup manipuler une collection en supprimant ou en ajoutant des objets en milieu de liste. Elles sont donc à utiliser avec précaution.

Les objets List

ArrayList

- Un objet bien pratique. ArrayList est un de ces objets qui n'ont pas de taille limite
- Ils acceptent n'importe quel type de données, y compris null ! Nous pouvons mettre tout ce que nous voulons dans un ArrayList.

Les objets List

ArrayList/ Exemple

```
import java.util.*;
public class TestAray {
  public static void main(String[] args) {

    ArrayList al = new ArrayList();
    al.add(12);
    al.add("Une chaîne de caractères !");
    al.add(12.20f);
    al.add('d');

    for(int i = 0; i < al.size(); i++)
    {
      System.out.println("donnée à l'indice " + i + " = " +
   al.get(i));
    }
  }
}
```

Les objets List

ArrayList/Méthodes

- Sachez aussi qu'il existe tout un panel de méthodes fournies avec cet objet :
  - add() permet d'ajouter un élément ;
  - get(int index) retourne l'élément à l'indice demandé ;
  - remove(int index) efface l'entrée à l'indice demandé ;
  - isEmpty() renvoie « vrai » si la liste est vide ;
  - removeAll() efface tout le contenu de l'objet ;
  - contains(Object element) retourne « vrai » si l'élément passé en paramètre est dans l'ArrayList.

Les objets List

ArrayList / ArrayList

- Contrairement aux LinkedList, les ArrayList sont rapides en lecture, même avec un gros volume d'objets. Elles sont cependant plus lentes si vous devez ajouter ou supprimer des données en milieu de liste. Pour résumer, si vous effectuez beaucoup de lectures sans vous soucier de l'ordre des éléments, optez pour une ArrayList ; en revanche, si vous insérez beaucoup de données au milieu de la liste, optez pour une Linkedlist.

Les objets List

Vector

- La classe Vector du package java.util permet de stocker des objets dans un tableau dont la taille évolue avec les besoins.
- Parmi les constructeurs disponibles, l'un n'attend aucun paramètre, l'autre attend une taille initiale en paramètre. On construira donc un objet vect de la classe Vector en écrivant simplement:
  - Vector vect=new Vector(); ou
  - Vector vect=new Vector(n); pour prévoir une capacité initiale de n.

Les objets List

Vector

- Ajouter un objet: On ajoute un objet à la fin du Vector en utilisant la méthode addElement. Par exemple, pour ajouter l'objet o, on écrira :
  - vect.addElement(o); Le premier élément ajouté a l'indice 0, le suivant l'indice 1, etc... La méthode size() renvoie le nombre d'éléments contenus dans le vecteur qui sera aussi l'indice du prochain objet ajouté.
- Lire un objet: On retrouve un objet à partir de son indice en utilisant la méthode elementAt. Par exemple on obtient l'objet d'indice n en écrivant :
  - o=vect.elementAt(n);

Les objets List

Vector

- Autres méthodes: La classe Vector contient de nombreuses autres méthodes qui rendent son utilisation très aisée. Notons :
  - contains(Object) : indique si l'objet est contenu dans le Vector.
  - copyInto(Object[]) : copie les éléments dans un tableau classique.
  - firstElement() : renvoie le premier élément.
  - indexOf(Object) : renvoie l'indice de l'objet
  - insertElementAt(Object, int) : insère l'objet à l'indice indiqué
  - isEmpty() : indique si le Vector est vide
  - lastElement() : renvoie le dernier élément
  - removeAllElements() : vide le Vector
  - removeElementAt(int) : retire l'objet dont l'indice est donné
  - setElementAt(Object, int) : place l'objet à l'indice donné
  - size() : renvoie le nombre d'éléments

Les objets Set

- Un Set est une collection qui n'accepte pas les doublons. Par exemple, elle n'accepte qu'une seule fois null, car deux valeurs null sont considérées comme un doublon. On trouve parmi les Set les objets HashSet, TreeSet, LinkedHashSet… Certains Set sont plus restrictifs que d'autres : il en existe qui n'acceptent pas null, certains types d'objets, etc.
- Les Set sont particulièrement adaptés pour manipuler une grande quantité de données. Cependant, les performances de ceux-ci peuvent être amoindries en insertion. Généralement, on opte pour un HashSet, car il est plus performant en temps d'accès, mais si vous avez besoin que votre collection soit constamment triée, optez pour un TreeSet.

Les objets Set

HashSet

- HashSet est la plus utilisée des implémentations de l'interface Set. On peut parcourir ce type de collection avec un objet Iterator ou extraire de cet objet un tableau d'Object :

```
import java.util.HashSet;
import java.util.Iterator;

public class TestHashSet {
  public static void main(String[] args) {
    HashSet hs = new HashSet();
    hs.add("toto");
    hs.add(12);
    hs.add('d');

    Iterator it = hs.iterator();
    while(it.hasNext())
      System.out.println(it.next());

    System.out.println("\nParcours avec un tableau d'objet");
    System.out.println("--------------------------------");

    Object[] obj = hs.toArray();
    for(Object o : obj)
      System.out.println(o);
  }
}
```

Les objets Set

HashSet

- Voici une liste des méthodes que l'on trouve dans cet objet :
  - add() ajoute un élément ;
  - contains(Object value) retourne « vrai » si l'objet contient value ;
  - isEmpty() retourne « vrai » si l'objet est vide ;
  - iterator() renvoie un objet de type Iterator ;
  - remove(Object o) retire l'objet o de la collection ;
  - toArray() retourne un tableau d'Object.

Les objets Map

- Une collection de type Map est une collection qui fonctionne avec un couple clé - valeur. On y trouve les objets Hashtable, HashMap, TreeMap, WeakHashMap… La clé, qui sert à identifier une entrée dans notre collection, est unique. La valeur, au contraire, peut être associée à plusieurs clés.
- Ces objets ont comme point faible majeur leur rapport conflictuel avec la taille des données à stocker. En effet, plus vous aurez de valeurs à mettre dans un objet Map, plus celles-ci seront lentes et lourdes : logique, puisque par rapport aux autres collections, il stocke une donnée supplémentaire par enregistrement.

Les objets Map

HashTable / Exemple

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

public class TestHashTable {

  public static void main(String[] args) {

    Hashtable ht = new Hashtable();
    ht.put(1, "printemps");
    ht.put(10, "été");
    ht.put(12, "automne");
    ht.put(45, "hiver");

    Enumeration e = ht.elements();

    while(e.hasMoreElements())
      System.out.println(e.nextElement()); }
}
```

Les objets Map

HashTable

- Vous pouvez également dire « table de hachage », si vous traduisez mot à mot
- On parcourt cet objet grâce aux clés qu'il contient en recourant à la classe Enumeration.
- L'objet Enumeration contient notreHashtable et permet de le parcourir très simplement.

Les objets Map

HashTable

- Cet objet nous offre lui aussi tout un panel de méthodes utiles :
  - isEmpty() retourne « vrai » si l'objet est vide ;
  - contains(Object value) retourne « vrai » si la valeur est présente. Identique à containsValue(Object value) ;
  - containsKey(Object key) retourne « vrai » si la clé passée en paramètre est présente dans laHashtable ;
  - put(Object key, Object value) ajoute le couple key - value dans l'objet ;
  - elements() retourne une énumération des éléments de l'objet ;
  - keys() retourne la liste des clés sous forme d'énumération.

Collections

- Une collection permet de stocker un nombre variable d'objets.
- Il y a principalement trois types de collection : les List, les Set et les Map.
- Chaque type a ses avantages et ses inconvénients.
- Les Collection stockent des objets alors que les Map stockent un couple clé - valeur.
- Si vous insérez fréquemment des données en milieu de liste, utilisez une LinkedList.
- Si vous voulez rechercher ou accéder à une valeur via une clé de recherche, optez pour une collection de type Map.
- Si vous avez une grande quantité de données à traiter, tournez-vous vers une liste de type Set.

Part 2

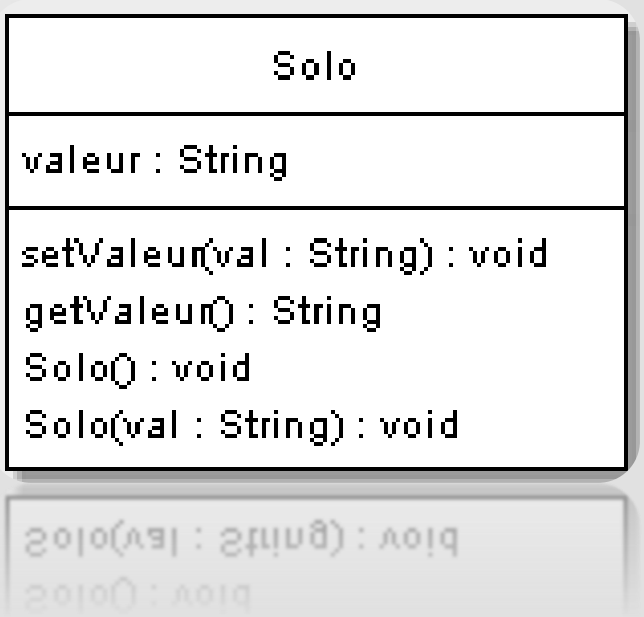
Classes et Méthodes génériques

Introduction

- Ce concept, ajouté au JDK depuis la version 1.5. Le principe de la généricité est de faire des classes qui n'acceptent qu'un certain type d'objets ou de données de façon dynamique !
- Les listes vu précédement acceptent tous les types de données. Par contre, un problème de taille se pose : lorsque vous voudrez travailler avec ces données, vous allez devoir faire un cast ! Et peut-être même un cast de cast, voire un cast de cast de cast…
- Solution depuis java 1.5 → générécité

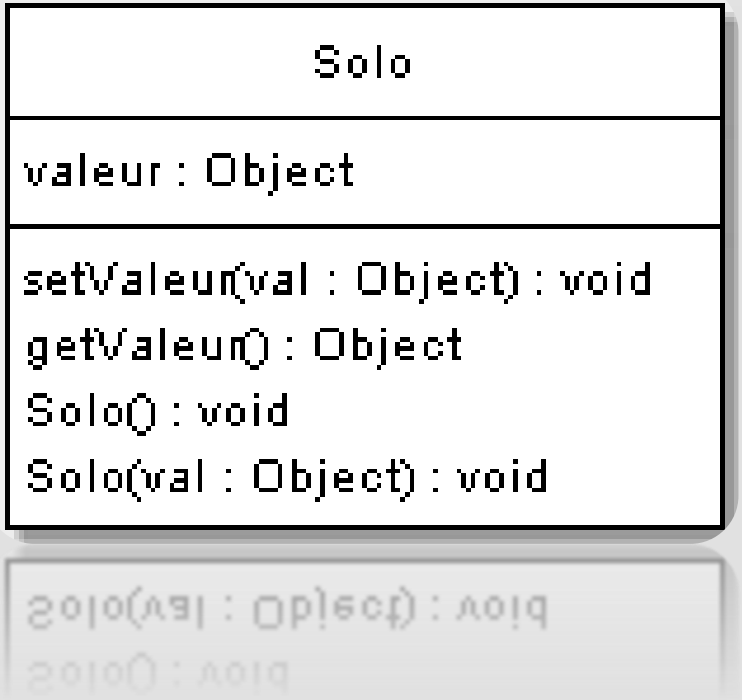
Principe de base

- Pour montrer la puissance de la généricité, nous allons voir un cas de classe qui ne l'utilise pas.
- Nous allons coder une classe Solo. Celle-ci va travailler avec des références de type String. Voici le diagramme de classe de cette dernière en figure suivante.



Principe de base

- On affecte une valeur, on peut la mettre à jour et la récupérer…
- Si on veut faire une classe qui permet de travailler avec n'importe quel type de données, on peut utiliser :



Principe de base

- Créez la classe Solo, ainsi qu'une classe avec une méthode main.

```
public class Solo {
        private Object valeur;
        public Solo(){
                this.valeur=null;}
        public Solo(Object valeur){
                this.valeur=valeur;}
        public Object getValeur(){
                return this.valeur;}
        public void setValeur(Object valeur){
                this.valeur=valeur;}}
```

```
public class TestSolo {
 public static void main(String[] args) {
  Solo val = new Solo(12);
  int nbre = val.getValeur();
  //erreuur
  System.out.println("nombre "+nbre);
 }
}
```

Principe de base

- Créez la classe Solo, ainsi qu'une classe avec une méthode main.

```
public class Solo {
        private Object valeur;
        public Solo(){
                this.valeur=null;}
        public Solo(Object valeur){
                this.valeur=valeur;}
        public Object getValeur(){
                return this.valeur;}
        public void setValeur(Object valeur){
                this.valeur=valeur;}}
```

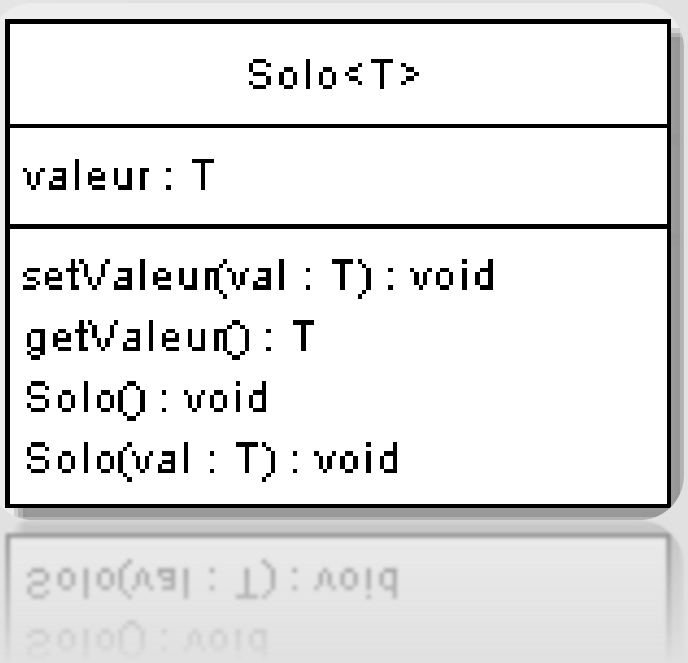
```
public class TestSolo
 public static void main(String[] args) {
  Solo val = new Solo(12);
  //int nbre = val.getValeur();

  int nbre = (Integer)val.getValeur();
  System.out.println("nombre "+nbre);
 }
}
```

On a essayé de mettre un objet de type **Object** dans un objet de type **Integer** : c'est interdit ! La classe Object est plus globale que la classe Integer, on ne peut pas effectuer cette opération, sauf si on caste l'objet en Integer comme ceci

Définition classe générique

- Pour le moment, on peut dire que la classe peut travailler avec tous les types de données en utilisant les cast, **mais** . On peut encore écrire une classe par type de donnée (SoloInt, SoloString, etc.).
- Et c'est là que la **généricité** s'avère utile, car avec cette dernière, on peut savoir ce que contient l'objet Solo et on n'aura qu'une seule classe à développer
- le diagramme de cette classe générique



Définition classe générique

- Une classe générique est une classe comprenant une ou plusieurs variables de type
- une classe paramétrée, dont le paramètre est un type d'objet

```
public class Solo<T> {
  //Variable d'instance
  private T valeur;

  //Constructeur par défaut
  public Solo(){
    this.valeur = null;}

  //Constructeur avec paramètre inconnu pour l'instant
  public Solo(T val){
    this.valeur = val;}

  //Définit la valeur avec le paramètre
  public void setValeur(T val){
    this.valeur = val; }

  //Retourne la valeur déjà « castée » par la signature de la méthode !
  public T getValeur(){
    return this.valeur;}      }
```

Définition classe générique

- Dans cette classe, le T n'est pas encore défini. Vous vous en occuperez à l'instanciation de la classe. **Par contre**, une fois instancié avec un type, l'objet ne pourra travailler qu'avec le type de données que vous lui avez spécifié ! Exemple de code :

```
public class TestSolo1 {
public static void main(String[] args) {
 Solo<Integer> val = new Solo<Integer>(12);
 int nbre = val.getValeur();
 System.out.println("nombre "+nbre);}}
```

- **Ce code fonctionne très bien, mais si vous essayez de faire ceci :**

```
public static void main(String[] args) {
 Solo<Integer> val = new Solo<Integer>("toto");
 //Ici, on essaie de mettre une chaîne de caractères à la place d'un entier
 int nbre = val.getValeur();
}
//ou encore ceci
public static void main(String[] args) {
 Solo<Integer> val = new Solo<Integer>(12);
 val.setValeur(12.2f);
 //Ici, on essaie de mettre un nombre à virgule flottante à la place d'un entier
}
```

Définition classe générique

- La classe Solo introduit une variable de type **T**, incluse entre **<>** après le nom de la classe
- Une classe générique (paramétrée) peut posséder plusieurs variables de type
  - utiliser l'opérateur virgule pour les séparer **public class Duo<T, S> {….}**
- Les variables de type peuvent être utilisées tout au long de la définition de la classe pour spécifier le type de retour des méthodes, le type des attributs, ou même le type de certaines variables locales
- rien n'empêche également d'utiliser des attributs ou d'autres éléments avec des types bien définis, c'est-à-dire non paramétrable, comme int, String, etc.
- Lorsqu'on déclare une référence de type Duo<String, Boolean>, on ne peux plus la changer en un autre type . Donc on ne peut pas modifier la déclaration générique d'un objet.

classes enveloppes (classe wrapper)

- On peut utiliser tous les types que nous souhaitons .

```
public static void main(String[] args) {
  Solo<Integer> val = new Solo<Integer>(12);
  Solo<String> valS = new Solo<String>("TOTOTOTO");
  Solo<Float> valF = new Solo<Float>(12.2f);
  Solo<Double> valD = new Solo<Double>(12.202568);
}
```

- **Remarque:** on'a pas utilisé ici les types de données qu'on emploie pour déclarer des variables de type primitif ! **Ce sont les classes de ces types primitifs.**
- En effet, lorsqu'on déclare une variable de type primitif, on peut utiliser ses classes **enveloppes (on parle aussi de classe wrapper)** ; elles ajoutent les méthodes de la classe Object à vos types primitifs ainsi que des méthodes permettant de caster leurs valeurs, etc…

autoboxing

- l'autoboxing, est une fonctionnalité du langage permettant de transformer automatiquement un type primitif en classe wrapper (ce qu'on appelle le boxing) et inversement, c'est-à-dire une classe wrapper en type primitif (ceci s'appelle l'unboxing). Ces deux fonctionnalités forment l'autoboxing. Par exemple :

```
public static void main(String[] args){
 int i = new Integer(12);        //Est équivalent à int i = 12
 double d = new Double(12.2586);  //Est équivalent à double d = 12.2586
 Double d = 12.0;
 Character c = 'C';
 ArrayLIst al = new ArrayList();
 //Avant Java 5 il fallait faire al.add(new Integer(12))
 //Depuis Java 5 il suffit de faire
 al.add(12);
 //...
}
```

Méthodes génériques

- On vient de voir comment définir une classe générique, Il est aussi possible de définir une seule méthode avec des paramètres de type :

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

- Cette méthode peut être définie dans une classe ordinaire
- C'est toutefois une méthode générique
- Les variables de type sont insérées après les modificateurs (public static, dans ce cas) et avant le type de retour

Méthodes génériques

- Lorsque on appelle une méthode générique, on peut placer les types réels, entourés des signes <>, avant le nom de la méthode :

  String[] noms= {"Marie", "possède", "une", "petite", "lampe"};

  String milieu = TableauAlg.<String>getMilieu(noms) ;

- Dans ce cas, on peut omettre le paramètre type <String> de l'appel de méthode. Le compilateur dispose de suffisamment d'informations pour un déduire la méthode que l'on souhaite utiliser. Il fait correspondre le type des noms (donc, String[]) avec le type générique T[] et en déduit que T doit être un String) On peut donc simplement appeler de la manière suivante :

  String milieu = TableauAlg.getMilieu(noms) ;

Part 3

Généricité et collections

Raw type

- Raw type = classe générique sans paramètre de type.
- Il se comporte comme une classe générique avec *Object* comme paramètre de type.

Ex: *ArrayList liste=new ArrayList(); // équivaut à ArrayList<Object>*

liste.add("toto");

liste.add(3); // Integer et String héritent de Object

On peut ajouter n'importe quel objet car tout objet hérite de *Object*.

Par contre, la méthode *get()* retourne un *Object* qu'il est nécessaire de **caster** correctement.

String s=(String)(liste.get(0)); // correct.

String s=(String)(liste.get(1)); // génère CastException

// car l'élément en position 1 est un Integer.

pas une bonne idée……

Introduction

- On peut utiliser la généricité sur les objets servant à gérer des collections. C'est même l'un des points les plus utiles de la généricité !
- En effet, lorsqu'on liste le contenu d'un ArrayList par exemple, on n'étiait jamais sûrs à 100 % du type de référence sur lequel on allait tomber (normal, puisqu'un ArrayList accepte tous les types d'objets)…
- D'où la nécessité d'utiliser des collections génériques

Exemple

```
import java.util.ArrayList; import java.util.List; import java.util.ListIterator;
public class TestColGen {
    public static void main(String[] args) {
        List<String> listeString= new ArrayList<String>();
        listeString.add("Une chaîne");
        listeString.add("Une autre");
        listeString.add("Encore une autre");
        listeString.add("Une dernière");
        ListIterator<String> li=listeString.listIterator();
        while(li.hasNext()){ System.out.println(li.next());  }
        System.out.println("----------------------------");
         List<Float> listeFloat = new ArrayList<Float>();
        listeFloat.add(12.25f);
        listeFloat.add(15.25f);
        listeFloat.add(2.25f);
        listeFloat.add(128764.25f);
         //for each
        for(float f : listeFloat)
          System.out.println(f);}}
```

ArrayList <E>

- Depuis Java 5, les collections sont implantées en utilisant des types «génériques». La généricité permet de paramétrer une classe ou une interface avec un ou plusieurs types de données paramètres formels qui seront remplacés par des types concrets lors de l'instanciation.
- Par exemple la classe ArrayList qui correspond à un tableau de taille variable se présente comme suit :

```
public class ArrayList<E> extends AbstractList<E> {
 public ArrayList() { … }
 public boolean add(E element) { … }          // ajouter à la fin
 public void add(int index, E element) { … }  // ajouter à indice index
 public E get(int index) { … }                // lire à indice index
 public int size() { … }                      // taille
 . . }
```

ArrayList <E>

- En fonction de la valeur du paramètre formel E la classe *ArrayList* pourra servir à stocker des ***String*** (il s'agira alors de la classe ***ArrayList<String>***), des ***Integer*** (il s'agira alors de la classe ***ArrayList<Integer>***)…

Ex: *ArrayList<Integer> listInteger = new ArrayList<Integer>();*

listInteger.add(new Integer(1)); // ou add(1)

Integer i = listInteger.get(0);

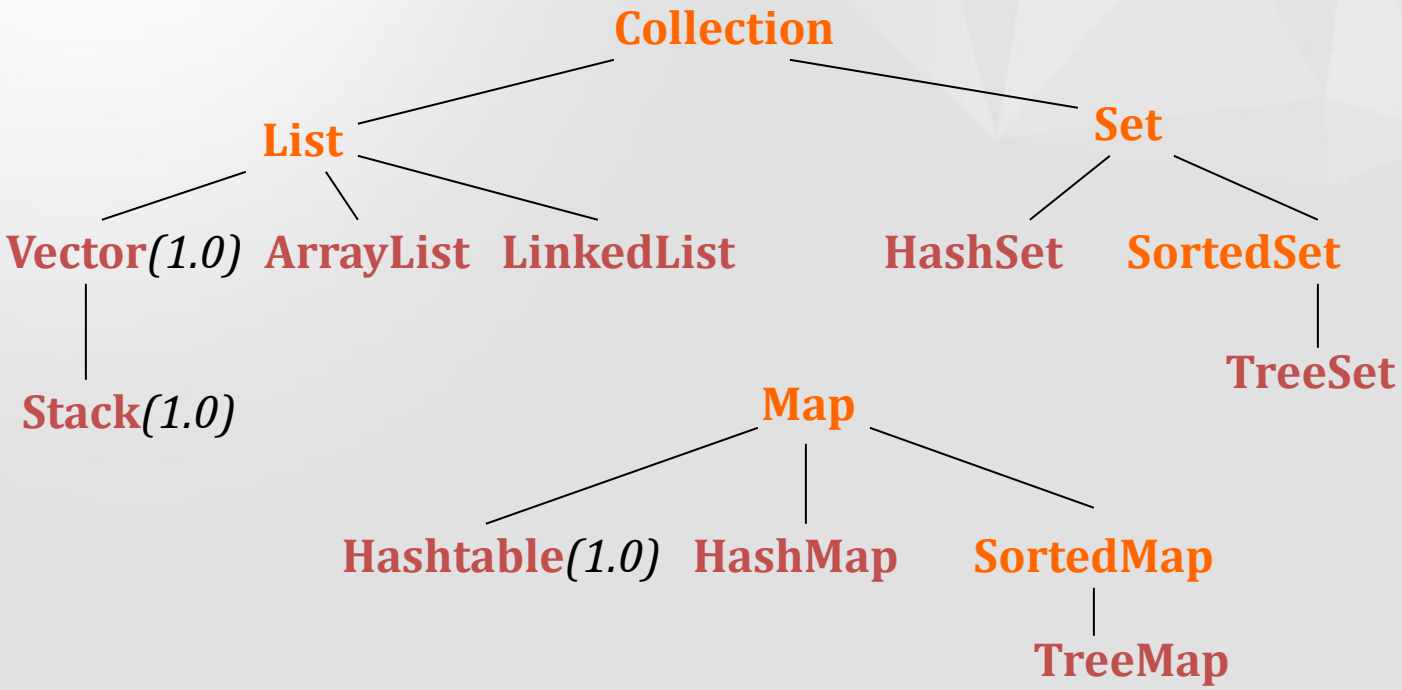
listInteger.add("chaîne"); // génère une erreur de compilation.

- Intuitivement *ArrayList<Integer>* est un *ArrayList<E>* avec toutes les occurrences de E remplacées par *Integer.*

Structuration de *java.util*

Rappel

- *java.util* regroupe les **classes** et **interfaces** suivantes :



Principales Méthodes de L'interface *Collection<E>*

- L'interface *Collection<E>* est la racine de la hiérarchie des interfaces *List* et *Set*. Elle contient les méthodes communes à toutes les collections d'objets (certaines peuvent ne pas être implantées).
- Principales méthodes de *Collection<E>* :
  - ***int size()*** : retourne le nombre d'éléments de la collection.
  - ***boolean isEmpty()*** : retourne true si la collection est vide.
  - ***boolean contains(Object elt)*** : retourne true si elt est dans la collection.
  - ***boolean add(E elt)*** : ajoute
  - ***boolean remove(Object elt)*** : supprime une occurrence de elt
  - ***Iterator<E> iterator()*** : retourne un 'itérateur' sur la collection.
  - ***void clear()*** : suppression de tous les éléments ().
  - ***Object[] toArray()*** : retourne un tableau contenant tous les éléments de la Collection.

Principales Méthodes de L'interface List<E>

- L'interface *List<E>* définit les opérations possibles sur les listes, en particulier :
  - ***E get(int i)*** : retourne l'élément qui se trouve à la position i.
  - ***E set(int i, E elt)*** : remplace l'élément en i par elt.
  - ***void add(int i, E elt)*** : ajoute elt en i et décale les autres éléments
  - ***E remove(int i)*** : supprime l'élément en i et décale les autres
  - ***int indexOf(Object o)*** : retourne la position de la première occurrence de o ou -1.
  - ***int lastIndexOf(Object o)*** : retourne la position de la dernière occurrence de o ou -1.
  - ***ListIterator<E> listIterator()*** : retourne un itérateur de liste.
  - ***List<E> subList(int from, int to)*** : sous-liste de from (inclus) à to (exclu).

Principales méthodes de la classe concrète ArrayList

- ArrayList<E> : implante l'interface *List<E>* et permet de créer des **tableaux d'objets de taille variable**. Les indices de ces tableaux commencent à 0. Elle implante aussi l'interface *Cloneable* ce qui fait qu'il est possible de cloner une *ArrayList* si besoin.
- Constructeurs:
  - ***ArrayList()*** : construit un tableau vide de capacité initiale 10; la capacité double à chaque fois que c'est nécessaire.
  - ***ArrayList(int c)*** : construit un tableau d'une capacité initiale donnée.

Principales méthodes de la classe concrète ArrayList

- Principales méthodes de ArrayList<E>:
  - ***int size()*** : nombre d'éléments du tableau
  - ***boolean isEmpty()*** : teste si le tableau est vide
  - ***boolean add(E elt)*** : ajoute *elt* à la **fin** du tableau
  - ***boolean add(int i, E elt)*** : **insère** *elt* à l'indice *i* et décale le reste
  - ***int indexOf(Object elt) :*** retourne l'indice de *elt* ou -1 (utilise la méthode *equals* pour trouver *elt*)
  - ***int lastIndexOf(Object elt)*** : retourne l'indice de la dernière occurrence de *elt* ou -1
  - ***E get(int i)*** : retourne l'élément d'indice *i*
  - ***void set(int i, E obj)*** : **modifie** l'élément à l'indice *i*
  - ***void clear()*** : supprime tous les éléments
  - ***Object clone()*** : duplique le tableau

Les tests d ’égalité

- Des méthodes de *ArrayList* ont besoin de tester si 2 objets sont identiques :
  - *boolean contains(Object* elt)
  - *int indexOf(Object elt) (et lastIndexOf)*
  - *boolean remove(Object elt)*
- *ArrayList<E>* utilise en interne la **méthode *equals*** pour savoir si 2 objets de type E sont égaux.

  Ex: *boolean contains(Object elt)* parcours la liste et teste pour chaque élément s’il est égal à l’élément elt.
- La méthode *equals* existe dans la classe *Object* et compare **les références des objets**.

  o1.equals(o2) donne vrai si et seulement si o1 et o2 correspondent à la même référence.
- Le comportement de la méthode *contains* **dépend donc de la manière dont la méthode *equals* est (re)définie dans la classe E.**

Les tests d ’égalité

Exemple

```
public class Point {
  int x,y;
  public Point(int x, int y) {
     this.x = x;
     this.y = y;
  }
}
public class PointEgal {
  int x,y;
  public PointEgal(int x,int y) {
    this.x=x;
    this.y=y;
  }
  public boolean equals(Object obj) {
    PointEgal other = (PointEgal) obj;
    return ((x==other.x)&&(y==other.y));
  }
}
```

La classe Point ne redéfinit pas la méthode *equals* (c’est donc celle de *Object* qui est utilisée).

La classe PointEgal redéfinit la méthode *equals* en spécifiant que deux objets sont égaux si et seulement si leur attributs sont égaux.

Les tests d ’égalité

Conséquence :

contains appliquée à une *ArrayList<Point>* retourne vrai si la **référence de l’objet en paramètre est dans la liste** :

```
ArrayList<Point> l = new ArrayList<Point>();
Point p = new Point(0, 0);
Point p2 = new Point(0, 0);
l.add(p);
System.out.println(l.contains(p2));      // affiche false
System.out.println(l.contains(p));       // affiche true
```

contains appliquée à une *arrayList<PointEgal>* retourne vrai si la liste **contient un objet ayant les mêmes attributs** que l’objet en paramètre :

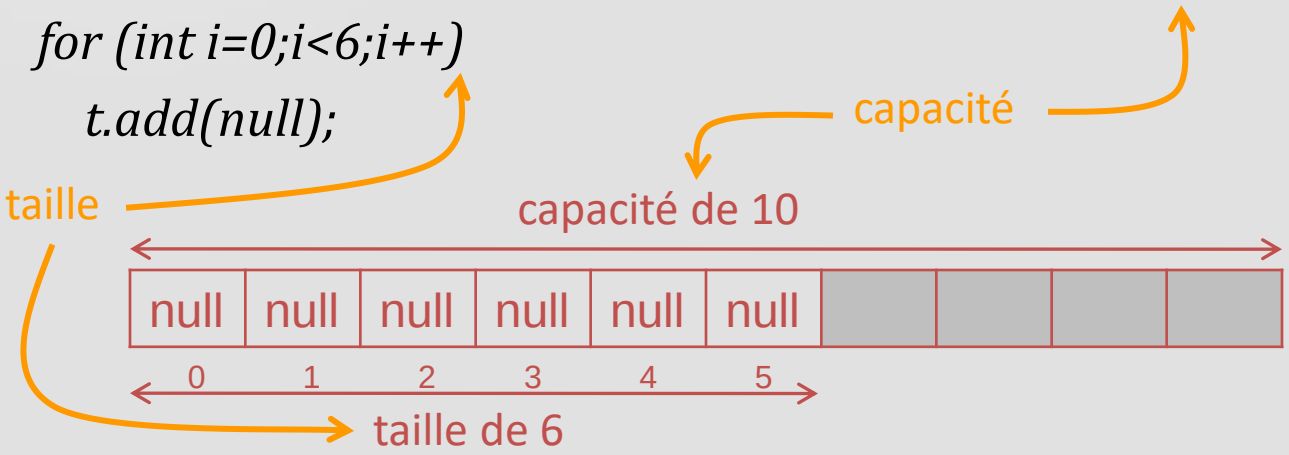
```
ArrayList<PointEgal> le=new ArrayList<PointEgal>();
PointEgal pe=new PointEgal(0,0);
PointEgal pe2=new PointEgal(0,0);
le.add(pe);
System.out.println(le.contains(pe2));  // affiche true
System.out.println(le.contains(pe));   // affiche true
```

Gestion de la mémoire

- **Taille = nombre effectif d'éléments contenus** dans l'*ArrayList*. Si l'*ArrayList* contient 6 éléments sa taille est de 6 et les éléments sont accessibles aux indices 0 à 5. Donnée par la méthode ***size().***
- **Capacité = nombre de cases contiguës réservées** pour stocker des éléments. Ces cases sont attribuées progressivement quand on ajoute des éléments.

Ex: ArrayList de capacité 10 ne contenant que 6 éléments

```
ArrayList<String> t=new ArrayList<String>(10);
for (int i=0;i<6;i++)
    t.add(null);
```



Gestion de la mémoire

- La taille augmente dès qu'on ajoute un objet.
- La capacité reste fixe tant que le nombre d'objets est inférieur à la capacité de la liste. Dès que ce nombre dépasse la capacité Java réalloue **automatiquement** de la mémoire.
- Dans le constructeur *ArrayList(int initialCapacity),* le paramètre *initialCapacity* ne désigne pas le nombre d'objets *null* initiaux mais le nombre maximum d'objets que l'*ArrayList* peut contenir.

La classe concrète *LinkedList<E>*

- *LinkedList<E>* : **liste doublement chaînée** (= listes symétriques vues en structures de données)
- Pour parcourir une liste doublement chaînée dans les deux sens, il faut utiliser un itérateur *ListIterator<E>* qui est une interface héritant de *Iterator<E>.*
- *LinkedList<E>* offre des méthodes pour ajouter, supprimer, obtenir le premier ou le dernier élément :
  - ***void addLast(E elt)*** : ajoute *elt* à la fin de la liste
  - ***void addFirst(E elt)*** : ajoute *elt* au début de la liste
  - ***E getLast()*** : retourne le dernier élément
  - ***E getFirst()*** : retourne le premier élément
  - ***E removeLast()*** : supprime et retourne le dernier élément
  - ***E removeFirst()*** : supprime et retourne le premier élément
- Comme *ArrayList, LinkedList* utilise *equals()* pour tester l'égalité entre 2 objets (ex: méthode *contains*).

Ex : parcours des listes chaînées
Interface Iterator<E>

```
LinkedList<Integer> tab = new LinkedList<Integer>();
for (int i=0; i<15; i++)
    tab.add(i);     // création d'une liste chaînée de 15 entiers
System.out.print("Premier element :");
System.out.println(tab.getFirst());
System.out.println("Parcours du debut a la fin :");
ListIterator<Integer> it = tab.listIterator();   // création de l'itérateur
while (it.hasNext())              // tant qu'il y a des éléments suivants
    System.out.println(it.next()*2);      // accéder à  l'élément suivant
System.out.println("Parcours a partir de la fin :");
while (it.hasPrevious())       // tant qu'il y a des éléments précédents
    System.out.println(it.previous()*2); // accéder à l'élément
    précédent
```

La classe concrète Vector<E>

- *Vector<E>* : **vecteur d'objets de taille variable.**

  Cette classe a été conservée bien qu'elle soit très similaire à *ArrayList<E>*. Elle a été intégrée au package et implante les interfaces *Collection<E>* et *List<E>*.

  Ex : *void setElementAt(E element, int index)* venant de *Vector<E>* a la même fonctionnalité que la méthode *E set(int index, E element)* venant de *ArrayList<E>.*

La classe Stack<E>

- *Stack<E> dérive de Vector<E> et permet de créer des piles.*
  - ***E push (E elt)** : empile et retourne l'élément elt dans la pile*
  - ***E pop ()** : dépile et retourne l'élément en haut de la pile*
  - ***E peek ()** : retourne l'élément en haut de la pile*
  - ***boolean empty ()** : retourne vrai ssi la pile est vide.*

Ex: `Stack<String> s=new Stack<String>();`

```
for (int i=0; i<10; i++)
    s.push(""+i);       // empile les 10 premiers entiers
for (int i=0; i<10; i++)
    System.out.println(s.pop());  // les affiche (de 9 à 0)
```

L'interface Set<E>

- L'interface *Set<E>* décrit les collections **ne pouvant contenir d'éléments dupliqués** (ensemble au sens mathématique du terme). La méthode *add()* réalise l'ajout uniquement si l'élément n'est pas déjà présent dans le *Set<E>*. L'unicité des éléments est basée sur la méthode ***equals*** définie dans la classe *Object* et que l'on peut bien entendu redéfinir pour un type particulier d'élément.
- Opérations ensemblistes définies par *Set<E>* :
  - ***s1.containsAll(Collection<?> s2)*** : true si s2 est un **sous-ensemble** de s1,
  - ***s1.addAll(Collection<? extends E> s2)*** : **union** de s1 et s2 dans s1,
  - ***s1.retainAll(Collection<?> s2)*** : **intersection** de s1 et s2 dans s1,
  - ***s1.removeAll(Collection<?> s2)*** : **différence** ensembliste de s1 et s2 dans s1.

L'interface *SortedSet<E>*

- L'interface *SortedSet<E>* dérive de l'interface *Set<E>* et décrit des ensembles **triés en permanence par valeurs croissantes**.
- En plus des méthodes héritées de *Set<E>*, *SortedSet<E>* possède des méthodes spécifiques liées au tri des éléments qu'elle contient comme :
  - ***E first()*** retourne le premier élément contenu dans l'ensemble trié (fonction de la manière dont *compareTo* est définie),
  - ***E last()*** retourne le dernier élément contenu dans l'ensemble trié.

La classe concrète *TreeSet<E>*

- TreeSet<E> = **ensemble d'éléments ordonnés,** implémente SortedSet<E>.
- Principales méthodes :
  - ***boolean add(Object elt)*** : ajoute elt si celui-ci n'est pas déjà présent.
  - ***boolean contains(Object elt)*** : retourne true ssi elt est présent.
  - ***int size()*** : retourne le nombre d'éléments.
  - ***boolean remove(Object elt)*** : enlève elt.
  - ***Iterator iterator()*** : retourne un itérateur pour parcourir l'ensemble.
- Les éléments sont triés grâce à la méthode *compareTo(Objet o)*. Il faut donc que les objets contenus dans l'ensemble implémentent l'interface *Comparable*.
- La méthode *compareTo(o)* renvoie 0 si le récepteur du message et le paramètre sont égaux, une valeur négative si le paramètre est plus petit et une valeur positive si le paramètre est plus grand. La méthode *equals* doit être cohérente avec la méthode *compareTo*.

Exemple

PointTrie représente des points triés (en priorité selon y puis selon x).

```
public class PointTrie implements Comparable<PointTrie> {
  int x,y;
  public PointTrie(int x, int y){
     this.x = x;
     this.y = y;
  }
  public int compareTo(PointTrie o){
     if (y<o.y) return(-1);
     if (y>o.y) return(1);
     if (x<o.x) return(-1);
     if (x>o.x) return(1);
     return 0;
  }
  public boolean equals(Object o){
     PointTrie p = (PointTrie) o;
     return ((x==p.x)&&(y==p.y));
  }
  public String toString() {return x+"  " +y;}
}
```

On peut utiliser la classeTreeSet<PointTrie>
car PointTrie implémente l'interface Comparable :

```
TreeSet<PointTrie> ts=new TreeSet<PointTrie>();
ts.add(new PointTrie(1,1));
ts.add(new PointTrie(5,2));
ts.add(new PointTrie(2,3));
ts.add(new PointTrie(2,1));
System.out.println(ts.last()); // affiche 2,3
```

Car ts est trié :

1,1
2,1
5,2
2,3

Le parcours des collections

- Il existe trois façons de parcourir des collections :
  - de la même manière que des tableaux classiques,
  - avec des itérateurs,
  - en utilisant des itérations simplifiées.

1. **Parcours à l'aide de la taille** grâce aux méthodes ***int size()*** et ***E get(int i)*** :

Ex : *ArrayList<String> li=new ArrayList<String>();*

```
li.add("un");
li.add("deux");
li.add("trois");
for (int i=0; i<li.size(); i++)
    System.out.println(li.get(i));
```

Le parcours des collections itérateurs

2. Parcours à l'aide des itérateurs

- En trois étapes:
  - instancier un itérateur sur la collection,
  - faire une boucle tant que l'itérateur possède un élément suivant,
  - récupérer l'élément suivant.
- **Interface *Iterator<E>*** :
  - ***boolean hasNext()*** : indique s'il reste des éléments à parcourir;
  - ***E next()*** : retourne l'élément suivant;

Exemple :

```
Collection<Integer> c = ...;
Iterator<Integer> it = c.iterator();  // l'itérateur dépend du type de la collection
while (it.hasNext()) {
   System.out.println(it.next());
}
```

Le parcours des collections itérateurs

Si la **collection** n'est **pas triée**, l'itérateur donne les valeurs dans un ordre quelconque.

HashSet = Set non trié

Exemple:

```
Set<String> s = new HashSet<String>();
s.add("3");
s.add("1");
s.add("4");
Iterator<String> it = s.iterator();
while(it.hasNext()) {
    System.out.println(it.next());
}
```

*affiche à l'écran **3 4 1** (ordre quelconque)*

Si la **collection** est **triée**, l'itérateur donne les valeurs dans l'ordre croissant.

Exemple:

```
Set<String> s = new TreeSet<String>();
s.add("3");
s.add("1");
s.add("4");
Iterator<String> it = s.iterator();
while(it.hasNext()) {
    System.out.println(it.next());
}
```

*affiche à l'écran **1 3 4***

L'interface ListIterator

L'interface *ListIterator<E>* hérite de *Iterator<E>* et définit des méthodes permettant de parcourir une **liste** dans les deux directions :

- ***boolean hasNext()*** : indique s'il reste des éléments pour un parcours en avant de la liste;
- ***boolean hasPrevious()*** : indique s'il reste des éléments pour un parcours en arrière de la liste ;
- ***E next()*** : retourne l'élément suivant ;
- ***E previous()*** : retourne l'élément précédent.

Exemple : affichage des éléments d'une liste du début à la fin puis de la fin au début

```
List<Integer> l = ...;
ListIterator<Integer> iterator = l.listIterator();
while(iterator.hasNext()) {
   System.out.println(iterator.next());
}
while(iterator.hasPrevious()) {
   System.out.println(iterator.previous());
}
```

Le parcours des collections
For each

3. Parcours par itérations simplifiées

- Manière plus légère pour itérer sur une collection (depuis Java 5) : ***for(Type var : Expression){ … }***
  Expression doit être une collection ou un tableau.
- A chaque itération la variable *var* de type *Type* est instanciée par l'élément suivant dans *Expression*. L'itération est finie quand il n'y a plus d'éléments.

```
ArrayList<Integer> al = new ArrayList<Integer>();
for(Integer n : al)
   System.out.println(n.intValue()*2);
```

ou même avec 'l'auto-unboxing' :

```
ArrayList<Integer> al = new ArrayList<Integer>();
for(int n : al)
   System.out.println(n*2);
```

Le parcours des collections
For each

- Avec **for each** Pas de modification possible de la collection ou du tableau.
- Il est possible d'utiliser des itérations simplifiées pour parcourir un **tableau** y compris avec des types de base :

```
int[ ] tab={1,2,3,4,5,6};
for (int val:tab)
    System.out.println(val);
```

Les tables – l'interface *Map<K, V>*

- Les **tables associatives** associent des **valeurs** à des **clefs** (table [K → V] en structure de données).
- L'interface *Map<K, V>* décrit des collections de couples (clé, valeur). Les clés ne peuvent pas être dupliquées. Pour chaque clé une *Map* fait correspondre une seule valeur :
  - ***V put(K key, V value)*** : associe *key* et *value*.
  - ***V get(Object key)*** : retourne la valeur associée à *key*.
  - ***V remove(Object key)*** : supprime le lien de cette *key* s'il existe.
  - ***boolean containsKey(Object key)*** : retourne *true* si *key* appartient au domaine.
  - ***boolean containsValue(Object value)*** : retourne *true* si value est associée à une clé.
  - ***Set<K> keySet()*** : le Set des clés.
  - ***Collection<V> values()*** : la Collection des valeurs.

L'interface *SortedMap(K, V)*

- **L'interface *SortedMap<K, V>*** décrit des tables triées en permanence par clés croissantes.
- De la même manière que pour *SortedSet<E>* les clés doivent implanter l'interface *Comparable*.

La classe concrète *HashMap<K,V>*

- **Classe *HashMap<K, V>*** : permet de créer des tables avec l'implémentation **rangement dispersé** (vue en structures de données).
  Ces tables utilisent en interne une **fonction de hachage** h(clé) pour calculer la position dans la table à partir de la clé. La taille de la table augmente **automatiquement** quand le taux de remplissage dépasse un seuil donné (0.75 par défaut).
  (HashMap implante aussi l'interface *Cloneable.)*
- Exemple : table qui associe à une chaîne représentant un nombre entier, ce nombre:
  HashMap<String, Integer>

clés	valeurs
"un"	1
"deux"	2
"trois"	3

La classe concrète *HashMap<K,V>*

Principaux constructeurs :

- ***HashMap()*** : construit une table vide avec une capacité par défaut de 11 et un taux de remplissage de 0.75
- ***HashMap(int initialCapacity)*** : construit une table vide avec une capacité donnée et un taux de remplissage de 0.75
- ***HashMap(int initialCapacity, float loadFactor)*** : construit une table vide avec une capacité et un taux de remplissage donnés.

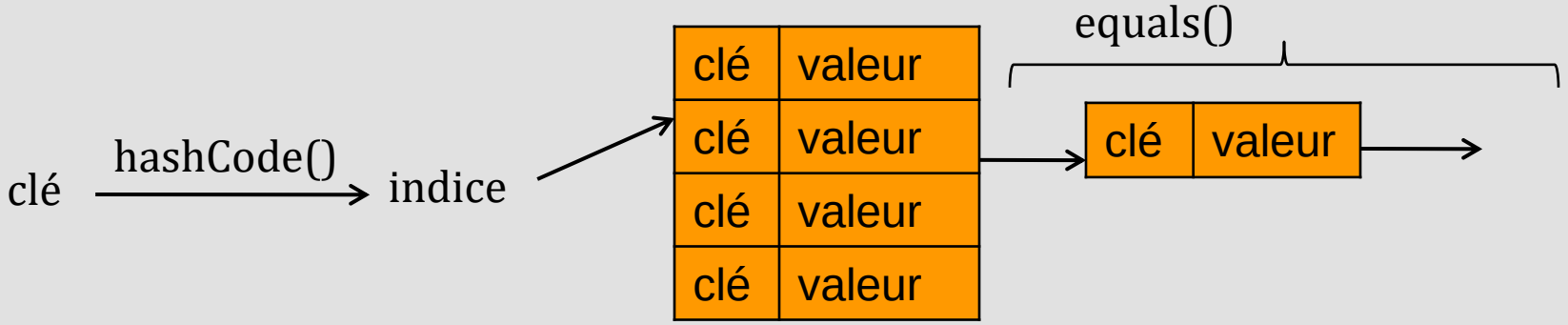
Principales méthodes :

Héritées de Map

- ***boolean containsValue(Object value)*** : teste si contient la valeur value
- ***boolean containsKey(Object key)*** : teste si contient la clé *key*
- ***V get(Object key)*** : retourne la valeur repérée par la clé *key*
- ***V put(K key, V value)*** : insère le couple (*key, value*)
- ***V remove(Object key)*** : supprime le couple repéré par la clé *key*
- ***Collection<V> values()*** : retourne la collection des valeurs
- ***Set<K> keySet()*** : retourne l'ensemble des clés
- ***Object clone()*** : duplique la table

Fonctionnement des *HashMap*

- A partir d'une clef de type K, un indice est calculé à l'aide de la méthode *hashCode()* de la classe K ;
- Un modulo sur l'indice calculé donne un indice compris dans l'intervalle voulu ;
- La case du tableau fournit une liste qui est parcourue pour trouver la clef cherchée (en faisant des tests d'égalité pendant le parcours).
- Les méthodes de la classe *HashMap<K,V>* reposent donc sur la définition des méthodes *hashCode* et *equals* de la classe K.



Fonctionnement des *HashMap*

- *Hashcode* et *equals* sont définies par défaut dans *Object.* La méthode *hashCode* de *Object* retourne une valeur calculée à partir de la référence de l'objet. La méthode *equals* de *Object* compare les références des objets. Les clefs sont donc identifiées par défaut par leurs **références**.
- Si on crée une *HashMap* qui associe un nombre à une *String*, tout fonctionne correctement car les méthodes *hashCode* et *equals* ont été correctement redéfinies par Java pour la classe *String* :

```
HashMap<String, Integer> m=new HashMap<String, Integer>();
m.put("un",new Integer(1));
m.put("deux",new Integer(2));
System.out.println(m.get(new String("un")));  // affiche bien 1
```

Fonctionnement des *HashMap*

- Si on a une classe où *hashcode* et *equals* ne sont pas redéfinies cela peut poser problème car les clefs sont identifiées par leurs références.

```
HashMap<Point, Integer> mp=new HashMap<Point, Integer>();
Point p=new Point(0, 0);
mp.put(p, new Integer(1));
System.out.println(mp.get(p));              // retourne bien 1
System.out.println(mp.get(new Point(0, 0))); // retourne null car ce //
    n'est pas le même objet Point (références différentes)
```

Si on ne redéfinit que *equals* (retourne vrai ssi 2 points ont les mêmes coordonnées), *hashCode* continue à disperser les clefs selon les références : pour 2 points de mêmes coordonnées on obtient 2 indices différents.

Si on ne redéfinit que *hashCode,* 2 points de mêmes coordonnées sont stockés dans la même liste mais comme *equals* reste basée sur les références, ces 2 clefs sont considérées comme différentes dans le parcours de la liste chaînée.

Il faut donc redéfinir ***hashCode*** **ET** ***equals*** et leur donner des comportements qui correspondent !

Fonctionnement des *HashMap*

- La **capacité** désigne le nombre d'éléments dans le tableau permettant de stocker la liste des clefs. Plus la capacité est grande plus les indices sont dispersés.
- Le **taux de remplissage** est un facteur qui détermine quand il faut restructurer la table. Vaut par défaut 0.75. Dès que le nombre de clefs dépasse la capacité multipliée par ce taux Java restructure automatiquement la table en doublant sa capacité.
- Trouver une **bonne méthode *hashCode* est difficile**. Il faut qu'elle répartisse correctement les éléments en évitant des accumulations pour certaines valeurs. Cela dépend du type des objets et de leurs occurrences.

  Ex: associer à la classe Personne le rang de la première lettre du nom n'est pas une bonne idée; il y a seulement 26 valeurs et le 3 (lettre C) arrivera plus souvent que le 26 (lettre Z).

Fonctionnement des *HashMap*

- Une solution classique utilise une **combinaison linéaire** des fonctions de *hashCode* des attributs utilisés avec des coefficients premiers.

  Ex: pour identifier des Point par leurs coordonnées x et y

```
public class PointHashCode {
    int x,y;
    public int hashCode() {
      final int premier = 31;
      int result = 1;
      result = result * premier + x;          // x est son propre hashcode
      result = result * premier + y;          // y est son propre hashcode
      return result;                          // 31*x + y + 31*31
    }
    public boolean equals(Object o) {         // utilise les même attributs x et y
      PointHashCode p = (PointHashCode) o;
      if (x != p.x) return false;
      if (y != p.y) return false;
      return true;
   }
}
```

Parcours des tables

- Il est possible de parcourir une table *Map<K,V>* en utilisant la méthode *keySet()* qui retourne l'ensemble des clefs sous la forme d'un *Set<K>*.

```
Map<String,Integer> m=new HashMap<String,Integer>();
m.put("un",1);
m.put("deux",2);
m.put("trois",3);
Set<String> clefs=m.keySet();
for (String s:clefs)
  System.out.println("clef : "+s+" ,valeur : "+m.get(s));
```

Les algorithmes

- La classe *Collections* (avec un s ! à ne pas confondre avec l'interface Collection) offre des **algorithmes prédéfinis** sous la forme de méthodes statiques.

  Exemple :

  static <T> void sort(List<T> l, Comparator<? super T> c)

  Remarque: comme les classes, les méthode peuvent être paramétrées par un type (« méthodes génériques »). On peut appeler *sort* par *<Integer>sort(...)*. Mais ce n'est pas obligatoire !

- *Collections* fournit des algorithmes opérant sur des collections (souvent des *List*), en particulier :
  - **tri** et de mélange,
  - **recherche**,
  - remplissage, **copie**, renversement, etc.

La classe *Collections* (avec un s !) Principales méthodes :

- ***static <T extends Comparable<?, super T> void sort(List<T> list)*** **:** tri avec la méthode *int compareTo(T o)* de l'interface *Comparable<T>*

  → *1 seul critère de tri possible !*
- ***static <T> void sort(List<T> list, Comparator<? super T> c)*** : tri avec la méthode *int compare(T o1, T o2)* de l'interface *Comparator<T>*

  → *plusieurs critères de tri possibles avec des objets de types différents qui implantent Comparator<T> en 2ème paramètre*
- ***static void reverse(List<?> l)*** : inversion de la liste
- ***static <T> void fill(List<? super T> list, T o)*** : remplissage de la liste avec l'objet o (remplace tous les éléments par o).
- ***static <T> void copy(List<? super T> dest, List<? extends T> src) :*** recopie de la liste src dans dest

- Les interfaces *Comparable* et *Comparator* sont génériques et des instances appropriées doivent être utilisées. Par exemple, pour trier une liste d'instances de la classe Etudiant, une classe implantant l'interface ***Comparator<Etudiant>*** doit être utilisée avec la méthode *sort* de la classe *Collections*.

La classe *Collections* (avec un s !) Exemple :

Exemple : ll est une *LinkedList<Integer>*

```
import java.util.*;
public class TestLL {
  public static void main(String[] args) {
    LinkedList<Integer> ll = new LinkedList<Integer>();
    ll.add(13);ll.add(8);ll.add(2);ll.add(5);
    System.out.println("Avant : "+ll);
    Collections.sort(ll);      // Integer implements Comparable<Integer>
                               // => méthode compareTo dans Integer
   // ou Collections.sort(ll, new ElementComparator());
    System.out.println("Apres : "+ll);
  }
}
```

Avant : [13, 8, 2, 5]
Apres : [2, 5, 8, 13]

```
// avec :
// class ElementComparator implements Comparator<Integer> {
//    public int compare(Integer o1, Integer o2) {
//      return -(o1.compareTo(o2));
//    }
// }
```

Avant : [13, 8, 2, 5]
Apres : [13, 8, 5, 2]