PROGRAMMATION PAR OBJETS

package java.util

utilitaires de Arrays

structures de données génériques

Utilitaires et collections Java

- Tableaux
  - intégrés dans le langage : «Objets» de taille fixe
  - type des éléments : primitif (homogène) ou objets (hétérogène)

- Bibliothèque (package) java.util
  - Arrays : utilitaires de manipulation de tableaux (égalité, tri, recherches)
  - Structures de données dynamiques (« collections »)
    - Set : ensembles
    - List : listes
    - Map : tables d'association
  - génériques depuis Java 5.0

Utilitaires sur les tableaux

- Fonctionnalités offertes par la classe java.util.Arrays
  - sous la forme de fonctions static
  - définies pour des tableaux d'objets, mais surchargées pour tous les types primitifs

- Egalité de contenu

```
boolean Arrays.equals(t1,t2) <=>
(t1.length == t2.length)
et (t1[i].equals(t2[i]),∀ i<t1.length)
```

- Rappel : la méthode equals de Object teste par défaut l'égalité physique (même instance au sens de ==).
- il est possible de la redéfinir pour tester l'égalité logique d'objets.
- Exemple : égalité de contenu entre chaînes définie dans String

Egalité de tableaux : exemple

```
public class Ouvrage {
  protected String titre, auteur;
  public Ouvrage(String tit, String aut) {
    titre=tit; auteur=aut;
  }
  public String toString() {
   return titre+" "+auteur;
  }
  public boolean equals(Object o) {
    return ( (o instanceof Ouvrage)
        && (auteur.equals(((Ouvrage)o).getAuteur())) //downcast
        && (titre.equals(((Ouvrage)o).getTitre())) );
  }
...}
```

Egalité de tableaux : exemple

```
Ouvrage t1[] = {
  new Ouvrage("Germinal","Zola"),
  new Ouvrage("C","Kernighan"),
  new Ouvrage("Java","Eckel") };
Ouvrage t2[] = {
  new Ouvrage("Germinal","Zola"),
  new Ouvrage("C","Kernighan"),
  new Ouvrage("Java","Eckel") };
```

- `t1==t2` est `false`
  - `<=>` `t1.equals(t2)` (égalité physique par défaut)
  - `tout comme : t1[i]==t2[i]`
- `Arrays.equals(t1,t2)` est `true` (égalité logique)
  - `tout comme t1[i].equals(t2[i])` (itérée sur les éléments)

Tri

```
void Arrays.sort(Object t[])
```

- Les éléments du tableau doivent offrir une relation d'ordre en implémentant la méthode compareTo spécifiée par l'interface Comparable <T> :

```
public interface java.lang.Comparable<T> {
  int compareTo(T obj);
  // < 0 si this < obj
  // = 0 si this = obj
  // > 0 si this > obj
}
```

Tri : exemple

```
// sachant :
class String implements Comparable<String>

// ordre sur les ouvrages <=> ordre sur leur auteur :
public class Ouvrage implements Comparable<Ouvrage> {
  public int compareTo(Ouvrage obj) {
    return
     auteur.compareTo(obj.getAuteur());
  }...}

//application
Arrays.sort(t1);

// resultat
Java Eckel
C Kernighan
Germinal Zola
```

Pour ordonner des éléments dans une collection, il faut que la classe de ces éléments implémentent l'interface `Comparable<E>`

Ordres multiples

- La technique du compareTo par l'interface **Comparable** est dite «de l'ordre naturel» (méthode interne de l'objet).
- Elle pose problème si:
  - on veut trier des objets dont la classe n'implémente pas Comparable
  - ou si l'ordre naturel ne convient pas
  - ou encore si l'on désire ordonner les objets selon plusieurs critères.
- Une autre technique est offerte en définissant l'ordre souhaité, non plus dans la classe des éléments, mais
  - lors de la construction de la collection
  - ou lors de l'appel d'un algorithme.
- Pour ce faire, on fournit en argument (du constructeur ou de l'algorithme un «objet comparateur» qui doit implémenter l'interface **Comparator**<T>:

```
interface java.util.Comparator<T> {
   public int compare(T o1, T o2);
  // < 0 si o1 < o2
  // = 0 si o1 = o2
  // > 0 si o1 > o2
  }
```

Ordres multiples

- Le tri de tableaux s'effectue alors en fournissant l'objet comparateur en paramètre :

```
<T> void Arrays.sort(T[] t, Comparator<T> c)
```

- Exemple

```
// nouvel ordre des ouvrages sur leur titre :
public class ComparateurDeTitres
        implements Comparator<Ouvrage>{
 public int compare(Ouvrage o1, Ouvrage o2) {
  return o1.getTitre().compareTo(o2.getTitre());
 }}

//application
Arrays.sort(t2, new ComparateurDeTitres());
```

Recherches ordonnées

- recherche d'un objet dans un tableau ordonné par ordre naturel

```
int Arrays.binarySearch(Object t[], Object obj)
```

- recherche d'un objet dans un tableau ordonné par comparateur

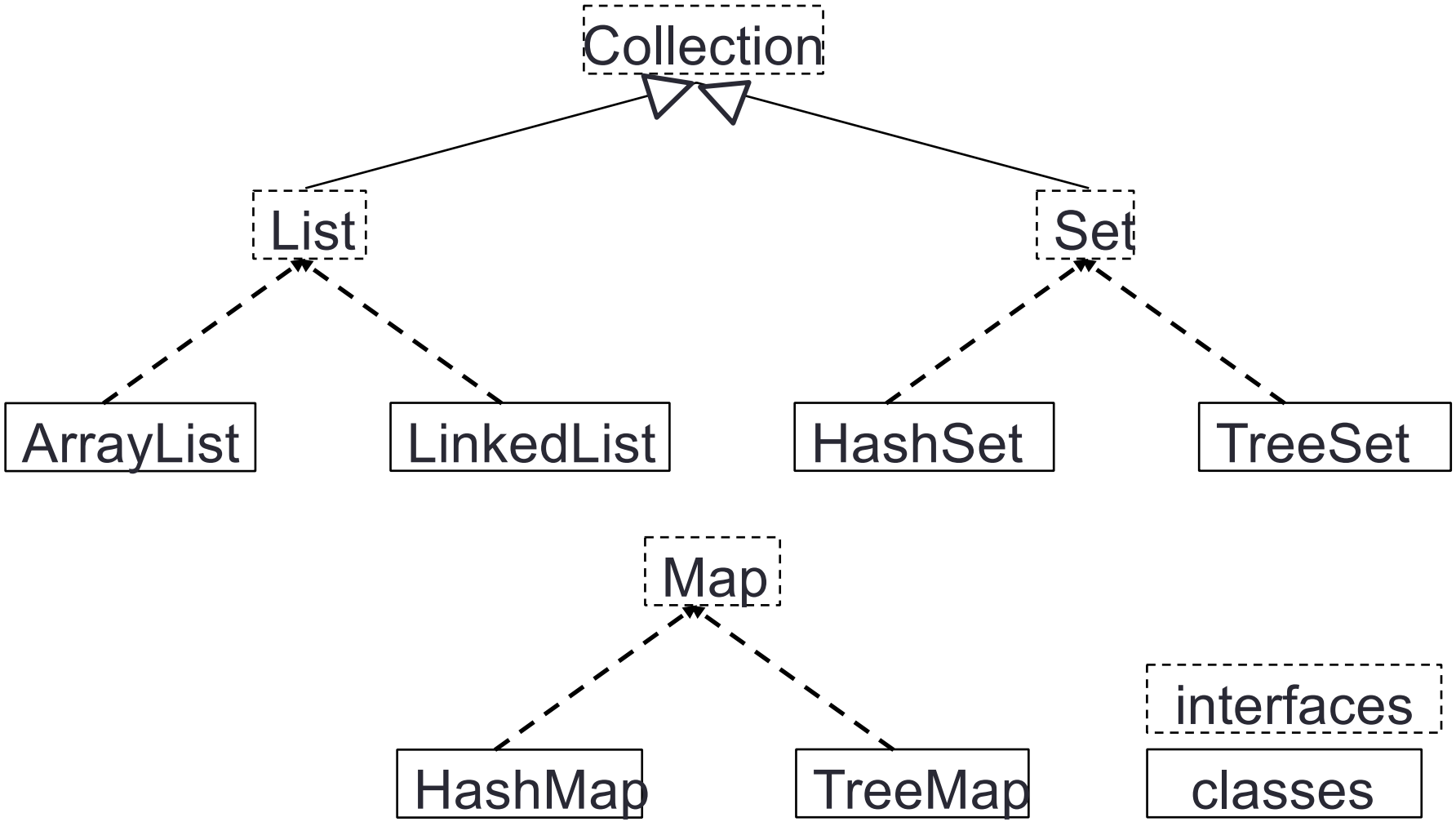
```
<T> int Arrays.binarySearch(T t[], T obj, Comparator<T> c)
```

- Ces méthodes renvoient :
  - l'indice de l'élément s'il est trouvé
  - (- <indice d'insertion> - 1) sinon, avec:
    - <indice d'insertion> = 0 si obj < t[0], t.length si obj > t[t.length-1]
    - t[<indice d'insertion> - 1] < obj < t[<indice d'insertion>], sinon
- La recherche ordonnée sur un tableau non trié (selon l'ordre correspondant) est imprédictible!

```
Arrays.binarySearch(t1,new Ouvrage("Germinal","Zola"))=2
Arrays.binarySearch(t1,new Ouvrage("Parapente","AliGali"))=-1
Arrays.binarySearch(t1,new Ouvrage("IA","Nilsson"))=-3

Arrays.binarySearch(t2,new Ouvrage("Germinal","Zola"),
  new ComparateurDeTitres()))=1
```

Bibliothèque de structures de données



Collection

- Les collections sont des regroupements dynamiques d'objets.
- Les formes les plus courantes sont les listes (doublons d'éléments possibles) et les ensembles (sans doublon).
- Les collections contiennent des objets
  - pour gérer des types primitifs utiliser les classes wrapper
- Avant 5.0 : le type des éléments est indifféremment Object
  - pas de contrôle statique de type
  - à contrôler dynamiquement « à la main » par casts
- Depuis 5.0 : les collections sont génériques :
  - paramétrées par le type des éléments
  - contrôle de type correspondant sur les opérations de manipulation.

Collection

- L'interface Collection<E> spécifie les fonctionnalités abstraites communes aux classes de collections :

```
public interface Collection<E>
 public boolean add(E o)
 public boolean remove(Object o)
 public boolean contains(Object o) //par equals()
 public int size()
 public void clear()
 public boolean isEmpty()
 public Object[] toArray() //inverse de Arrays.asList(t)
 public boolean equals(Object o)
```

- Il existe des versions itérées de add, contains, remove : suffixées par All

```
boolean addAll(Collection<? extends E> c)
boolean containsAll(Collection<?> c)
<=> c inclus dans this
```

Attention, il est impossible d'ajouter un nouvel élément dans la liste obtenue par `Arrays.asList(t)`.

Différentes collections

Le framework Collections propose plusieurs implémentations possédant chacune un comportement et des fonctionnalités particulières.

Collection	Ordonné	Accès direct	Clé / valeur	Doublons	Null
ArrayList	Oui	Oui	Non	Oui	Oui
LinkedList	Oui	Non	Non	Oui	Oui
HashSet	Non	Non	Non	Non	Oui
TreeSet	Oui	Non	Non	Non	Non
HashMap	Non	Oui	Oui	Non*	Oui
TreeMap	Oui	Oui	Oui	Non*	Non

* Non sur les clés et donc les associations, mais oui sur les valeurs

Itération sur les éléments

- Les collections offrent une interface d'itération sur leurs éléments

```
public interface Collection<E> extends Iterable<E> {
 public Iterator<E> iterator();
}
```

- Interface d'itération

```
public interface Iterator<E> {
 public boolean hasNext()
 public E next() throws NoSuchElementException
}
```

- Exemple de la classe `Circuit`

```
public void run() {
 Iterator<Porte> iter = composants.iterator();
 while (iter.hasNext()) iter.next().run();}
```

- et même en 5.0, comme tout Iterable (tableaux, collections, ...):

```
for(Porte p : composants) p.run();
```

`p` prend successivement la valeur de chacune des références des éléments de la collection. Pas exploitable si l'on doit modifier la collection.

Collections d'éléments de type primitif

- Pas possible d'ajouter un élément de type primitif dans une collection
- type primitif : utiliser les classes wrappers

```
ArrayList<Double> l = new ArrayList<Double>();
```

- en 5.0 l'"auto-boxing/unboxing" n'oblige pas à wrapper/dewrapper

```
double x;
l.add(new Double(x)); // wrapping ou depuis 5.0 plus simplement:
l.add(x); //"autoboxing » (automatique)

// restitution : somme des elements
double s=0.0;
for(int i=0;i<l.size();i++)
 // s=s+l.get(i).doubleValue(); // unwrapping,  ou depuis 5.0 :
 s=s+l.get(i); // "auto-unboxing" (automatique)

// par iteration "for each"
for(double x : l) s=s+x;
```

Les listes : java.util.List

- Les listes sont des collections d'objets (avec doublons possibles) ordonnées de manière externe par indice de rangement.

- Elles sont issues d'une même interface `List<E>` qui ajoute à `Collection` les opérations d'accès direct indicé:
  `get(i), add(i,x), set(i,x), …`

- Deux classes de listes :
  - listes chainées: `LinkedList<E>`
    plus performantes sur les opérations de mise à jour (ajout/retrait)
  - listes contigües: `ArrayList<E>`
    plus performantes sur les opérations d'accès indicé
    (voir aussi la classe `Vector` plus ancienne)

`Vector` n'est conservée que par compatiblité ascendante et elle ne devrait pas être utilisée dans les nouveaux programmes.

Les listes : java.util.List

```
public interface List<E> extends Collection<E>
  public void add(int index, E element)
   // sachant que add ajoute en queue
  public E get(int index)
   throws IndexOutOfBoundsException
  public E set(int index, E element)
   throws IndexOutOfBoundsException
  public E remove(int index)
   throws IndexOutOfBoundsException
  public int indexOf(Object o)
    throws ClassCastException
    // indice de la 1ere occurrence
    // -1 si !this.contains(o)
  public List<E> sublist(int fromIndex, int toIndex)
    throws IndexOutOfBoundsException
```

Exemple

```
import java.util.*;
public class Circuit {
 protected List<Porte>
     composants = new ArrayList<Porte>();
 public void brancher(Porte p) {
   composants.add(p);
 }
  public void remplacer(int i, Porte p) {
  composants.set(i,p);
 }
  public void run() {
  for (int i=0;i<composants.size();i++) {
     composants.get(i).run();
 }
}
```

ArrayList vs LinkedList

- Pas de spécificité à Java
- Si l'ajout ou la suppression d'éléments se font essentiellement :
  - à la fin de la collection, alors il faut utiliser la classe ArrayList
  - à une position aléatoire dans la collection, alors il faut utiliser la classe LinkedList
- Un élément peut être accédé directement par son index dans une ArrayList, ce qui n'est pas possible avec une LinkedList sauf pour le premier et le dernier élément.

	get	add	contains	next	remove(0)
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)

Les ensembles : java.util.Set

- Tout comme les List, les Set implémentent l'interface Collection mais interdisent les doublons au sens de equals:
  `∀o1,o2, !o1.equals(o2)`
  - Cette propriété est vérifiée lors des ajouts
  - mais elle n'est pas maintenue, la modification d'objets peut la rendre caduque : deux éléments peuvent devenir égaux («aliasing »)

- Contrairement aux List, l'ordre des éléments est interne et donc **pas d'accès direct** indicé.

- Il existe principalement deux sortes de Set :
  - TreeSet
  - HashSet

HashSet / TreeSet

- HashSet<E>
  - Les éléments sont rangés de façon hachée par appel de leur méthode `hashCode()` (de `Object`, redéfinissable).
  - Les performances de `HashSet` (en *O*(1)) sont meilleures que `TreeSet` pour la recherche (et donc aussi pour les autres actions).
  - Ensembles non ordonnés
- TreeSet<E>
  - Ou ensembles ordonnés, les éléments sont rangés dans un arbre binaire ordonné, ce qui assure des manipulations en $\log_2(n)$.
  - L'ordre des éléments est :
    - soit leur ordre naturel (`compareTo` de `Comparable`) si aucun `Comparator` n'est fourni au constructeur. La classe `E` des éléments doit alors implémenter `Comparable<E>`.
    - soit déterminé par un `Comparator` d'éléments fourni au constructeur :
      `public TreeSet<E>(Comparator<E> c)`

Choix d'une implémentation de type Set

- Le choix se fait essentiellement sur :
  - La volonté / nécessité d'avoir un ordre.
  - Les performances suivant l'ajout, la recherche, ou le passage à l'élément suivant.

Classe	add()	contains()	next()
HashSet	O(1)	O(1)	O(h/n)
TreeSet	O(log n)	O(log n)	O(log n)

Set : exemple

- Un ensemble d'ouvrages ordonné selon leur ordre naturel (par auteur) construit sur le tableau t1 (cf. chapitre Arrays)

```
Set<Ouvrage> ensemble
   = new TreeSet<Ouvrage>(Arrays.asList(t1));
ensemble.add(new Ouvrage("Parapente","Ali Gali"));
for(Ouvrage o : ensemble) System.out.println(o);

// resultats...
Parapente Ali Gali
Java Eckel
C Kernighan
Germinal Zola
```

Comme toute collection, un ensemble peut être construit vide ou à partir d'une autre collection.

Les tables d'association : java.util.Map

- Elles permettent de maintenir des associations clé-valeur <K,V>:
  - chaque clé est unique
  - les clés et valeurs sont des objets (wrapper si types de bases)
- L'interface abstraite `Map` spécifie les opérations communes aux classes de tables d'association :

```
public interface Map<K,V> {
 V put(K key, V value);
 V get(Object key); //!this.containsKey(key)=>null
 boolean containsValue(Object value);
 boolean containsKey(Object key);
 V remove(Object key);
Set<K> keySet() //l'ensemble des cles
Collection<V> values() // la liste des valeurs
...}
```

HashMap / TreeMap

- Il existe principalement deux sortes de tables : les **`HashMap`** et les **`TreeMap`** qui implémentent l'interface `Map`.
- `HashMap<K,V>`
  - Tables de hachage
  - Elles utilisent la méthode `hashCode()` des objets clés.
  - l'ensemble des clés est un `HashSet`.
  - Les performances de HashMap sont meilleures que `TreeMap` mais pas d'ordre sur les clés

- `TreeMap<K,V>`
  - permet de gérer des tables ordonnées sur les clés.
  - l'ensemble des clés est un `TreeSet` (arbre binaire ordonné assurant un accès en $\log_2(n)$).
  - les clés doivent donc être ordonnables
    - soit en implémentant `Comparable` (ordre naturel)
    - soit en fournissant un `Comparator` au constructeur :

```
public TreeMap<K,V>(Comparator<K> c)
```

HashMap / TreeMap

- Le critère de choix est essentiellement l'ordre de tri des éléments que la collection doit utiliser :
  - HashMap : aucun ordre précis pour les éléments qui doivent avoir l'implémentation de leurs méthodes hashCode() et equals() correctement codées
  - TreeMap : l'ordre naturel des éléments ou l'ordre défini par l'instance de type Comparator associée à la collection

	get()	containsKey()	next()
HashMap	O(1)	O(1)	O(h/n)
TreeMap	O(log n)	O(log n)	O(log n)

TreeMap : exemple

- Bibliothèque
  - table code-Ouvrage ordonnée par les codes (String)
  - TreeMap<String,Ouvrage>

```
public class NonDisponibleException extends
  Exception {}
public class Ouvrage {
  protected String titre, auteur;
  protected boolean emprunte;
  protected int compteur; // nombre d'emprunts
  public int getCompteur() {return compteur;}
  public void emprunter() throws NonDisponibleException {
    if (emprunte) throw new NonDisponibleException();
    else { emprunte=true; compteur++;}
}
```

TreeMap : exemple

```
public class Bibliotheque {
 protected Map<String,Ouvrage> ouvrages
   = new TreeMap<String,Ouvrage>();
 public void add(String code, Ouvrage o) {
   ouvrages.put(code,o);
 }
 public int totalEmprunts() {// parcours des valeurs
  int total=0;
  Iterator<Ouvrage> iter = ouvrages.values().iterator();
  while (iter.hasNext())
   total=total+iter.next().getCompteur();
  return total;
 } //suite ...
```

TreeMap : exemple

```
//suite …
 public void listing() {//"for each » sur les cles
  for(String code : ouvrages.keySet())
   System.out.println(code+":"+ouvrages.get(code));

 public void emprunter(String code)
  throws OuvrageInconnuException, NonDisponibleException {
 try {
   ouvrages.get(code).emprunter();
 } catch (NullPointerException ex) {
       throw new OuvrageInconnuException();
 } // NonDisponibleException : propagée (cf. throws)
}
 public class OuvrageInconnuException extends Exception {}
```

TreeMap : exemple

```
// application (main)

Bibliotheque bib = new Bibliotheque();
bib.add(“I101”,new Ouvrage(“C”,”Kernighan”));
bib.add(“L202”,new Ouvrage(“Germinal”,”Zola”));
bib.add(“S303”,new Ouvrage(“Parapente”,”Ali Gali”));
bib.add(“I345”,new Ouvrage(“Java”,”Eckel”));
bib.listing();

/* resultat : ordre lexicographique des codes (String)
I101:C Kernighan
I345:Java Eckel
L202:Germinal Zola
S303:Parapente Ali Gali
*/
```

TreeMap : exemple

```
// application (suite)

String code; // ...

try {
 bib.emprunter(code);
} catch (OuvrageInconnuException ex) {
  System.out.println("ouvrage "+code+" inexistant");
} catch (NonDisponibleException ex) {
  System.out.println("ouvrage"+code+"non dispo");
}
```

La classe Collections

- La classe Collections propose plusieurs méthodes statiques pour effectuer des opérations sur des collections.

Méthode	Rôle
void copy(List, List)	Copier tous les éléments de la seconde liste dans la première
Object max(Collection)	Renvoyer le plus grand élément de la collection selon l'ordre naturel des éléments
Object max(Collection, Comparator)	Renvoyer le plus grand élément de la collection selon l'ordre précisé par l'objet Comparator
void reverse(List)	Inverser l'ordre de la liste fournie en paramètre
void shuffle(List)	Réordonner tous les éléments de la liste de façon aléatoire
void sort(List)	Trier la liste dans un ordre ascendant selon l'ordre naturel des éléments
void sort(List, Comparator)	Trier la liste dans un ordre ascendant selon l'ordre précisé par l'objet Comparator