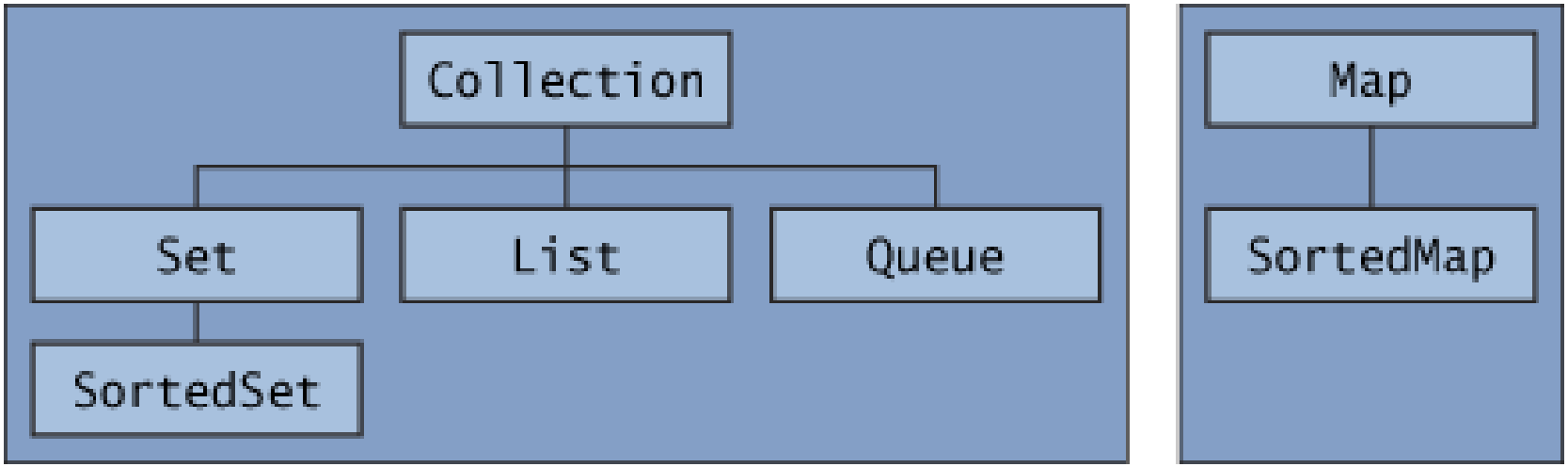
Programmation Objet
Java–Collections

Loubna KARBIL

L.karbil@uiz.ac.ma

Collections

- Dans la bibliothèque Java on retrouve
  - interfaces collection
  - implémentations
  - Algorithmes



Collections

Il s’agit d’interfaces génériques

- Collection<E>: base de la hierarchy
  - Set<E>: ensemble d’éléments de type E (sans duplication)
    - SortedSet<E>: ensembles ordonnés
  - List<E>: suite d’éléments de type E (avec duplication)
  - Queue<E>: file d’elements de type E
  - Deque<E>: « double-ended » queue, file et pile au même temps
- Map<K,V>: association clés-valeurs (clés de type K, valeurs de type V)
  - SortedMap<K,V> avec un ordre sur les clefs

L’interface Collection

- **Collection<E>** (interface) représente un groupe (une collection) d’objets de type E
  - Interface la plus abstraite de la bibliothèque
  - Intérêt : on utilise **Collection** pour maximiser la généralité de notre code
  - Tous les autres classes contiennent des constructeurs qui peuvent initialiser une autre structure avec une **Collection**
  - **Ex :**

```
public void f(Collection<String> c){
    List<String> l = new ArrayList<>(c) ;
    ..
}
```

  - On passe comme paramètre un objet qui implémente l’interface, dans la méthode on transforme l’objet en liste.

L’interface Collection

- Désavantage de l’interface **Collection** : trop générale, ne permet que d’accès basique sur les éléments.
- Cependant, on peut parcourir les éléments (suffit pour beaucoup de cas), et on peut ajouter des éléments
  - **Ex :**

```
import java.util.*;
class ListTest {
        public static void main(String [] args){
                Collection<String> l = new ArrayList<>();
                l.add("1");   l.add("2");  l.add("3");
                f(l);
        }
        public static void f(Collection<String> c){
                for(String s:c) System.out.println(s);
        }
}
```

L’interface Collection

- **NB :** l’interface Collection nous offre aussi quelques méthodes « optional »
  - **boolean remove(Object o) ;**
- Certains classes qui implémentent l’interface Collection n’ont pas une implémentation de remove
- A-t-on le droit de ne pas implémenter une méthode d’une interface implémentée ?
  - Non ! Le compilateur ne l’accepte pas !
- En réalité, ces classes contiennent une implémentation qui ne fait rien, et lance une UnsupportedOperationException
  - Violation des principes de l’OOP dans la bibliothèque de Java !
  - Mais → simplification massive
    - Sinon, on aurait trop de cas à traiter...

Iterator

- La forme for-each de la boucle for marche pour les collections, et en générale pour chaque objet qui implémente l'interface **Iterator<E>**

```
public interface Iterator<E> {

    boolean hasNext();

    E next();

    void remove(); //optional

}
```

- Autre manière de parcourir une collection : on utilise un Iterator

Iterator

```
class ListTest {
        public static void main(String [] args){
                Collection<Integer> l = new ArrayList<>();
                l.add(1); l.add(2); l.add(3); l.add(4);
                f(l);
        }
        public static void f(Collection<?> c){
                for(Iterator<?> i = c.iterator(); i.hasNext(); )
                        System.out.println(i.next());
        }
}
```

Méthode générique

Iterator

- La forme for-each de la boucle for marche pour les collections, et en générale pour chaque objet qui implémente l'interface **Iterator<E>**

```
public interface Iterator<E> {

    boolean hasNext();

    E next();

    void remove(); //optional

}
```

- Autre manière de parcourir une collection : on utilise un Iterator
  - Avantage : on peut utiliser remove() (pas possible avec la boucle for-each)
  - Avantage : on peut parcourir plusieurs collections à la fois

Listes

- List == Collection + Accès positionnel

```
public interface List<E> extends Collection<E> {
    void add(E e) ; //Normale pour collections
    void add(int index, E e) ; //À position index
    E get(int index) ; //Quasi-tableau
    ..
}
```

- Une list peut être utilisé comme une collection, mais les éléments sont numérotés (comme un tableau)
  - On peut tenter d'acceder à une position
    - Mais si elle n'existe pas → Exception
  - On peut insérer un élément au milieu
    - Les élément suivants sont déplacés
    - NB : on ne peut pas laisser des trous !

Listes

- L’interface List<E> est implémentée par deux classes :
  - ArrayList<E>
  - LinkedList<E>
- Différence :
  - Équivalentes en termes de fonctionnalité
  - Quelques opérations sont plus efficaces pour l’une ou l’autre (voir documentation)

Listes

- Ex :

```
class ListTest {
        public static void main(String [] args){
           List<Integer> l = new ArrayList<>();
           l.add(1); l.add(2); l.add(3); l.add(4);
           System.out.println(l); // [1,2,3,4]
           l.add(2,7);
           System.out.println(l); [1,2,7,3,4]
           l.add(8,8); // !!!! IndexOutOfBoundsException !
    }
}
```

Queue et Deque

- Queue : interface FIFO

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
     //aussi void add(E e) ; héritée
}
```

- Insertion/Suppression/Inspection en deux versions (add/offer, remove/poll, element/peek)
  - Une version lance exception en cas d'erreur
  - L'autre retourne null ou false

Queue et Deque

- Deque : interface FIFO et LIFO
  - On peut ajouter un élément au début ou à la fin
  - On peut prendre un élément au début ou à la fin
  - Comme avec Queue, deux versions de chaque opération

    addFirst, addLast – offerFirst, offerLast,

    …
- Implémentée par
  - ArrayDeque<E>
  - LinkedList<E>

Set

- Interface qui représente un ensemble où chaque élément appartient au plus une fois (pas de doublons)

```
import java.util.*;
```

-
```
public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            s.add(a);
      System.out.println(s.size() + " distinct words: " + s);
    }
}
```

- Implémentée par HashSet, TreeSet, LinkedHashSet

Map

- Interface qui représente un tableau associatif :
- Pour Map<K,V>
  - Pour chaque élément clé (de type K), on stocke un valeur unique (de type V)
  - → Un tableau de T peut être vu comme un Map<Integer,T> (mais avec la condition que les clés forment un intervalle 0..size-1)
- Méthodes principales :
  - **V get(Object k) ; //retourne null si l'objet k n'existe pas**
  - **void put(K k, V v) ;**
- Implémentée par HashMap, TreeMap, LinkedHashMap.

Map

```
public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
$ java Freq a s d s d e f
5 distinct words:
{a=1, s=2, d=2, e=1, f=1}
```

HashMap vs TreeMap

- La bibliothèque de Java fait un maximum d'effort pour maximiser l'abstraction
- Cependant, parfois les détails de l'implémentation ne peuvent pas être évités.
  - Ex : La classe HashMap (et la classe HashSet, etc.) exige la (re)définition de la méthode **int hashCode()**
  - Ex : La classe TreeMap exige que le type K (les clés) implémente l'interface **Comparable** (les clés sont triés).

Hashing

- Hash function : une fonction **hashCode()** qui étant donné un objet quelconque produit un **int** sous les conditions :
  - Si **a.equals(b)** alors **a.hashCode()==b.hashCode()**
  - Le même objet donne toujours la même valeur de **hashCode**.
  - Les valeurs produites sont aussi « aléatoires » (dispersées) que possible
- **NB :** C'est tout à fait possible que deux objets différents produisent le même hashCode
  - En fait c'est inévitable ! (pourquoi?)
- La fonction **hashCode** est définie dans la classe **Object**
  - → héritée partout
- Sa définition par défaut n'est pas satisfaisante pour des classes où on a redéfini **equals**

Hashing - Algorithmes

- L’intérêt de la fonction hashCode (si elle est bien définie) est qu’elle nous permet de partitionner rapidement nos données dans des classes petites et équilibrées.
- Comment ça marche? Supposez qu’on veut implémenter la classe Map
  - Avec un tableau : efficace mais trop inflexible
  - Avec une liste chaînée : trop inefficace (il faut parcourir toute la list pour chercher un élément)
- HashMap = un tableau des listes
  - Étant donné put(k,v), on ajoute v dans la liste qui correspond à la clé k. La liste est facile à trouver (tableau)
  - Si la correspondance clés ↔ listes est équilibrée, les listes sont beaucoup plus courtes → Efficacité !
- Bon hashCode → les clés sont dispersées aléatoirement, bonne pérformance !

Hashing - Leçons

- Les classes Hash* sont très utilisées. Pour les utiliser correctement
  - Si vous utiliser une classe qui redéfinit equals, il faut donner une implémentation de **hashCode**.
  - Il faut une implémentation
    - Quasi-aléatoire (→ bien équilibrée)
    - Mais persistante (une valeur constante pour chaque objet et les objets auxquels il est égale)
- Ex : Pour la classe BigNum, on peut multiplier les chiffres de notre nombre.
- Pour la classe String, Java multiplie les valeurs numériques de tous les caractères.
- ...

TreeMap

- Une autre manière très efficace d'implémenter ces classes est d'utiliser les arbres binaires de recherche.
  - Pour que ces algorithmes fonctionnent, il faut qu'on puisse comparer les clés
- → TreeMap ne fonctionne que si la classe des clés implémente l'interface Comparable<>
  - !! Violation des principes de l'OOP. Cette exigence n'est pas évident par sa signature
  - Cause : TreeMap extends Map, qui n'a pas cette condition
- Et si on tente d'utiliser TreeMap avec une mauvaise classe ?

TreeMap sans Comparable ?

```
class Foo { }

class ListTest {

        public static void main(String [] args){

                TreeMap<Foo,Integer> t = new TreeMap<>();

                Foo f = new Foo();

                t.put(f,2);//!!

                System.out.println(t);

        }

}
```

- Exception in thread "main" java.lang.ClassCastException: Foo cannot be cast to java.lang.Comparable

TreeMap avec Comparable

```
class Rational implements Comparable<Rational> {
        int n,d;
        public Rational(int n, int d) { this.n = n; this.d=d; }
        public int compareTo(Rational r){
                return n*r.d - r.n*d;
        }
        public boolean equals(Object o) {
                return compareTo((Rational)o)==0;
        }
        public String toString() { return n+"/"+d; }
}
```

TreeMap avec Comparable

```
class ListTest {
        public static void main(String [] args){
                TreeMap<Rational,Integer> t = new TreeMap<>();
                t.put(new Rational(1,7),2);
                t.put(new Rational(1,4),2);
                t.put(new Rational(1,2),2);
                t.put(new Rational(1,3),2);
                t.put(new Rational(1,6),2);
                System.out.println(t);
        }
}
```

- **$ java ListTest**

  {1/7=2, 1/6=2, 1/4=2, 1/3=2, 1/2=2}

TreeMap – Leçons

- Pour utiliser les classes de la bibliothèque qui utilisent les arbres de recherche il faut avoir une méthode de comparaison entre les éléments
  - → Interface Comparable
- Quand vous définissez une nouvelle classe
  - Redéfinissez la méthode toString toujours
  - Redéfinissez la méthode equals souvent
    - Dans ce cas, redéfinissez aussi hashCode
  - Si possible, implémentez Comparable