

COLLECTIONS D'OBJETS

Collections génériques, Interfaces abstraites, package java.util

Walter Rudametkin

Maître de Conférences

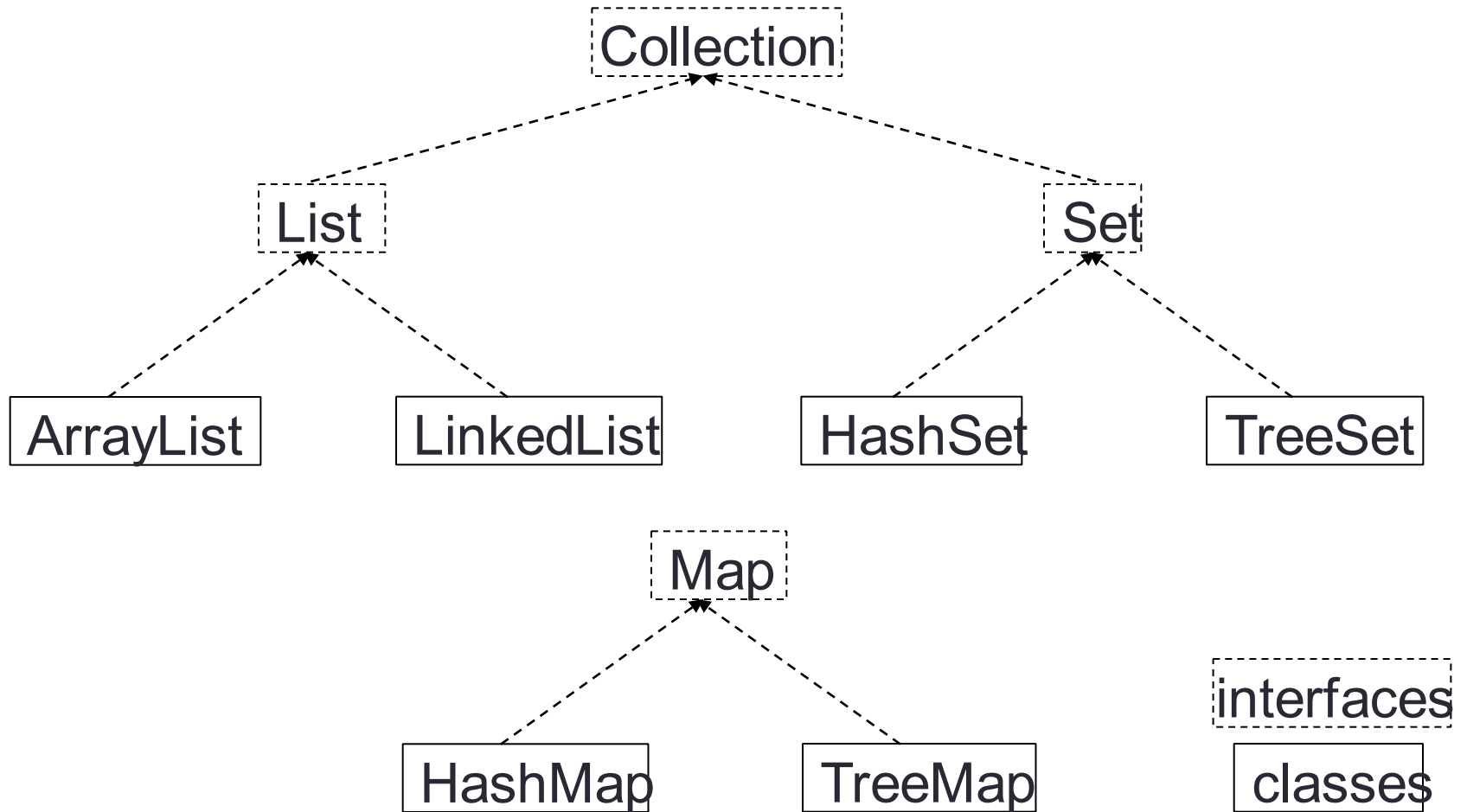
Bureau F011

Walter.Rudametkin@polytech-lille.fr

Collections d'objets

- Tableaux
 - intégrés dans le langage (avec syntaxe à la C)
 - un tableaux est un «objet» (allocation dynamique)
 - de taille fixe
 - type des éléments : primitif (homogène) ou **objets (polymorphe)**
- Bibliothèque (package) java.util
 - Structures de données dynamiques dont listes et tables d'association <clé-valeur>, ...
 - génériques depuis 5.0
- mais aussi :
 - **utilitaires algorithmiques**: sort, binarySearch, ...
 - Date, Calendar, ...

Classification des collections



Interfaces abstraites

- A l'extrême des classes abstraites
 - pas de structure (pas de variables d'instance, static possibles)
 - que des méthodes abstraites (« pure protocole »)
- Elles sont implémentées par les classes
 - relation **implements** (n-aire)
 - en plus de **extends** (unaire)

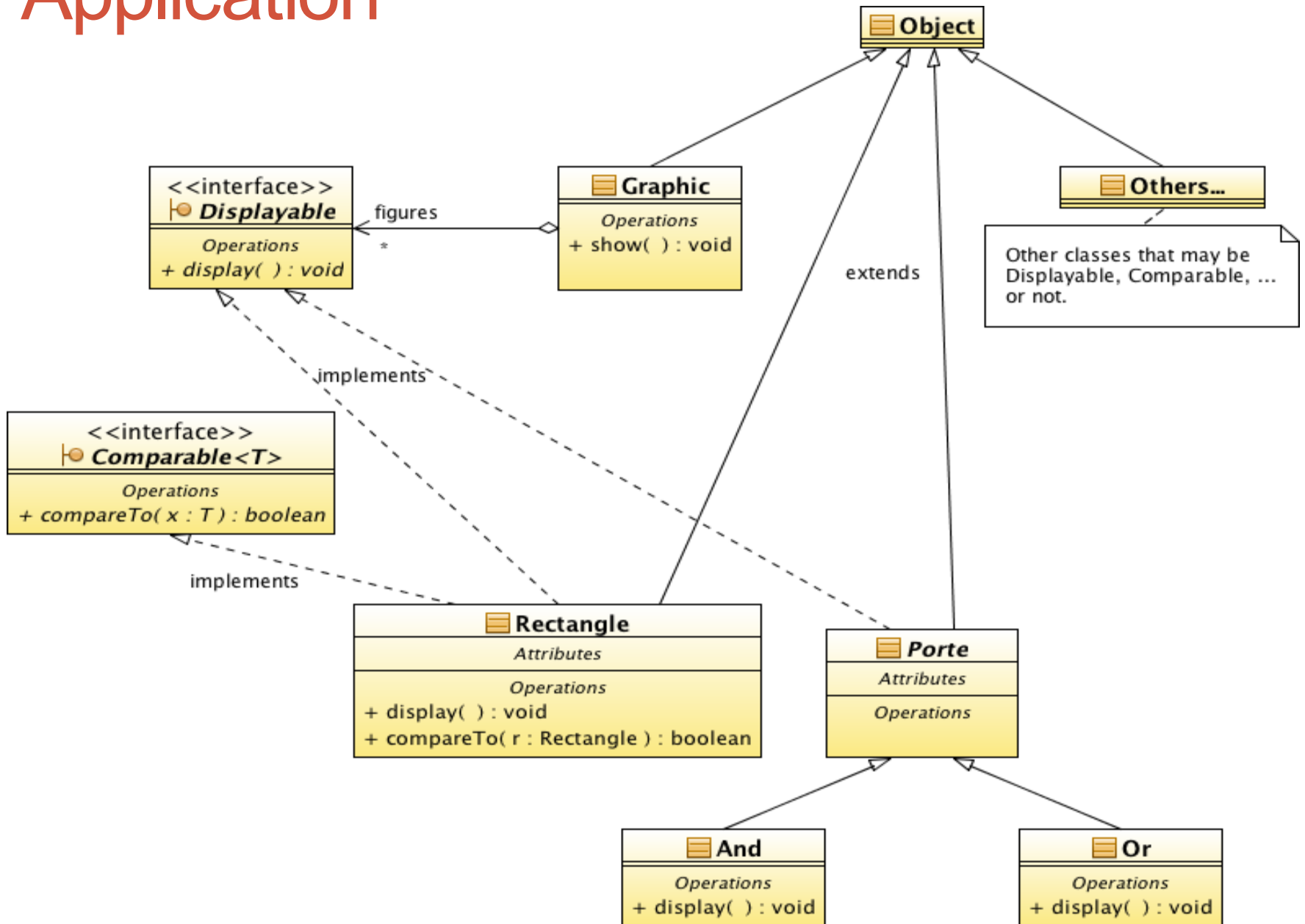
- Exemples (de la bibliothèque java.util)

```
public class ArrayList<E>  
    extends AbstractList<E>  
    implements List<E>, Cloneable, Serializable
```

- Remarque: une interface peut étendre (par **extends**) une ou plusieurs autre(s) interface(s)

```
public interface List<E> extends Collection<E>,  
    Iterable<E>
```

Application



Application en Java

```
public interface Displayable {  
    void display();  
}
```

```
public interface Comparable<T> {  
    int compareTo(T t);  
}
```

```
public class Rectangle implements Displayable,  
    Comparable<Rectangle> {  
    public double surface() {...}  
    public void display() {...}  
    public int compareTo(Rectangle r) {  
        return (int) (this.surface() - r.surface());  
    }  
}
```

Application en Java

```
public abstract class Porte implements Displayable {  
    // display() remains abstract  
}
```

```
public class And extends Porte {  
    public void display() { //code...}  
}
```

```
public class Or extends Porte {  
    public void display() { //code...}  
}
```

Application en Java

```
public class Graphic {
    protected List<Displayable> figures;
    public void add(Displayable fig) { figures.add(fig); }
    public void show() {
        for (Displayable fig : figures) fig.display();
    }
}
```

```
public class Pavage {
    protected List<Rectangle> rectangles;
    public void sort() {
        Collections.sort(rectangles); // <= Comparable
    }
}
```


Collection

- Les collections sont des regroupements dynamiques d'objets.
- Les formes les plus courantes sont:
 - les listes: accès indicé, doublons d'éléments possibles
 - les ensembles: sans doublons
- Les collections contiennent des objets => pour gérer des types primitifs utiliser les classes Wrapper

- Avant 5.0 : le type des éléments est Object
 - pas de contrôle statique de type plus fin
 - contrôle dynamique « 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 test .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` :

```
public boolean addAll(Collection c);
public boolean containsAll(Collection c);
```

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 chaînées: `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` d'origine avant Java5)

Les listes : java.util.List

```
public interface List<E> extends Collection<E>

    public void add(int index, E element)
    // sachant que add(E element) 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 1ere occurrence, -1 si !this.contains(o)

    public List<E> sublist(int from, int to) throws
        IndexOutOfBoundsException
```

Exemple

```
import java.util.*;

public class Circuit {
    protected List<Porte> composants = new ArrayList<Porte>();
    //ou ArrayList<Porte> composants = ...

    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();
        }
    }
}
```

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;  
    void remove(); //enlève le dernier element  
}
```

- Exemple de la classe `Circuit`

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

- et même en 5.0, comme tout **Iterable** (tab, Collection, ...):

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

Collections d'éléments de type primitif

- type primitif : utiliser les classes wrappers

```
List<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)); //ou plus simplement:
```

```
l.add(x); //"autoboxing" en 5.0 (automatique)
```

```
double s=0.0;
```

```
for(int i=0;i<l.size();i++){
```

```
    // s=s+l.get(i).doubleValue();
```

```
    s=s+l.get(i); // "auto-unboxing" en 5.0
```

```
}
```

```
s=0.0;
```

```
// par iteration "for each"
```

```
for(double x : l) s=s+x;
```

Collections avant 5.0

- pas de paramètre de type générique
- type statique des éléments = Object => casts « à la main »

```
public class Circuit {
    protected List composants = new ArrayList(); //Object

    public void brancher(Porte p) {
        composants.add(p); /*compatible avec Object*/
    }

    public void run() {
        for (int i=0;i<composants.size();i++){
            //restitution des elts => downcasts
            ((Porte)composants.elementAt(i)).run();
        }
    }

    public void afficher() {
        //iterators non generiques => downcasts
        Iterator iter = composants.iterator();
        while(iter.hasNext()) ((Porte)iter.next()).display();
    }
}
```


Les utilitaires de la classe Collections

- La classe Collections offre des utilitaires (static) sur les List :

- tri : `sort`
- recherche ordonnée : `binarySearch`

```
public class Collections {  
    public static void sort(List<E> list)  
    public static int binarySearch(List<E> list, E x) ...}
```

- Les éléments doivent fournir une relation d'ordre en implémentant 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  
}
```

- Il existe l'équivalent sur les tableaux fourni en static par la classe **Arrays**

Exemple

- Ouvrage : relation d'ordre sur leur auteur

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

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

Exemple

- Application : liste d'ouvrages ordonnée par auteur

```
List<Ouvrage> ouvrages = new ArrayList<Ouvrage>();
```

```
//ajouts
```

```
ouvrages.add(new Ouvrage("Germinal", "Zola"));
```

```
ouvrages.add(new Ouvrage("C", "Kernighan"));
```

```
ouvrages.add(new Ouvrage("Java", "Eckel"));
```

```
//tri
```

```
Collections.sort(ouvrages);
```

```
// affichage du resultat
```

```
Java Eckel
```

```
C Kernighan
```

```
Germinal Zola
```

Les tables d'association :

java.util.Map

- Elles permettent de maintenir des associations clé-valeur $\langle K, V \rangle$:
 - chaque clé est unique, elles constituent un Set
 - les clés et valeurs sont des objets
- L'interface 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);  
    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 (cf. Object)
 - 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 : leur classe de clé doit implémenter l'interface Comparable

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++;}  
    }  
}
```

Exemple 1/4

```
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 ...
```

Exemple 2/4

```
//...
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 {}
```


Exemple 3/4

```
// application (dans un 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
*/
```

Exemple 4/4

```
// application (suite du main)

String code; // a obtenir...

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