Algorithmie & Programmation - Cours 4

Les types de données abstraits

Dr. Jérémie Sublime

LISITE Laboratory - RDI Team - ISEP

jeremie.sublime@isep.fr

Plan

Plan

Rappels des cours précédents

Dans les cours précédents nous avons étudié les tableaux :

- Une structure permettant de stocker des données de même type (entiers, flottants, chaines de caractères, etc.)
- Nous avons vu qu'il était même possible de créer des tableaux multi-dimensionnels
- Les tableaux sont des structures simples et indexées se prêtant facilement au stockage et à la manipulation de données avec divers algorithmes (e.g : algorithmes de tri).

Index	0	1	2	3	4	5	6
Valeur	45	154	58	78	31	5	74

Table: Exemple de tableau avec des entiers

Limites de l'allocation statique (1/2)

Les tableaux sont pratiques, mais ont un énorme inconvénient : Ils sont contraints

- Leur taille est fixée au début de programme lors de la déclaration du tableau (dans beaucoup de langages dont Java).
- Une fois fixée, cette taille ne peut plus bouger.

Exemple de problème avec l'allocation statique

Imaginons que nous réalisions un jeu d'aventure. Notre personnage découvre des trésors au fil du jeu, qu'il ajoute à son coffre à trésor. Ce serait idiot de créer un objet coffre de type Tableau. S'il est prévu pour contenir 100 objets et que l'on en découvre 101, il sera impossible d'enregistrer notre 101ème objet dans le coffre ! Il serait judicieux que le coffre puisse contenir autant de trésors qu'on le souhaite.

Limites de l'allocation statique (2/2)

Pour contourner le problème de l'allocation statique, il est nécessaire d'utiliser ou de créer des containers spécifiques dont la taille peut être dynamique.

Idée

- Nous avons vu que les tableaux à 2 dimensions étaient en fait des tableaux de pointeurs vers d'autres tableaux.
- On pourrait imaginer des structures récursives utilisant des pointeurs pour faire de l'allocation dynamique.

- Les types de données abstraits répondent à cette problématique.

Plan

Listes chaînées (1/3)

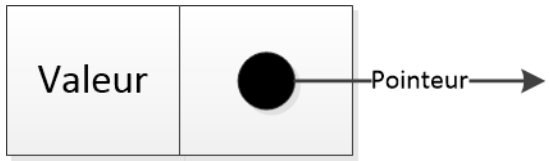
Nous allons nous intéresser à une premier type abstrait : Les listes chaînées.

Composition d'une liste chaînée

Une liste chaînée est un **objet** récursif composé de :

- Un emplacement mémoire pour stocker une valeur (entier, flottant, chaine, objet, etc.)
- Un "**pointeur**" vers un autre objet de même type liste.

Chaque objet est donc à la fois une liste et un maillon de la liste.



Listes chaînées (2/3)

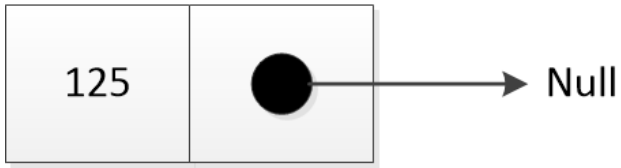


Figure: Exemple de liste chaînée avec un élément unique

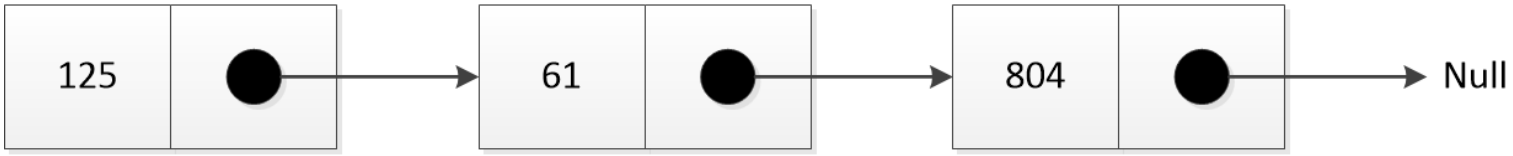


Figure: Exemple de liste chaînée avec 3 éléments

Listes chaînées (3/3)

Points forts des listes chaînées

- Les listes ne sont limitées en taille que par la mémoire de l'ordinateur
- Leur taille peut être changée de façon dynamique et n'a pas besoin d'être prédéfinie

Points faibles des listes chaînées

- On ne peut pas accéder directement à un élément donné : Il faut parcourir toute la liste jusqu'à cet élément.
- Avec ce chaînage simple, chaque élement de la liste n'a accès qu'aux éléments suivants, les précédents sont inaccessibles.

Remarque : Les tableaux sont en réalités des listes contraintes (en taille) et indexées (pour l'accès direct).

Listes chaînées : Implémentation Java (1/2)

En Java, une liste s'implémente dans une classe à part:

```
public class ListeEntier{
   int entier;
   ListeEntier suivant;
}
```

Listes chaînées : Implémentation Java (1/2)

En Java, une liste s'implémente dans une classe à part:

```
public class ListeEntier{
   int entier;
   ListeEntier suivant;
}
```

Une liste peut aussi contenir plus d'une valeur:

```
public class ListeEtudiant{
   int id;
   String prenom;
   String nom;
   ListeEtudiant suivant;
}
```

Listes chaînées : Implémentation Java (2/2)

- Une liste étant un **objet**, on créée une nouvelle liste avec le mot clé **new**.
- Des éléments peuvent être ajoutés à la suite de la liste grâce au pointeur vers l'élément suivant.
- De la même façon, on parcourt une liste en descendant les pointeurs.

```
 //création de la liste avec un 1er étudiant
ListeEtudiant maListe = new
    ListeEtudiant(1,"Antoine","Dupont");
//ajout d'un second étudiant
maListe.suivant = new ListeEtudiant(2,"Asma","Touari");
```

Remarque

Nous verrons dans les prochains cours que les listes étant des **objets**, il est possible de coder des méthodes interne pour les parcourir, ajouter des éléments, en supprimer, les trier, etc.

Parcours de listes simples

Exemple de pseudo-code récursif pour parcourir une liste.

```
FONCTION afficheListe(L : liste)
        SI L==Null ALORS
                Retourner ;
        SINON
                AFFICHER(L.valeur)
                afficheListe(L.suivant)
        FIN SI;
Fin FONCTION
```

Insertion en début de liste simple

```
FONCTION InsererTete(L : liste, valeur : item)
        liste ancien = L
        nouvelleListe = nouvelle Liste(valeur)
        nouvelleListe.suivant=L
        Retourner nouvelleListe;
Fin FONCTION
```

Insertion en fin d'une liste simple

```
FONCTION InsererFin(L : liste, valeur : item)
        liste courant = L
        TANT QUE courant.suivant!=null
                courant = courant.suivant
        FIN TANT QUE
        courant.suivant = Nouvelle Liste(valeur)
        Retourner L;
Fin FONCTION
```

Insertion en milieu de liste simple

```
FONCTION InsererPos(L : liste, valeur : item, entier : pos)
        liste courant = L
        entier i = 0;
        TANT QUE (courant.suivant!=null ET i!=pos)
                courant = courant.suivant
                i++
        FIN TANT QUE
        liste FinListe=courant.suivant
        courant.suivant = Nouvelle Liste(valeur)
        courant.suivant.suivant=FinListe
        Retourner L;
Fin FONCTION
```

Les listes doubles chaînées

Les listes doubles chaînées sont très similaires aux listes chaînées simples :

- Chaque élément se compose des valeurs, d'un pointeur vers l'élément d'avant et d'un pointeur vers l'élément d'après.
- Le pointeur vers l'élément d'avant permet de remonter dans la liste, ce qu'on ne pouvait pas faire avec un chaînage simple.

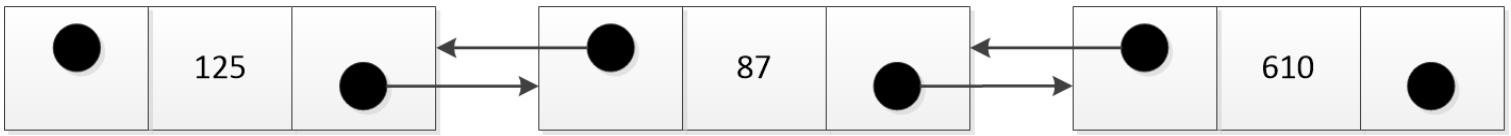


Figure: Exemple de liste double chaînée avec 3 éléments

Les arbres

Les arbres sont comme les listes chaînées simples sauf qu'ils peuvent pointer sur plusieurs éléments fils.

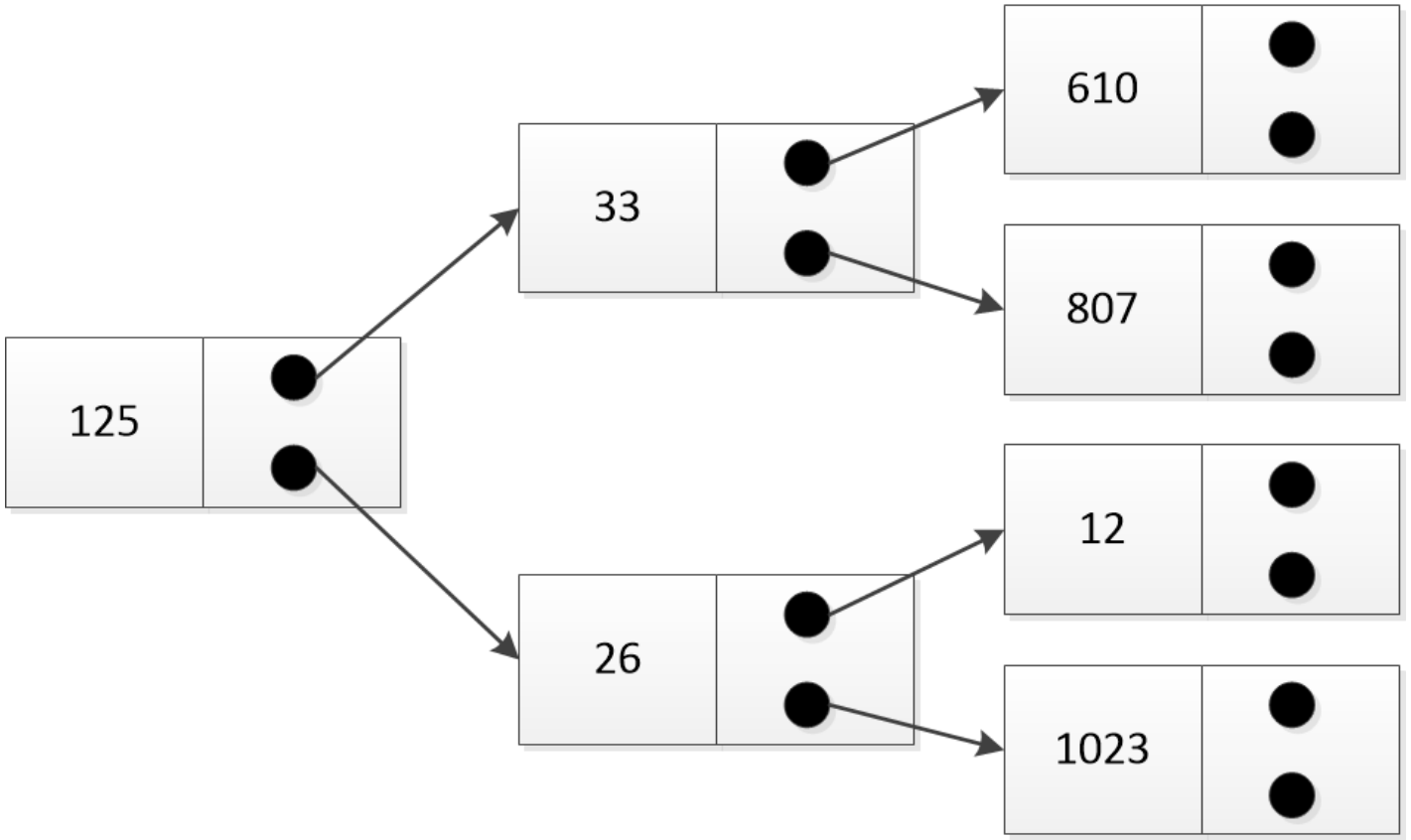


Figure: Exemple d'un arbre binaire (deux successeurs)

Les listes cycliques

Les listes cycliques sont des listes chaînées simples dont le pointeur du dernier élément de la liste pointe toujours vers le 1er élément :

- Il est ainsi possible de parcourir toute la liste avec un seul pointeur par noeud.
- Cette architecture ne permet pas de trier ou de compter les éléments de la liste de façon efficace.

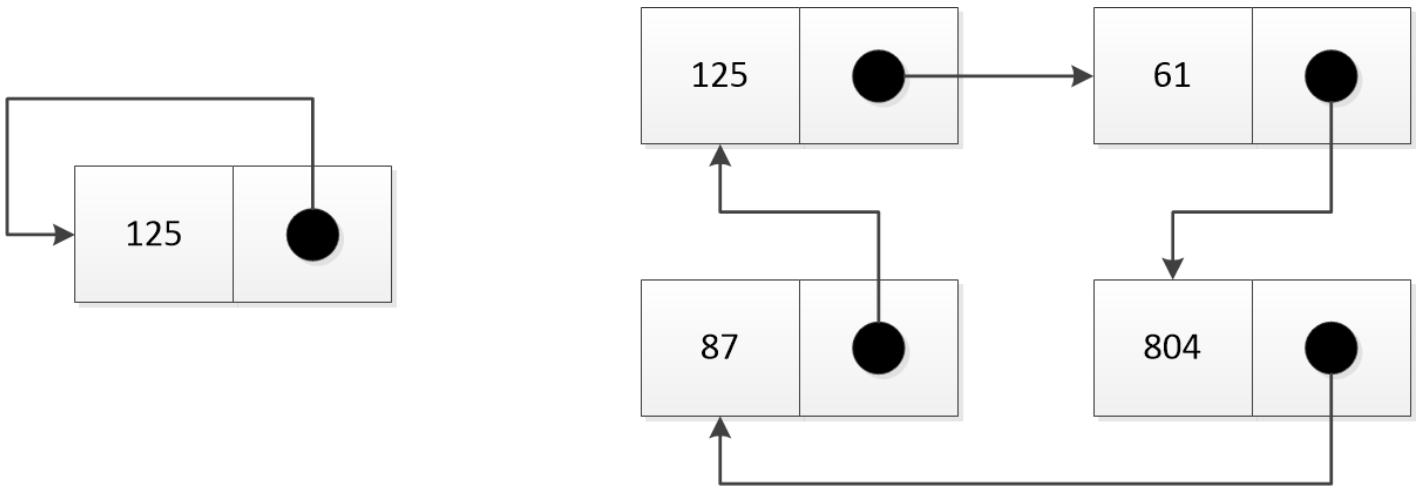
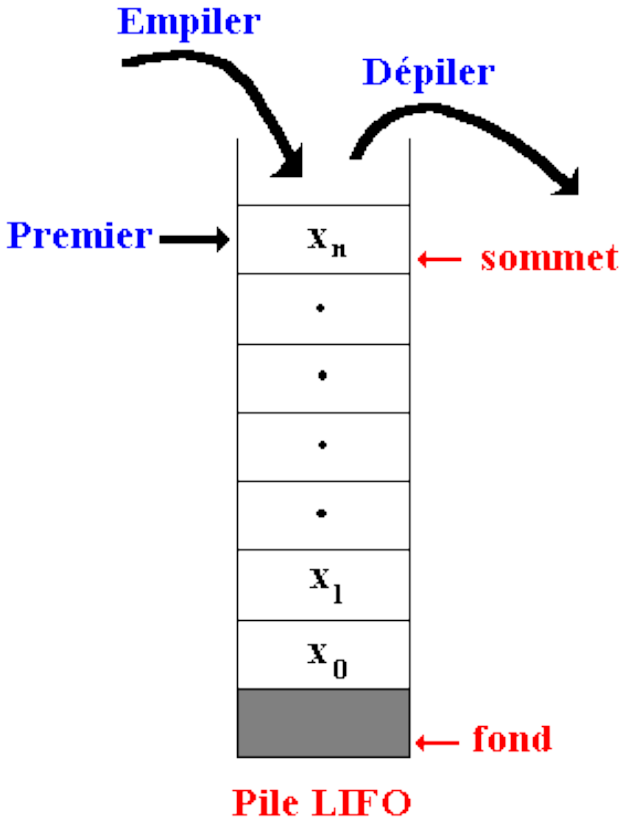


Figure: Exemples de listes cycliques avec 1 et 4 éléments

Les piles LIFO

Les piles sont un type particulier de liste :

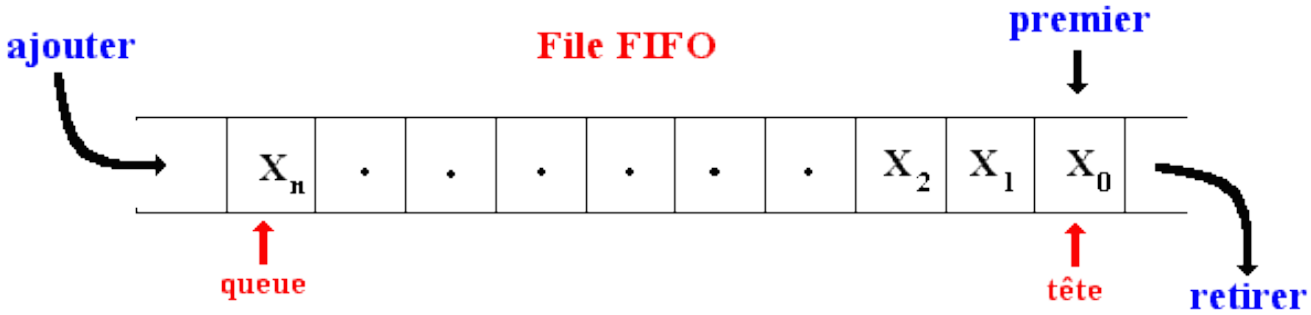
- On ne peut ajouter, accéder ou retirer des éléments qu'en début de liste : Last In First Out (LIFO)
- Les piles ont trois opérations principales :
  - Empiler (push) : Rajouter un élément à la pile
  - Dépiler (pop) : Retirer un élément
  - Montrer (peek) : Renvoie le contenu du 1er élément sans le retirer.



Les files FIFO

Les files sont un autre type particulier de liste :

- On ne peut ajouter des éléments qu'en fin de liste et ne retirer les éléments qu'en début de liste : First In First Out (FIFO)
- Les piles ont trois opérations principales :
  - Ajouter (put) : Rajouter un élément en fin de liste
  - Retirer (take) : Retirer le 1er élément de la liste
  - Montrer (peek) : Renvoie le contenu du 1er élément sans le retirer.



Les maps

- Une map désigne n'importe quel collection d'objet (liste simple ou double, pile, file, etc.) qui à une clé associe une valeur.
- Une map se différence d'un tableau car elle est de taille dynamique.

Plan

Types abstraits en Java

Il existe de nombreux types abstraits pré-implémentés en Java :

- Tableaux extensibles à volonté : List, Vector, ArrayList, LinkedList
- Maps (liste de couples clé/valeur) : HashMap, Hashtable, TreeMap, etc.
- Collections (pas d'objets en double) : Set

Types abstraits en Java

Il existe de nombreux types abstraits pré-implémentés en Java :

- Tableaux extensibles à volonté : List, Vector, ArrayList, LinkedList
- Maps (liste de couples clé/valeur) : HashMap, Hashtable, TreeMap, etc.
- Collections (pas d'objets en double) : Set

Avant d'utiliser directement ces types qui nécessitent des connaissances en programmation orientée objet, en TP nous apprendrons à recoder plusieurs de ces objets.

Les ArrayList

Les *ArrayList* sont le type correspondant aux listes chaînées simples en Java.

- Les ArrayList acceptent des éléments de types différents, y compris *null* !

Packages

Les packages suivants sont nécessaires pour utiliser les ArrayList :

- *java.util.ArrayList* : Pour le type LinkedList
- *java.util.ListIterator* (facultatif) : Pour parcourir les listes

Exemple : Les ArrayList

```
public class Test {

  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 ArrayList: propriétés

Les ArrayList sont des **objets**. Elles ont dons des méthodes pré-codées :

- *size()* : renvoie le nombre d'éléments de la liste
- *add(object element)* : rajouter un élément
- *get(int index)* : retourne l'élément d'indice *index*
- *remove(int index)* : supprime l'élément de l'indice *index*
- *isEmpty()* : indique si la liste est vide
- *removeAll()* : efface tout le contenu de la liste
- *contains(Object element)* : retourne "true" si l'élement est dans la liste

Les LinkedList

Une liste double 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 LinkedList Java acceptent aussi des éléments de types différents et ont les mêmes méthodes que les ArrayList.

Packages

Les packages suivants sont nécessaires pour utiliser les LinkedList :

- *java.util.LinkedList* : Pour le type LinkedList
- *java.util.List* : Pour le type List
- *java.util.ListIterator* : Pour parcourir les listes

Exemple : Les LinkedList

```
public class Test {

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

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

  }
}
```

Les itérateurs

Un itérateur est un objet qui a pour rôle de parcourir une collection : List, ArrayList, LinkedList, etc.

```
public class Test {
  public static void main(String[] args) {
    List l = new LinkedList();
    l.add(12);
    l.add("toto ! !");
    l.add(12.20f);

      ListIterator li = l.listIterator();

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

  }
}
```

ArrayList vs LinkedList

- Les ArrayList sont plus rapides que les LinkedList en lecture, même avec un gros volume d'objets.
- Les LinkedList sont plus rapides pour ajouter ou supprimer un objet en milieu de liste.

ArrayList vs LinkedList : En résumé

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 Maps

- Une collection de type Map est une collection qui fonctionne avec un couple clé/valeur.
- On y trouve les objets *Hashtable*, *HashMap*, *TreeMap*, *WeakHashMap*, etc.
- La clé, qui sert à identifier une entrée dans notre collection, est unique.
- La valeur, au contraire, peut être associée à plusieurs clés.

L'objet Hashtable

Packages

Les packages suivants sont nécessaires pour utiliser les Hashtable :

- *java.util.Hashtable*
- *java.util.Enumeration* : pour parcourir une map.

L'objet Hashtable

Packages

Les packages suivants sont nécessaires pour utiliser les Hashtable :

- *java.util.Hashtable*
- *java.util.Enumeration* : pour parcourir une map.

- *isEmpty()* : indique si la map est vide
- *contains(Object value)* et *containsValue(Object value)* : retourne "true" si l'élement est dans la map.
- *containsKey(Object key)* : retourne "true" si la clé existe.
- *put(Object key, Object value)* : ajoute le couple key/value dans la map.
- *elements()* : retourne une **énumération** des éléments.
- *keys()* : retourne la liste des clés sous forme d'**énumération**.

Exemple : Hashtable

```
public class Test {
  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());
    }

  }
}
```

Hashtable vs HashMap

Les objets Hashtable et HashMap sont quasi-identiques, les différences sont les suivantes :

- HashMap accepte les objets *null* ce qui n'est pas le cas de Hashtable
- Hashtable peut être **multi-threadé**, pas HashMap.

Les sets en Java

- Un Set est une collection qui n'accepte pas les doublons. (même sur l'objet *null*)
- On trouve parmi les Set les objets *HashSet*, *TreeSet*, *LinkedHashSet*, etc.
- Certains Set sont plus restrictifs que d'autres : il en existe qui n'acceptent pas null, certains types d'objets, etc.

L'objet HashSet

Packages

Les packages suivants sont nécessaires pour utiliser les HashSet :

- *java.util.HashSet*
- *java.util.Iterator* : pour parcourir un set.

L'objet HashSet

Packages

Les packages suivants sont nécessaires pour utiliser les HashSet :

- *java.util.HashSet*
- *java.util.Iterator* : pour parcourir un set.

- *isEmpty()* : indique si le set est vide
- *contains(Object value)* : retourne "true" si la valeur est dans le set.
- *add(Object value)* : rajoute un élément dans le set.
- *remove(Object o)* : Retire l'objet *o* du set.
- *iterator()* : retourne un **iterator**.
- *toArray()* : retourne un tableau des objets.

Exemple : HashSet (1/2)

```
public class Test {
  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());
    }

  }
}
```

Exemple : HashSet (2/2)

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

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

  }
}
```

Plan

Dans les prochains cours

- Programmation orientée objet (2 cours et 3 TP)
- Interfaces graphiques (2 TP)
- Gestion d'erreurs et lecture de fichiers (1 Cours)
- Projet (2 séances de TP dédiées + 1 séance de POO)