CNAM – Aix en Provence

Programmation Avancée NFP 121

Généricité et Collection

Chap. 4

Erwan TRANVOUEZ
Maître de Conférences

erwan.tranvouez@polytech.univ-mrs.fr
http://erwan.tranvouez.free.fr

Objectifs de la session :

- Comprendre l'intérêt de la généricité et voir en quoi elle s'appuie sur les concepts Orienté Objet
- Appliquer ces principes sur la gestion d'objets…

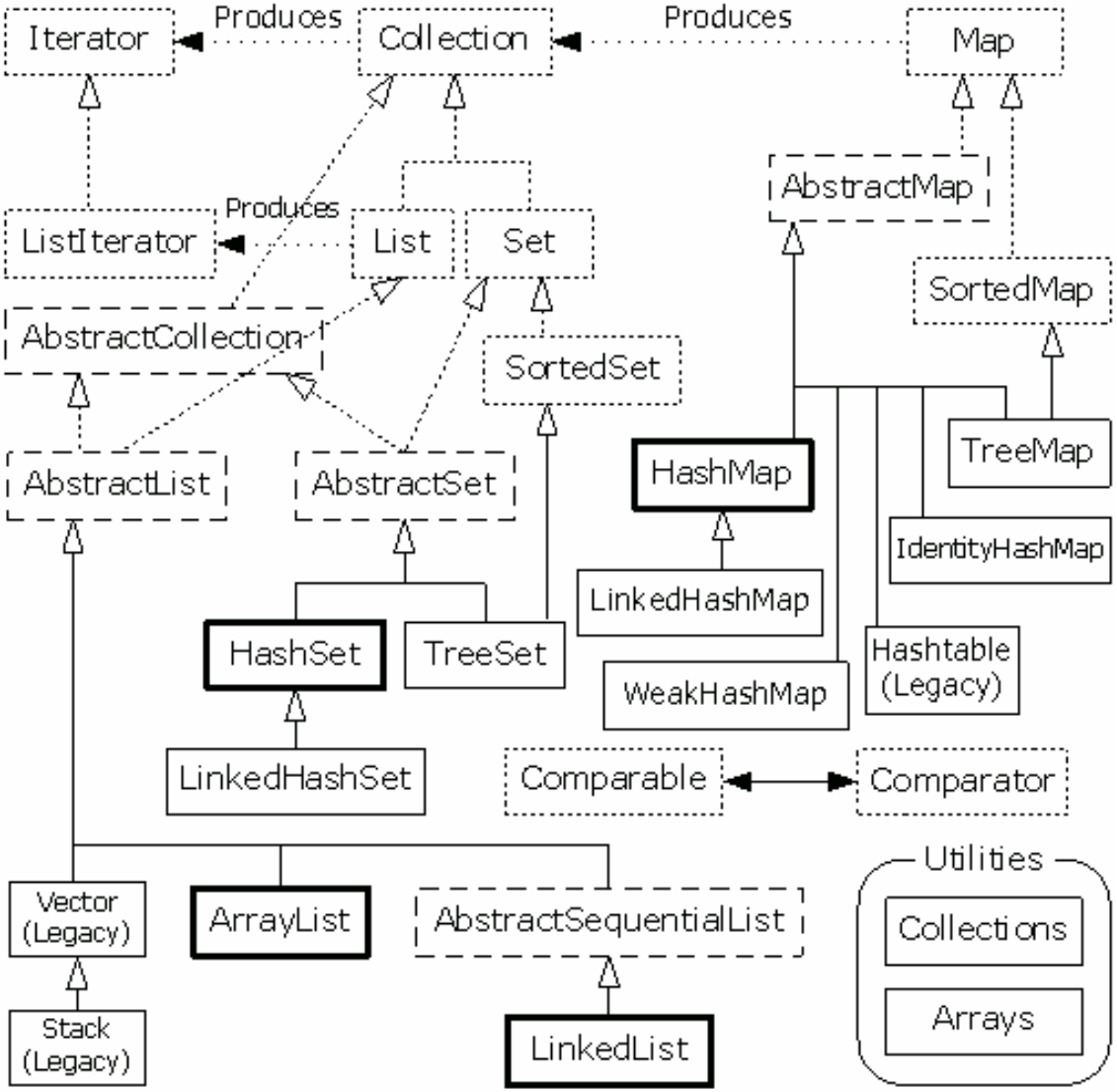
- Biblio centrée Collection:
  - **Tutorial Java**, http://java.sun.com/docs/books/tutorial/collections/

1. Généricité et Collection

Origines

- Gérer des ensemble d'objets autrement que par des tableaux … en évitant au programmeur de développer lui-même des classes utilitaires…
- Les premières collections font leur apparition avec java :
  - array (tableau), Vector, Hashtable
- Avec la 1.2 ajout de la libraire Java Foundation Classes (pendant des Mxxxsoft FC) qui contiennent (entre autres) des collections un peu plus sophistiquées : les « Collections framework » (~infrastructure de stockage). Elle propose :
  - Des ***interfaces*** : devant être implémentée par les classes pour qu'elles puissent être manipulées par les collections
  - Des ***implémentations*** : ie classes utilitaires implémentant une stratégie particulière de stockage
  - Des ***algorithmes*** … génériques implémentés au sein de classes en se basant largement sur le polymorphisme…

Vision générale…



Legacy :
Anciennes classes toujours supportées pour assurer la compatibilité des codes mais bien qu'ils conservent les mêmes interfaces (ie méthodes appelables) l'implémentation peut varier (ie être en fait une encapsulation d'une autre classe)

Avantage

- Programmation simplifiée :
  - Sélection de la bonne collection en fonction du besoin
  - Utilisation / éventuellement adaptation..
- Recours à des librairies éprouvées
- Facteur de stabilité du code produit…
- …

Retour sur les tableaux

- Généricité limitée :
  - Object [] tab = new Object [10];
    - Peut contenir 10 instances d'Object … donc de n'importe quel type
  - MaClasse [] tab = new MaClasse[10];
    - Ne peut contenir que des instances de MaClasse ou de classes héritant de MaClasse
- Taille fixe :
  - Redimensionner le tableau revient à
    - Créer un nouveau (plus grand) tableau
    - Copier les valeurs de l'un à l'autre (étape simplifiée grâce à System. arraycopy())
    - => Ipas de realloc comme en C/C++
- Concept OO d'encapsulation non respectée :
  - On accède directement (sans contrôle) à une valeur (d'où le risque de lever une exception).
  - Ex: *tab[i]=0;*

1er effort de généricité : java.util.Vector

- Propose une implémentation d'un tableau dynamique basique :
  - Stockage d'éléments dans un tableau à 1 dimension
  - Ajout / Récupération / suppression d'éléments dans le tableau
  - Conversion dans d'autres format (tableau, Enumeration, Iterator…)
  - Se redimensionne automatiquement dés que le tableau n'est plus suffisant…

- Au départ: version non optimisée et potentiellement dangereux à utiliser avec des Threads.
- `http://java.sun.com/j2se/1.4.2/docs/api/java/util/Vector.html`

java.util

Class Vector

java.lang.Object
  └ java.util.AbstractCollection
    └ java.util.AbstractList
      └ java.util.Vector

All Implemented Interfaces:
Cloneable, Collection, List, RandomAccess, Serializable

Ex d'utilisation de Vector

- Vector v = new Vector();
- v.add(monObjet);
- v.add(new String[2]);
- String s = (String) v.elementAt(12);
- if (! v.isEmpty() && v.contains(o)){
  v.remove (v.indexOf(o)) // eq. à v.remove(o);
  }

….

Exercices

- Exercice 1 :
  - Créer et manipuler un tableau de Compte (compte en banque)
- Exercice 2 :
  - Créer VectorCompte qui est un tableau dynamique qui ne doit contenir que des instances de Compte.
  - => redéfinir
    - boolean add(Object o) => add(Compte) + add(Object o) qui renvoie faux si !(o instanceof Compte)
    - void setElementAt(Object obj, int index)
    - Object elementAt(int index) => renvoie un Compte
  - **Quel problème rencontre t'on ?**

Explication

- Ces explications ne sont valables que pour une JVM ne dépassant pas la 1.4.
  - Il n'est pas possible de redéfinir une méthode en ne changeant QUE son type de retour.
  - En effet cela rendrait impossible en cours d'exécution la levée de l'ambiguïté d'un appel comme celui-ci :
    - System.out.println(elementAt(2));
    - => impossible de savoir si l'on veut le elementAt qui renvoie une instance d'Object ou de Compte
  - Solution (pas très élégante)
    - Ajouter une méthode compteAt() ou elementCompteAt()
    - Redéfinir elementAt qui peut soit renvoyer null systématiquement ou lever une RuntimeException (car on n'était pas censé utiliser cette fonction.
    - Solution qui tiens plus du pis aller puisqu'il s'agit en fait de casser chez la classe fille le « contrat » qui la liait à la mère : ie la fille s'engage à traiter les messages qui sont hérités de la mère.
    - C'est un peu comme si dans une classe on redéfinissait toString() pour renvoyer null.
- Pour les JVM plus récentes, cet exemple ne marche plus du fait de typage dynamique… (cf. ci après).

La gestion de Liste avec Iterator

- Version évoluée de l'interface Enumeration
  - Rappel :
    - Enumeration e = v.elements() ;
      while(e.hasMoreElements()) {
      System.out.println(e.nextElement());
      }
  - Propose un mécanisme de parcours sous forme de pile
- L'interface iterator propose en plus une méthode de suppression (remove())

- `http://java.sun.com/j2se/1.4.2/docs/api/java/util/Iterator.html`

Autre stratégie de stockage : les tables de hachage…

- Comment ranger des éléments dans un tableau en utilisant un index non numérique :
  - i ème élément du tableau tab : tab[i]
  - Élément dont le nom est ‘toto’ : tab[‘toto’]
- Non géré au niveau syntaxe Java (pas de type spécifique), mais utilisation de classes dédiées, parmi lesquelles la première Hashtable.
- Intérêt:
  - Avant tout accélérer le temps de recherche d’un élément dans un tableau (ie évite le parcours de tous les éléments)
  - Cacher l’aspect indice qui est en fait de l’ordre de la cuisine interne de stockage
- => suppose d’être capable d’associer une clef (pas forcément unique) à un indice (ex. ‘toto’ -> 10)

Problème de la fonction de hachage

- Problème: détermination de la fonction de « hachage » qui associe à une clef un indice …
  - Idéalement la fonction est bijective, ie a 1 clef correspond 1 seul indice et réciproquement.
  - Si on a f(k) = f(k') => un même indice référencera 2 valeurs de clef possible… (on parle de collision)
- Solution :
  - Ajouter de manière contiguë k & k'.
  - A charge du programmeur de les tester ensuite…
- Ex:
  - f(s) s étant un String.
  - f(s) = nbCar('t',s)x10+nbCar('o',s)
  - Ou nbCar compte le nombre d'occurrences d'un caractère dans une chaîne
  - =>
    - f('toto') = 22 = f('otot')

…	
22	Toto
…	

…	
22	Toto
	Otot
…	

Ex. de collision

1ère Implémentation en java : java.util.Hashtable

- Propose une implémentation d'une table de hachage dynamique basique :
  - Stockage d'éléments Object dans un tableau à 1 dimension avec utilisation d'un objet (Object) comme clef.
  - Contraintes sur les objets clefs :
    - Implémenter hashcode() : ie la classe de l'objet propose une fonction de hachage adaptée (ex. String propose la méthode suivante : `s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]`)
    - Implémenter equals() : afin de vérifier que 2 clefs sont équivalentes
  - Plus fonctions classiques des tableaux dynamiques
    - Ajout / Récupération / suppression d'éléments dans le tableau
    - Conversion dans d'autres format (tableau, Enumeration, Iterator…)
    - Se redimensionne automatiquement dés que le tableau n'est plus suffisant…
- `http://java.sun.com/j2se/1.4.2/docs/api/java/util/Hashtable.html`

Exemple

- ```
  HashTable tableHash  = new HashTable();
  tableHash.put("Paul PERSONNE", p1);
  tableHash.put("Pierre NOWAN", p2);
  ```
  Avec p1 et p2 tous deux des instance de Personne.

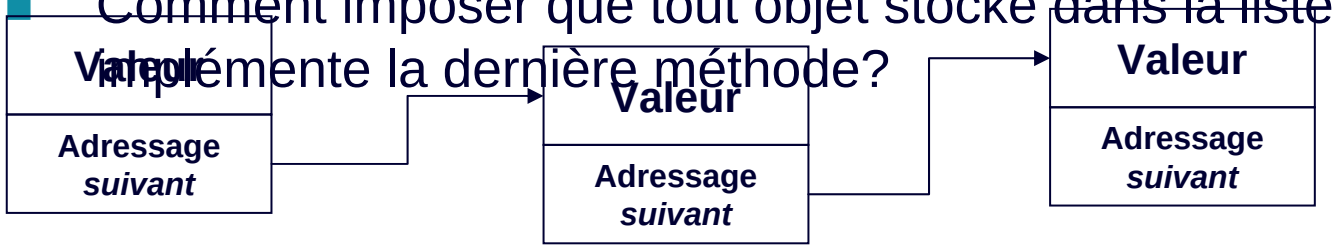
  Pour retrouver le "dossier " d'une personne il suffit de construire sa clé à partir de son nom & prénom et de le chercher :

  ```
  String clef = prenom+ " " + nom.toUpperCase();
  Personne p = (Personne) tableHash.get(clef);
  ```

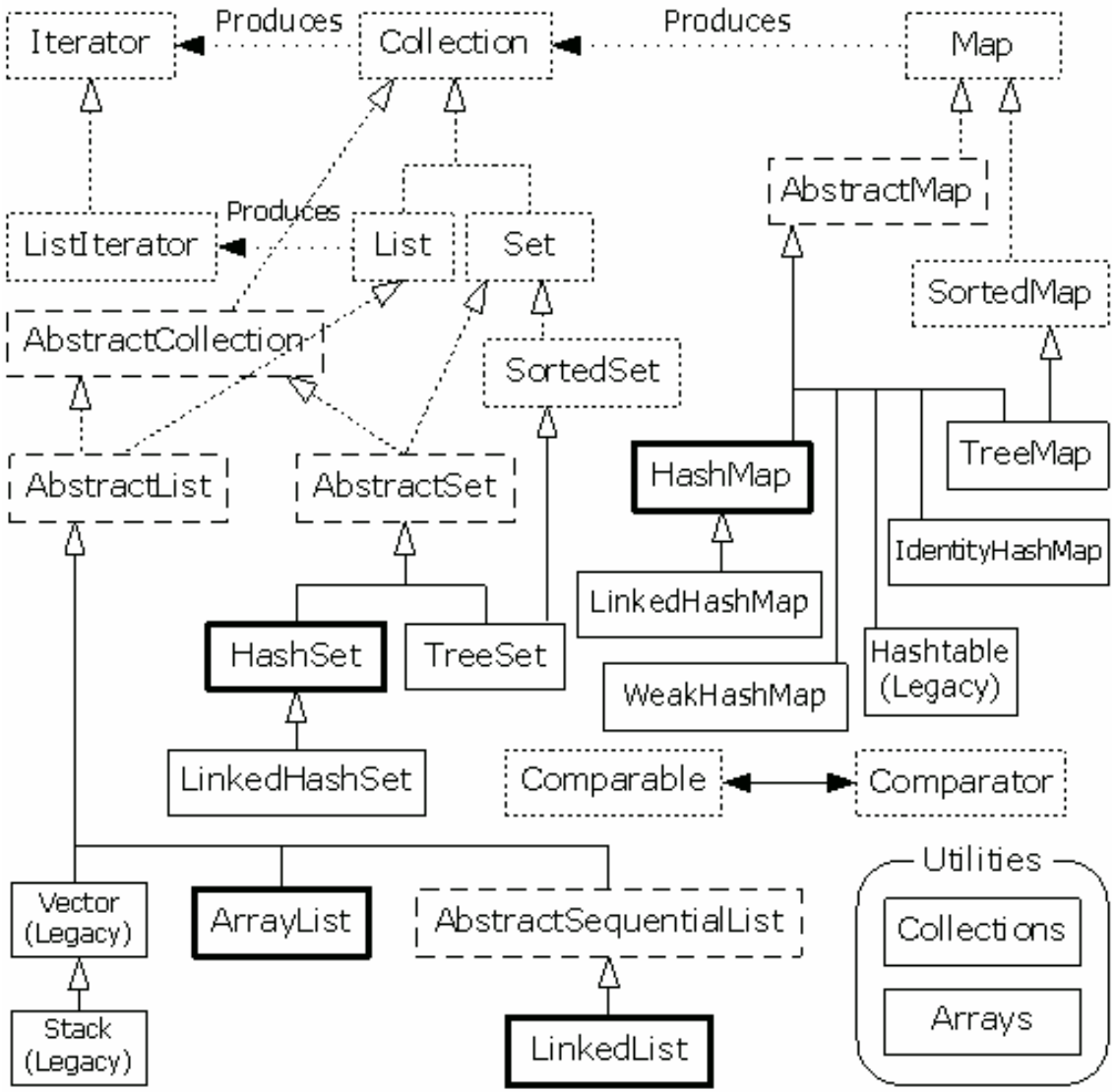
- Cela suppose une vérification de la clé utilisée …
- Méthodes de gestion des clefs utilisées:
  - Keys() : renvoie une énumération des clefs
  - containsKey(o) : true/false selon que o est déjà utilisé comme clef ou non (ie pour la gestion de collision)
  ```

Exercice : implémentation d'une liste chaînée… : ListeChainee

- Le principe est relativement simple :
  - La classe possède
    - un attribut valeur (Object) correspondant à l'objet à stocker.
    - un attribut suivant (ListeChainee) qui « pointe » sur l'élément suivant de la liste (null si cet élément est le dernier).
    - Un attribut tête qui pointe sur le 1er élément de la liste.
    - Des méthodes d'ajout : ajouterDebut(Object valeur) , ajouterFin(Object valeur)
    - Des méthodes de suppression : supprimerDebut(), supprimerFin(), supprimerElement(Object o).
    - Une méthode de recherche sur la base d'une clef : trouverValeur(Objet key)
  - Comparez votre résultat avec la classe utilitaire LinkedList.
  - Comment imposer que tout objet stocké dans la liste chaînée implémente la dernière méthode?

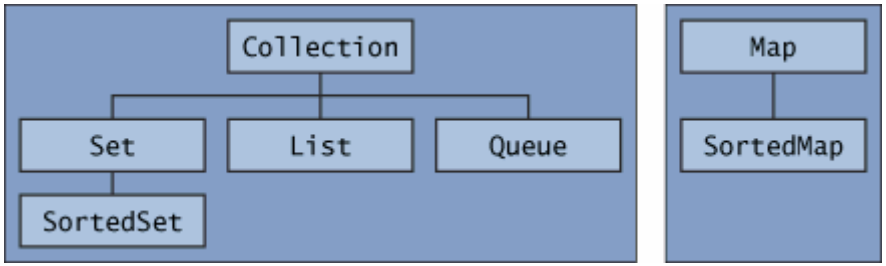


Retour sur la hiérarchie



Réorganisation des librairies

- L'évolution vers un cadre de gestion des ensembles (framework) à conduit à réifier certains concepts :

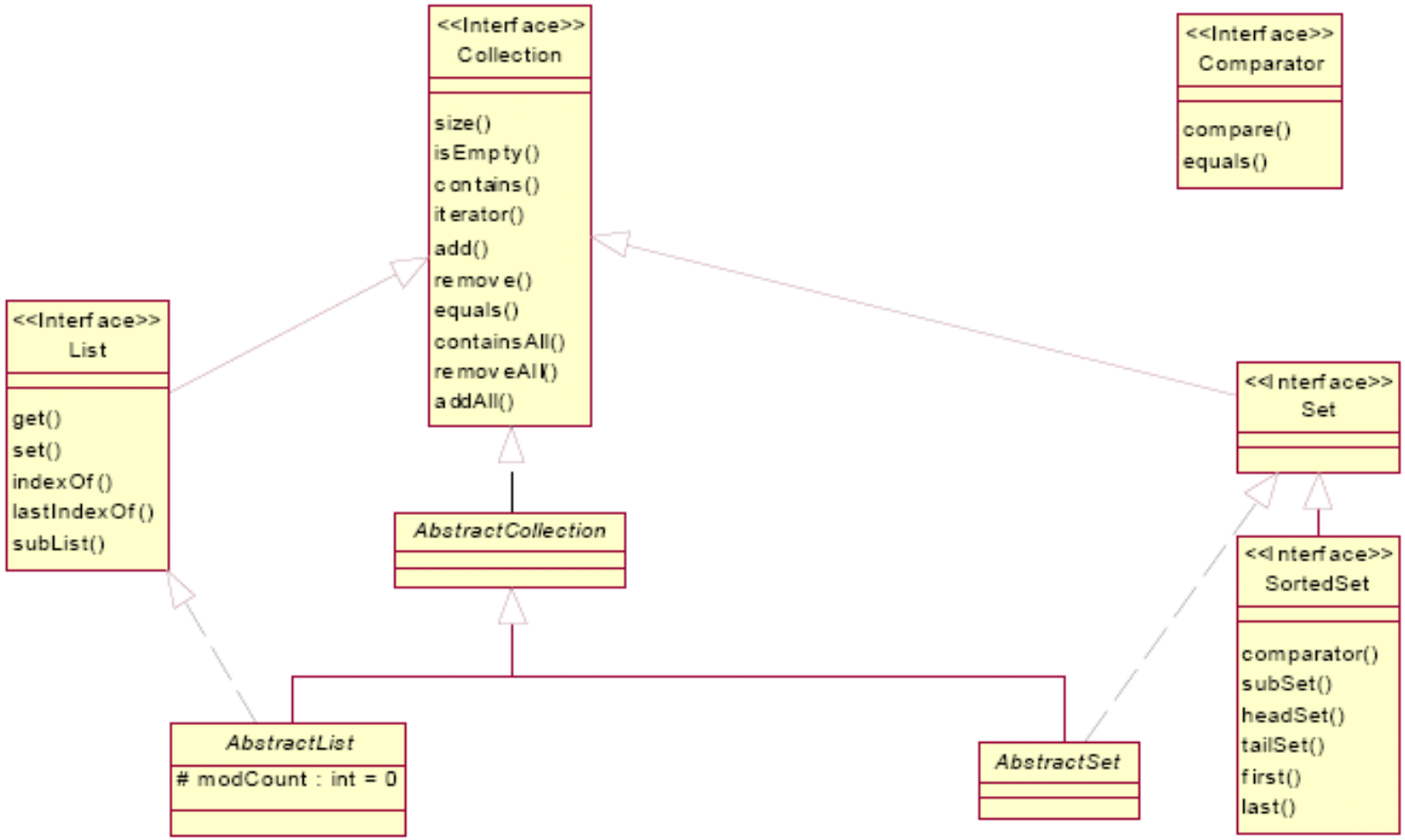


- Collection : ensemble d'éléments
- Set : ensemble sans doublon
- List : ensemble ordonné d'élément
- Queue : gestion de liste au sens d'attente (pile type FIFO, FILO etc…)

  - SortedSet : ensemble ordonné de manière croissante
  - SortedMap : ensemble type dictionnaire (utilisé par ex. comme clé pour un HashTable).

Source : tutorial Java

Vue générale



Source : tutorial Java

Interface Comparable

- Se limite à proposer 2 méthodes :
  - compare()
  - equals()

- Garantit à toute classe utilitaire qu'une classe implémentant l'interface Comparable est capable d'être triée.
- Ex: http://java.sun.com/javase/6/docs/api/java/util/Collections.html

  - Propose une implémentation d'une série de fonctions utilitaires se basant notamment sur le fait qu'un classe implémente l'interface Comparable
    - static int **frequency**(Collection c, Object o) : Compte le nb de fois que o apparaît dans la Collection c : c peut donc être une ArrayList, une HashList etc…
    - public static Object **max**(Collection coll) : no comment
    - public static Object **max**(Collection coll, Comparator comp) cf. ci après
    - Static void sort(List list) : trie une liste en se basant sur l'algorithme 'merge sort' / tri par fusion.

Délégation de comparaison

- L'implémentation de l'interface Comparable suffit à garantir la capacité à ordonner 2 instances d'une même classe.
- Problèmes :
  - la méthode equals() étant « codée » dans la classe, il n'est plus possible de la modifier qu'en héritant de la classe et en redéfinissant la méthode equals…
  - Un seul critère de comparaison est possible et « institutionnalisé » au travers de l'implémentation de la méthode equals().
    - Ex: tri en fonction de critères différents (age, taille, etc…)
- Solution :
  - Externaliser la comparaison : utiliser un Comparator

L’interface Comparator

- Propose 2 méthodes
  - compare(Object o1, Object o2)
  - equals(Object o1) // méthode « interne » aux Comparators
- Principe :
  - On conçoit autant de classes implémentant l’interface Comparator que de critère de comparaison
  - Ex:
    - `ComparaisonChaineParLongueur implements Comparator`
    - `ComparaisonChaineParNbDeVoyelle implements Comparator`
  - Ces classes ne sont destinées à être instanciée qu’une seule fois.
  - Lorsqu’un traitement requiert une comparaison il délèguera la comparaison à l’instance Comparator adéquate.
  - Possibilité de modifier le critère de tri, sans modifier la classe dont on veut comparer les instances.

L'interface Comparator

- Ex d'implémentation :
- ```
  public class ComparaisonChaineParLongueur implements Comparator{
      compare(String s1, String s2) {
          return s1.length() – s2.length();
      }
      /* … */}
  ```
- Utilisation du comparator :
  - ```
    public static String max(String s1, String s2,
    Comparator c) {
        if( c.compare(s1, s2) > 0)
            return s1;
        else
            return s2;
    }
    ```
  - La fonction de tri `Collections.max(Collection coll, Comparator comp)` se base sur ce principe…

Généricité

- Limitations :
  - Stocke des Objets => toute manipulation requiert des casts…
  - Pas de vérification d'homogénéité de type par défaut dans une collection (=> il faut les étendre et le développer soit même)
- Solution :
  - Ajout d'un mécanisme de typage automatique
  - => Renommage des classes Collection<A>, Iterator<A> …
- Avantage :
  - Souplesse de programmation
  - Généricité accrue
- Inconvénient :
  - « Casse » la syntaxe vue jusqu'ici
  - Obscurcis le code et ne protège pas de ClassCastException …

Exemple

```
public interface List <E>{
    void add(E x);
    Iterator<E> iterator();
}
```

E n'est pas connu à l'avance…

Utilisation :

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```