Le langage JAVA

Les collections

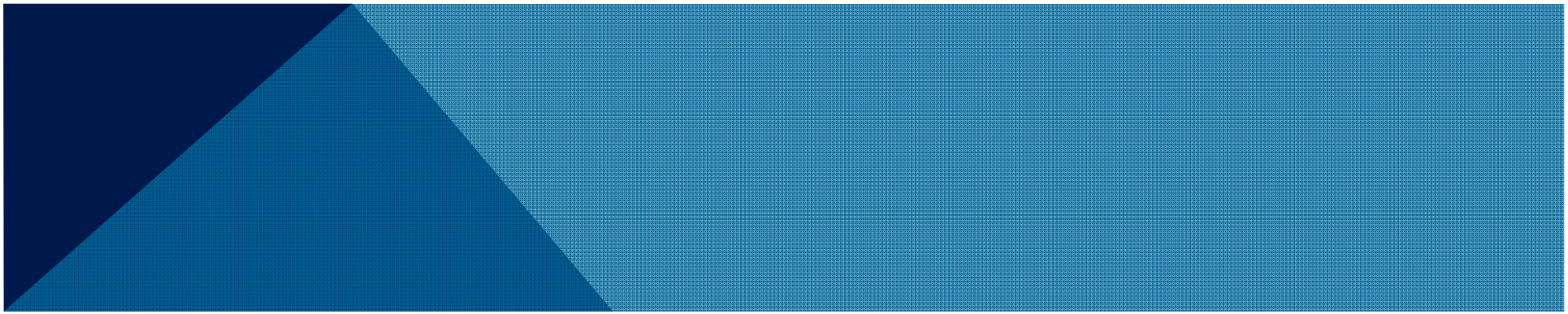
INTRODUCTION, DÉFINITION, MOTIVATION

- Collection : objet (conteneur) qui regroupe de multiples éléments (pas forcément de même type) dans une seule structure.
- Utilisation de collections pour :
  - ✓ stocker, rechercher et manipuler des données
  - ✓ transmettre des données d'une méthode à une autre
- Objectifs
  - ✓ adapter la structure collective aux besoins de la collection
  - ✓ ne pas re-programmer les traitements répétitifs classiques (affichage, saisie, recherche d'éléments, …)
- Exemples :
  - ✓ un dossier de courrier : collection de mails
  - ✓ un répertoire téléphonique : collection d 'associations noms/numéros de téléphone.
- Par exemple, un tableau est une collection
- Le JDK fournit d'autres types de collections sous la forme de classes et d'interfaces. Ces classes et interfaces sont dans le paquetage **java.util**

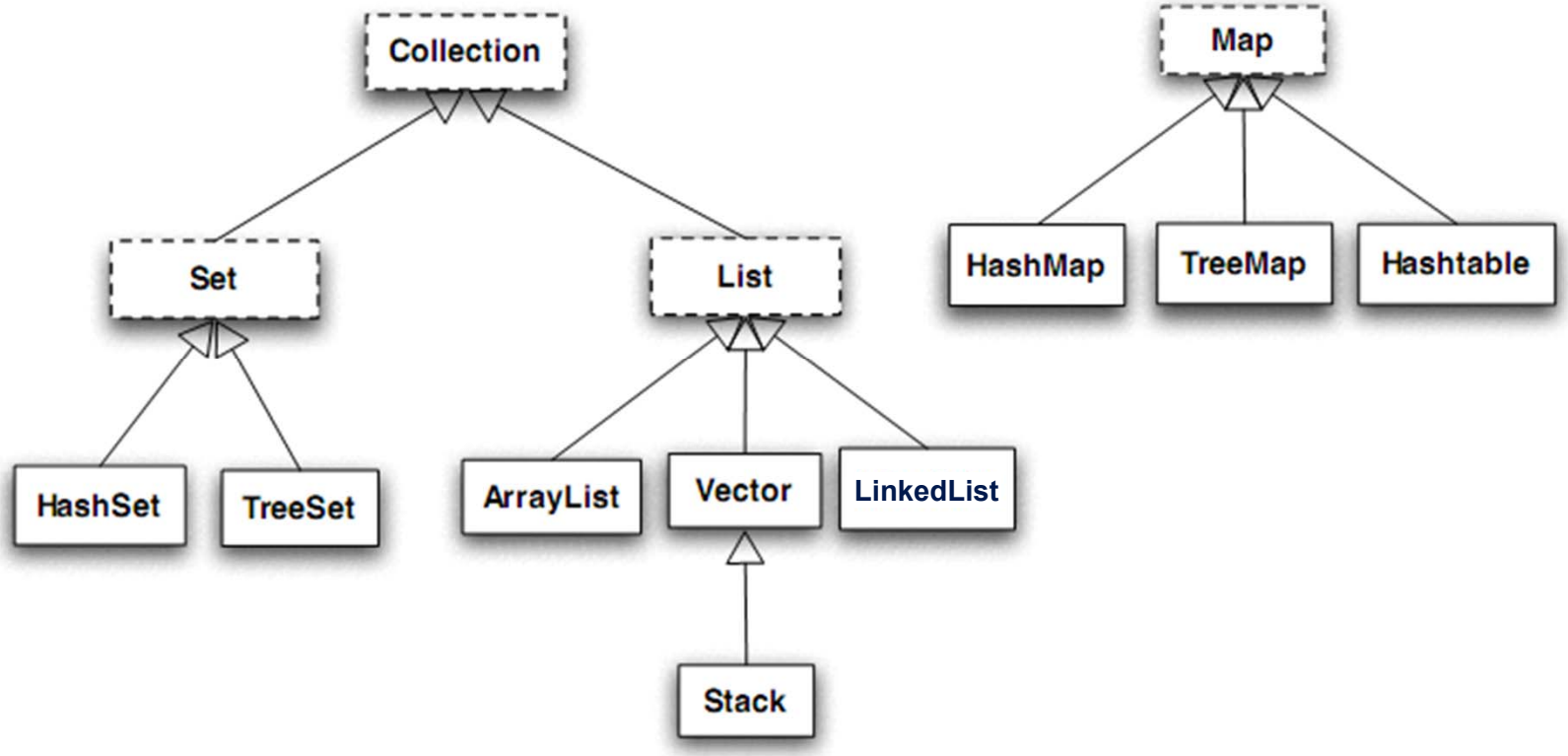
LES COLLECTIONS

- **Collections = structures de données**
  - ✓ Listes
  - ✓ Ensembles
  - ✓ Tableaux
  - ✓ Arbres
  - ✓ Tables de hashage
  - ✓ …

- **Ces structures sont complexes à implanter (⇒ Efficacité)**
  - ✓ Ordonnées ou non
  - ✓ Doublons ou non
  - ✓ Accès aux données indexées ou non
  - ✓ Recherche
  - ✓ Tris



ARBORESCENCE DES COLLECTIONS EN JAVA



Les collections Java contiennent par défaut des éléments de type Object

STRUCTURES COLLECTIVES CLASSIQUES

- **Tableau**
  - ✓ accès par index
  - ✓ recherche efficace si le tableau est trié (par dichotomie)
  - ✓ insertions et suppressions peu efficaces
  - ✓ défaut majeur : nombre d'éléments borné

`type[]` et `Array`

- **Liste**
  - ✓ accès séquentiel : premier, suivant
  - ✓ insertions et suppressions efficaces
  - ✓ recherche lente, non efficace

`interface List`

- **Tableau dynamique = tableau + liste**

`class ArrayList`

GÉNÉRICITÉ DES COLLECTIONS

- Avant le JDK 5.0, les collections peuvent contenir des objets de n'importe quel type. Pour les types primitifs, il faut donc utiliser explicitement les classes enveloppantes des types primitifs, Integer par exemple.

- A partir du JDK 5.0, on peut indiquer le type des objets contenus dans une collection grâce à la généricité, sachant que, par défaut, ce sont des objets de la classe **Object** :

  Collection < TypeDObjet >

  Pour les types primitifs, les conversions entre les types primitifs et les classes enveloppantes peuvent être implicite avec le « boxing » /« unboxing ».

LES INTERFACES DES COLLECTIONS

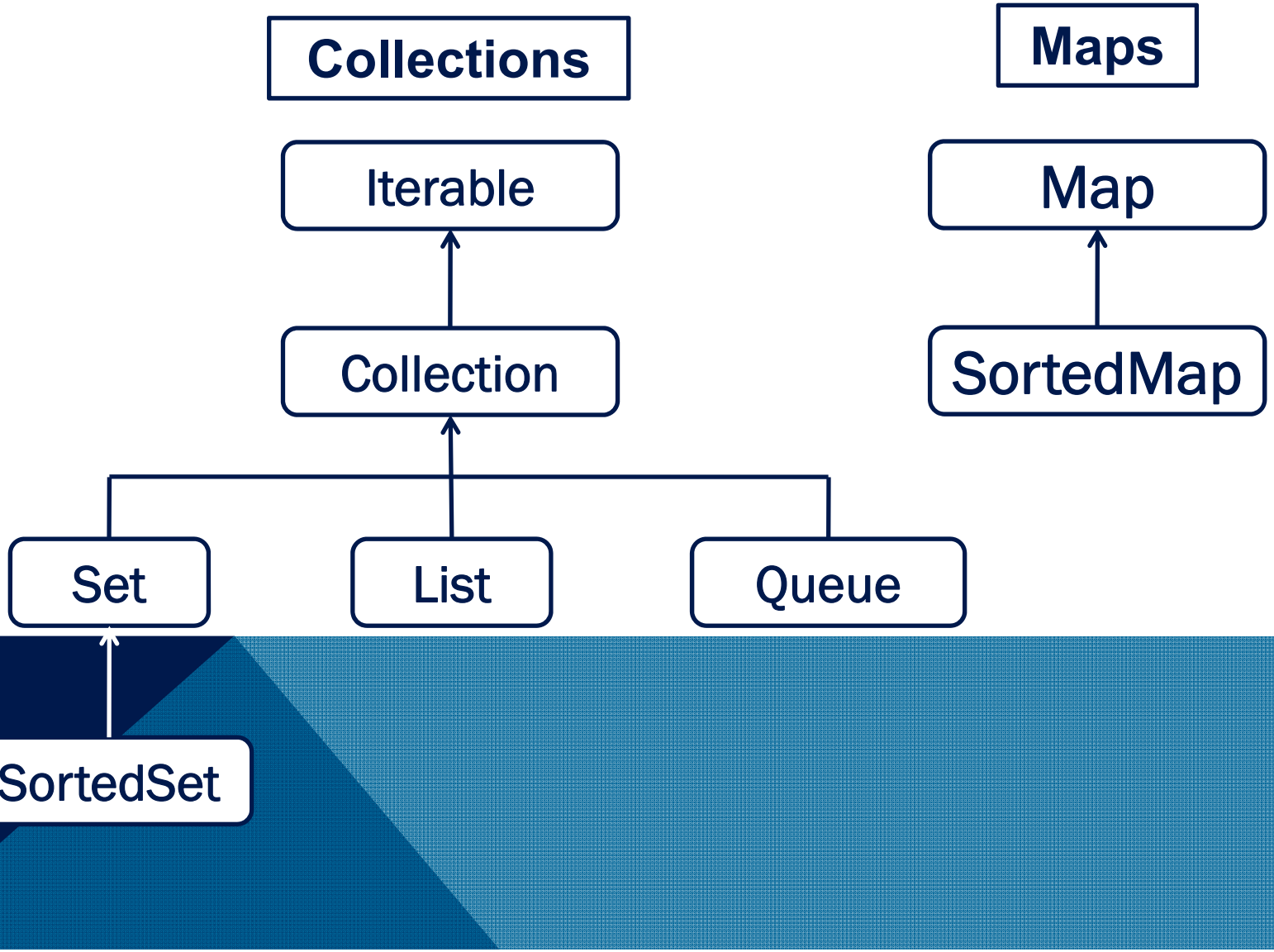
- Il y a des interfaces dans 2 hiérarchies principales :

Collection <E>
Map <K,V>

- **Collection** correspond aux interfaces des collections proprement dites

- **Map** correspond aux collections indexées par des clés ; un élément de type V d'une map est retrouvé rapidement si on connaît sa clé de type K (comme les entrées d'un dictionnaire ou les entrées de l'index d'un livre)

HIÉRARCHIE DES INTERFACES POUR LES COLLECTIONS



QUELQUES STRUCTURES DE DONNÉES CLASSIQUES

- **Vector :** implante un tableau redimensionnable dynamiquement; les éléments sont indexables

- **ArrayList :** liste ordonnée implantée sous la forme d'un tableau; remplace Vector aujourd'hui

- **HashSet** : ensemble d'éléments, non ordonnés, sans doublons, avec les opérations d'union et d'intersection

- **Hashtable :** implante une table de hashage : *élement = clef* $\rightarrow$ *valeur*

CLASSES CONCRÈTES D'IMPLANTATION DES INTERFACES

		Classes d'implantation			
		Table de hachage	Tableau de taille variable	Arbre balancé	Liste chaînée
Interfaces	Set<E>	HashSet<E>		TreeSet<E>	
	List<E>		ArrayList<E>		LinkedList<E>
	Map<K,V>	HashMap<K,V>		TreeMap<K,V>	

EXEMPLE DE LISTE

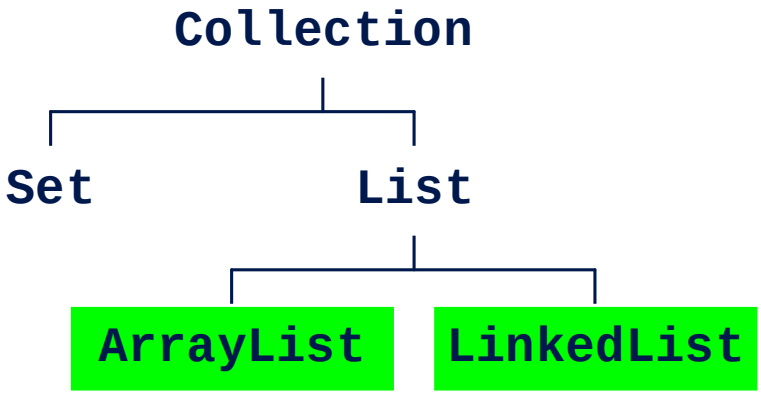
```
public class ExempleListe {

    public static void main(String[] args) {

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

       l.add("Pierre Jacques") ;   // ajout
       l.add("Pierre Paul");
       l.add("Jacques Pierre");
       l.add("Paul Jacques");
       l.remove("Pierre Paul") );   // suppression

       Collections.sort(l);   // tri de la collection
       System.out.println(l);
    }
}
```

EXEMPLE DE MAP

```
public class ExcempleMap {

    public static void main(String[] args) {

       Map <String, Integer> frequences = new HashMap <String, Integer> ();

       for (String mot : args) {
              Integer freq = frequences.get(mot);
              if (freq == null)
                     freq = 1;
              else
                     freq = freq + 1;
              frequences.put(mot, freq);
       }
       System.out.println(frequences);
    }
}
```

PAQUETAGE JAVA.UTIL

- Interface **Collection**
- Interfaces **Set** et **List**
- Méthodes :
  - ✓ **boolean add(Object o)**
  - ✓ **boolean remove(Object o)**
  - ✓ ...
- Plusieurs implantations :
  - ✓ tableau : **ArrayList**
  - ✓ liste chaînée : **LinkedList**



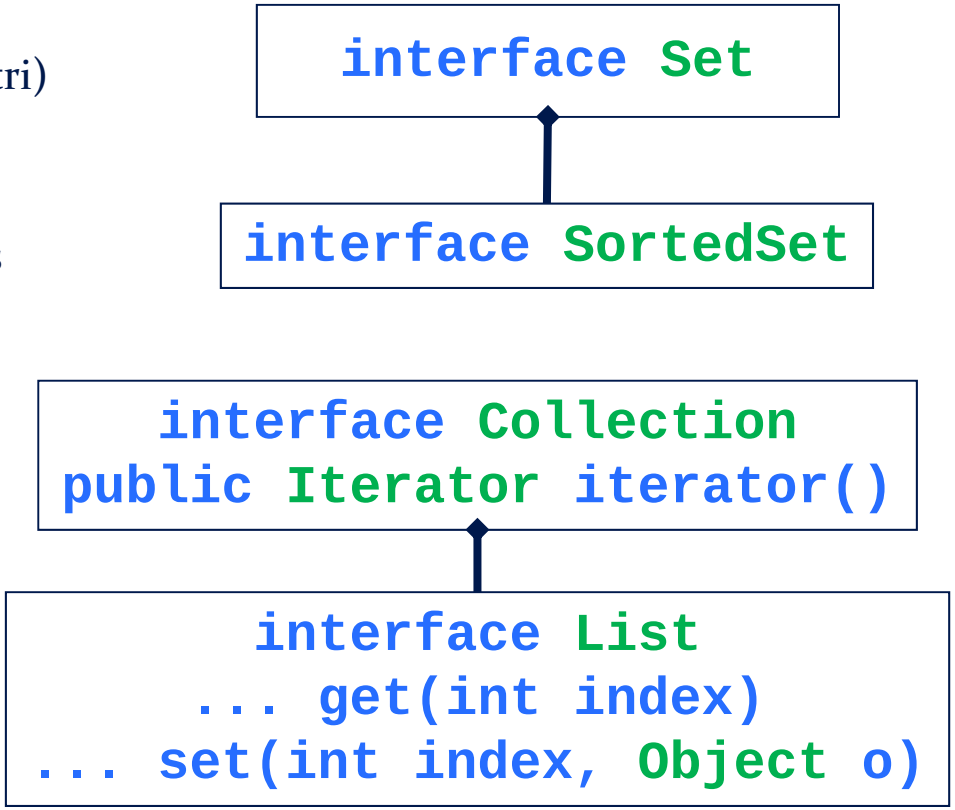
- Les algorithmes génériques disponibles sont : tri, maximum, copie ...
  - ✓ Accessibles par les **méthodes statiques de Collection**

MÉTHODES COMMUNES DES COLLECTIONS

`boolean add(Object)` : ajouter un élément
`boolean addAll(Collection)` : ajouter plusieurs éléments
`void clear()` : tout supprimer
`boolean contains(Object)` : test d'appartenance
`boolean containsAll(Collection)` : appartenance collective
`boolean isEmpty()` : test de l'absence d'éléments
`Iterator iterator()` : pour le parcours (cf Iterator)
`boolean remove(Object)` : retrait d'un élément
`boolean removeAll(Collection)` : retrait de plusieurs éléments
`boolean retainAll(Collection)` : intersection
`int size()` : nombre d'éléments
`Object[] toArray()` : transformation en tableau
`Object[] toArray(Object[] a)` : tableau de même type que a

CARACTÉRISTIQUES DES COLLECTIONS

- Ordonnées ou non :
  - ✓ Ordre sur les éléments ? (voir tri)
- Doublons autorisés ou non :
  - ✓ liste (**List**) ➔ avec doubles
  - ✓ ensemble (**Set**) ➔ sans doubles
- Besoins d'accès :
  - ✓ indexé
  - ✓ séquentiel, via **Iterator**



FONCTIONNALITÉS DES LISTES

- Ces collections implantent l'interface **List** :

  - ✓ **`ArrayList`**
    - Liste implantée dans un tableau
    - accès immédiat à chaque élément
    - ajout et suppression lourdes

  - ✓ **`LinkedList`**
    - Liste implantée dans une liste chaînée
    - accès aux éléments lourd
    - ajout et suppression très efficaces
    - permettent d'implanter les structures FIFO (file) et LIFO (pile)
    - méthodes supplémentaires : **addFirst(), addLast(), getFirst(), getLast(), removeFisrt(), removeLast()**

FONCTIONNALITÉS DES ENSEMBLES

- Ces collections implantent l'interface **Set** :
  - ✓ ➔ Éléments non dupliqués
  - ✓ **`HashSet`**
    - Structure de table de hashage
    - utiliser pour cela la méthode **hashCode()**
    - accès très performant aux éléments
  - ✓ **`TreeSet`**
    - Structure d'arbre binaire de recherche
    - maintient l'ensemble trié en permanence
    - méthodes supplémentaires : **first()** (mini), **last()** (maxi), **subSet(deb,fin)**, **headSet(fin)**, **tailSet(deb)**

RECHERCHE D'UN ÉLÉMENT

- Méthode
  - ✓ **public boolean contains(Object o)**
  - ✓ Dans l'interface Collection et redéfinie selon les sous-classes
- Utilise l'égalité entre objets :
  - ✓ égalité définie par **boolean equals(Object o)**
  - ✓ par défaut (classe **Object**) : égalité de références
  - ✓ à redéfinir dans chaque classe d'éléments
- Cas spéciaux
  - ✓ doublons : recherche du premier ou de toutes les occurrences ?
  - ✓ structures ordonnées : plus efficace si les éléments sont comparables (voir tri)

TRI D'UNE STRUCTURE COLLECTIVE

- Algorithmes génériques :
  - ✓ **Collections.sort(List l)**
  - ✓ **Arrays.sort(Object[] a,…)**
- Condition : collection d'éléments dont la classe définit des règles de comparaison :
  - ✓ en implémentant l'interface **java.lang.Comparable**
  - ✓ **implements Comparable**
- en définissant la méthode de comparaison :
  - ✓ **public int compareTo(Object o)**
  - ✓ **a.compareTo(b) == 0** si **a.equals(b)**
  - ✓ **a.compareTo(b) < 0** pour a strictement inférieur à b
  - ✓ **a.compareTo(b) > 0** pour a strictement supérieur à b

GÉNÉRICITÉ DES ALGORITHMES

- Il est important de n'utiliser que les méthodes communes
- Déclaration :
  - ✓ **Collection maColllection = new ArrayList();**
- Parcours des éléments de la collection par un objet **Iterator** :
  - ✓ Parce que l'accès indexé n'est pas toujours disponible (méthode **get()**)
  - ✓ On utilise alors la méthode **Iterator iterator()**
  - ✓ On se déplace avec les méthodes **next()** et **hasNext()**
  - ✓ Exemple :

```
Collection col = new TreeSet();
Iterator i = col.iterator();
while (i.hasNext())
        traiter ((transtypage)i.next());
```