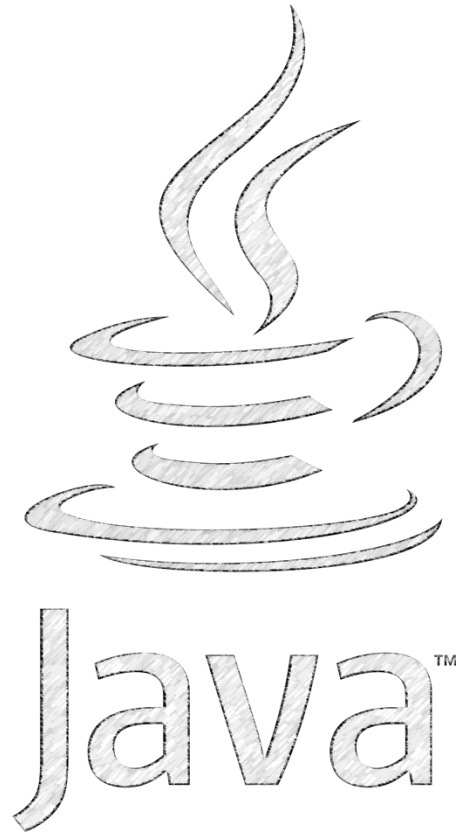


Programmation Orientée Objets avec Java

Stéphane Malandain, Yassin Rekik



Chapitre 4

Les collections

Concepts traités

2

- Java Collections Framework
- Tableaux associatifs (Map)

Java Collections Framework

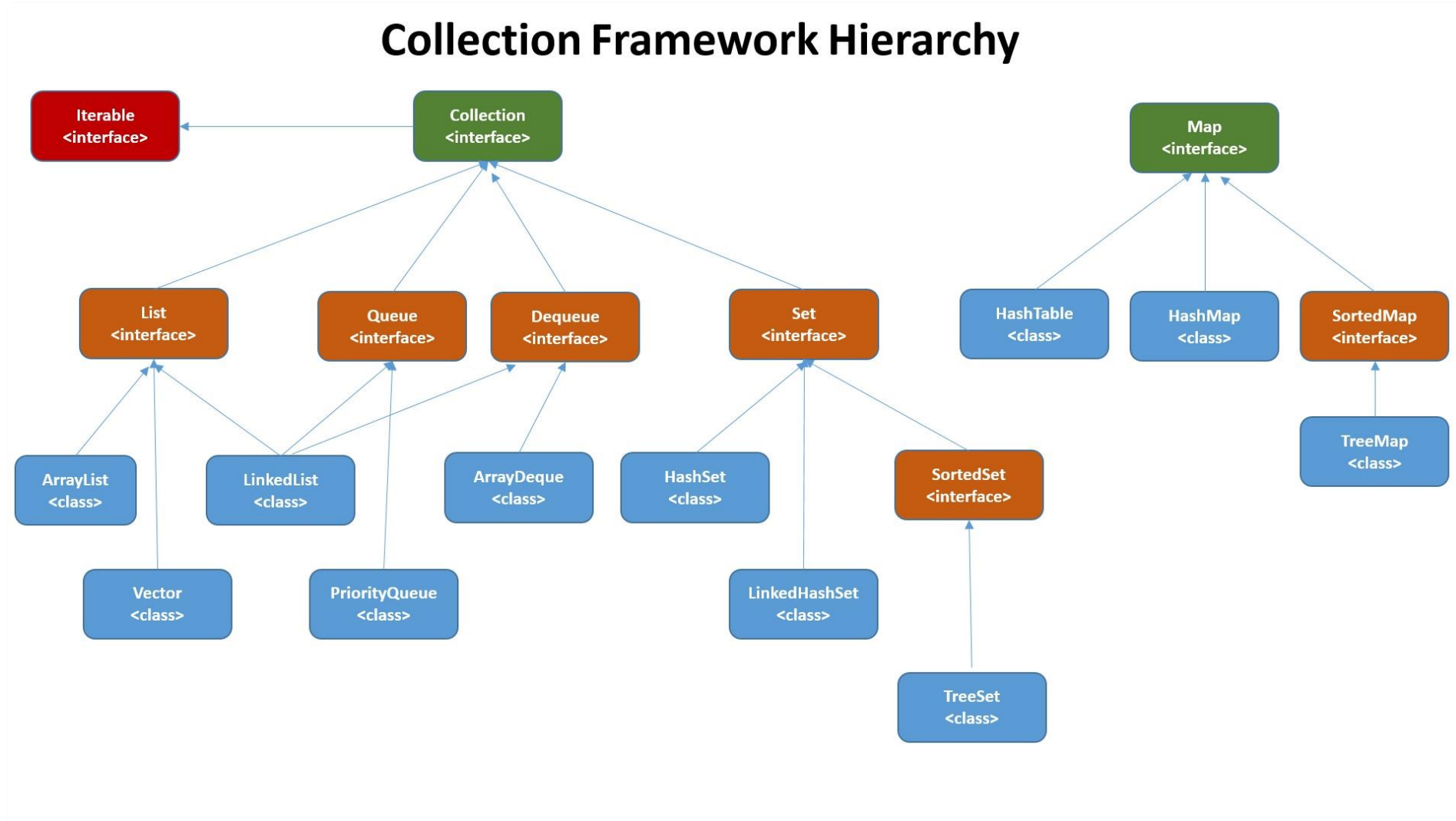
Java Collections Framework

4

- Le package `java.util` contient un ensemble d'outils pour la manipulation ou la réalisation de collections appelé '*Java Collections Framework*'.
- Le diagramme suivant illustre la hiérarchie des principales interfaces qui doivent être respectées par les implémentations

Principales Interfaces

5



Source : <https://facingissuesonit.com/2019/10/15/java-collection-framework-hierarchy/>

3 catégories de classes offertes par Java

6

- Les ensembles
 - Collections avec aucun élément de dupliqué (redéfinition du `equals` obligatoire)
 - L'implémentation `HashSet` utilise une table de hachage (redéfinition du `hashCode` obligatoire)
 - Les `SortedSet` proposent un ordre total des éléments (les objets doivent donc être `Comparable`)
- Les queues
 - Insèrent et récupèrent les éléments selon une stratégie (LIFO, FIFO, priorité de l'objet, etc.)
- Les listes
 - Séquences d'objets ou l'utilisateur a le contrôle sur l'endroit de la liste ou il doit insérer un élément

- L'interface principale (`Iterable`) indique que ces structures de données sont itérables :
 - Il est possible de les parcourir sans se soucier des détails d'implémentations (`list`, `stack`, `tree`, ...)
 - Un `Iterable` doit retourner un `iterator` (`Iterator`)
 - C'est celui-ci qui navigue à travers une collection
 - Le parcours se fait à l'aide d'un motif demande-récupération (`hasNext`/`next`)

- Toute collection qui peut être parcourue est itérable
- Elle respecte donc l'interface suivante :

```
1  public interface Iterable<T> {  
2      default void forEach(Consumer<? super T> action);  
3      Iterator<T> iterator();  
4      default Spliterator<T> spliterator();  
5  }
```

- Il suffit à un `Iterable` de retourner un `Iterator` pour avoir gratuitement la méthode `forEach` (nous ne parlons pas ici de `spliterator`)

- La méthode `forEach` prend un `Consumer`. Il s'agit d'une fonction qui prend un élément et ne retourne rien (`void`)
- On peut employer la syntaxe lambda :

```
List.of(1,2,3).forEach(i -> System.out.println(i) );
```

- La fonction lambda prend bien un élément de la collection de type entier et applique une action qui ne retourne rien. La méthode `println` affiche un résultat, mais ne retourne rien.

- Un itérateur est un objet mutable qui parcourt une collection sans modifier l'état de celle-ci.
- C'est l'itérateur qui garde l'état de l'élément courant sur lequel il pointe. C'est donc une classe statique.
- Grâce à un itérateur, on peut parcourir des collections de cette manière :

```
1  Iterator<Integer> it = Set.of(1,2,4).iterator();
2  while( it.hasNext() ) { // il est nécessaire de demander si l'itérateur
3      |           |           |           |           // a encore quelque chose à "consommer"
4      |           int i = it.next(); // si c'est le cas, l'élément peut être récupéré (référéncé)
5      |           System.out.println(i);
6      }
```

- Si l'appel à `hasNext()` retourne `false`, un appel à `next()` lève une exception.

- Interface d'un Iterator :

```
1  public interface Iterator<E> {  
2      default void forEachRemaining(Consumer<? super E> action);  
3      boolean hasNext();  
4      E next();  
5      default void remove();  
6  }
```

- Toute classe qui implémente `Iterator` doit implémenter `next` et `hasNext`
- La méthode `remove` est réalisée par défaut. Elle permet de supprimer un élément de la liste d'un parcours. Réalisée, elle retourne une exception ! Par défaut, elle interdit donc de supprimer un élément.

Iterable<E> et Iterator<E>

13

- L'avantage de respecter cette convention iterable-iterator est qu'elle permet d'utiliser la boucle de parcours (`foreach`)

```
1  for(int i: Set.of(1,2,4)) {  
2      System.out.println(i);  
3  }  
4  Set.of(1,2,4).forEach( i -> System.out.println(i) )
```

- Complétez la classe `Test` pour permettre de parcourir ses éléments.

```
1  public class Test implements Iterable<Integer> {  
2      private int[] is = {1,2,3,4,4,3,2};  
3      public Iterator<Integer> iterator() {  
4  
5  
6  
7  
8  
9      }  
10 }
```

```
1  for(int i: new Test()) {  
2      ...  
3  } // doit afficher 1,2,3,4,4,3,2  
4
```

- A l'aide d'une classe privée non statique :

```
1  public class Test implements Iterable<Integer> {
2      // privée car nous ne souhaitons pas que l'utilisateur puisse créer lui-même
3      // un TestIterator
4      private class TestIterator implements Iterator<Integer> {
5          private int index = 0;
6          public boolean hasNext() {
7              return index < is.length; // /\ ne pas utiliser `this`
8          }
9          public Integer next() {
10             int res = is[index];
11             index += 1;
12             return res;
13         }
14     }
15     private int[] is = {1,2,3,4,4,3,2};
16     public Iterator<Integer> iterator() {
17         return new TestIterator();
18     }
19 }
```

- A l'aide d'une classe anonyme :

```
1  public class Test implements Iterable<Integer> {
2      private int[] is = {1,2,3,4,4,3,2};
3      public Iterator<Integer> iterator() {
4          return new Iterator<Integer>() {
5              private int index = 0;
6              public boolean hasNext() {
7                  return index < is.length; // /\ ne pas utiliser `this`
8              }
9              public Integer next() {
10                 int res = is[index];
11                 index += 1;
12                 return res;
13             }
14         };
15     }
16 }
```


Collection et List

17

- Un `Iterable` ne propose pas de fonctionnalité pour ajouter, supprimer ou récupérer un élément spécifique
- L'interface `Collection` indique comment doit se comporter une collection :

```
1  public interface Collection<E> extends Iterable<E> {  
2      boolean add(E e);  
3      boolean remove(Object o);  
4      boolean isEmpty();  
5      boolean contains(Object o);  
6      void clear();  
7      int size();  
8      boolean removeAll(Collection<?> c);  
9      default boolean removeIf(Predicate<? super E> filter);  
10     default Stream<E> stream();  
11 }
```

- Elle ne propose pas de fonctionnalités pour récupérer ou insérer un élément à une position donnée
- C'est normal puisque cela dépend du type de la structure (`Set`, `Queue` ou `List`)
- Par exemple, une queue ne peut récupérer que l'élément de tête ou encore il n'est pas possible d'insérer un élément à une position arbitraire

Collection et List

19

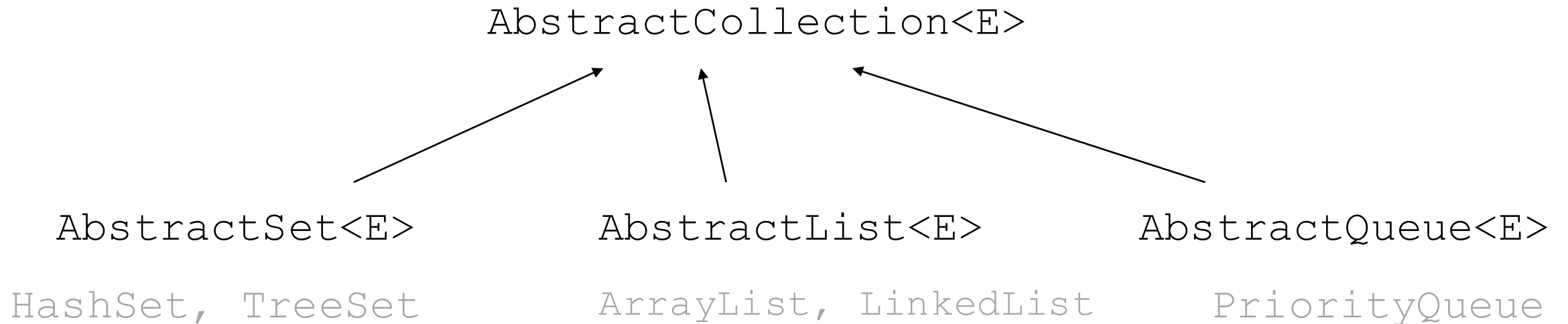
- Dans le cas d'une `List`, d'autres méthodes indiquent leur comportement
- Extrait de l'interface `List` :

```
1  public interface List<E> extends Collection<E> {  
2      void add(int index, E element);  
3      E get(int index);  
4      boolean remove(int index);  
5      ...  
6  }
```

Classes abstraites du framework

20

- Classes abstraites disponibles pour réduire l'effort car certaines choses se répètent pour plusieurs implémentations
- Le diagramme de classe ci-dessous illustre la hiérarchie des classes abstraites du framework :



Exemple : la classe ArrayList

21

- Dans le cas de `ArrayList` par exemple, la classe respecte l'interface `List` et hérite de la classe `AbstractList`

```
1  public class ArrayList<E>
2  extends AbstractList<E>
3  implements List<E>, RandomAccess, Cloneable, Serializable
4
```

Exemple : la classe ArrayList

22

- `AbstractList` implémente elle-même l'interface `List`
- Elle met à disposition un `Iterator` pour n'importe quel type de `List`.
- Il suffit à l'utilisateur de redéfinir les méthodes `size()` et `get(int)` pour une liste immuable, et les méthodes `set(int, E)` et `remove(int)` pour une liste mutable
- La méthode `size` est abstraite dans `List`, la méthode `get(int)` dans `AbstractList`; Elles doivent donc obligatoirement être redéfinies !
- Les méthodes `set` et `remove` ne sont pas abstraites et retournent une exception si elles sont appelées et qu'elles ne sont pas correctement redéfinies

- Il faut bien faire la distinction entre une méthode abstraite et une méthode concrète qui retourne une exception !
- Dans le premier cas, le compilateur refusera de compiler si elle n'est pas redéfinie.
- Dans le second, il s'agira d'une erreur d'exécution !

- Que peut-on reprocher à cette méthode ?

```
1  public void test(ArrayList<Integer> myList) {  
2      |    for(int i: myList) {  
3          |    /* do something with i */  
4          |    }  
5      |    }  
6  }
```


- Le reproche est d'être beaucoup trop conservateur.
- Nous obligeons l'utilisateur de test à nous donner une `ArrayList` alors que nous faisons que parcourir celle-ci. Si l'utilisateur se retrouve avec un autre type de collections, celui-ci devra la transformer pour appeler `test`.
- Aussi, il aurait été préférable dans ce cas d'accepter un `Iterable` comme argument.

List / ArrayList : Examples

26

```
1  List<Integer> list1 = new ArrayList<>();
2  list1.add(4);
3  int i = list1.get(0); // i=4;
4  List<Integer> list2 = List.of(5,6,7); // Immuable
5  Collection<Integer> col = new LinkedList<>(); // doublement chaînée
6
```

List / ArrayList : Examples

27

```
1 List.of(1,2,3,4,5,6,7).forEach( v -> System.out.println(v));  
2 List.of(1,2,3,4,5,6,7).forEach( System.out::println(v));
```

```
1 // consumer: T -> ()  
2 void forEach(Consumer<T> consumer)  
3
```

<https://docs.oracle.com/javase/10/docs/api/java/util/List.html>

Map : les tableaux associatifs

28

- Interface `Map<K, V>`
- Classe abstraite associée : `AbstractMap<K, V>`
- Structure de données paramétrique clé-valeur
 - `K` : Le type de la clé
 - `V` : le type de la valeur
- A chaque clé est associée une valeur
- Peut être représenté par une liste de couples
 - `[(K1, V1), (K2, V2), (K3, V3)]`

Map : les tableaux associatifs

29

Implémentations courantes

- `HashMap<K, V>`
 - La clé doit redéfinir la méthode `hashCode()`
- `TreeMap<K, V>`
 - Implémente également l'interface `SortedMap<K, V>`
 - La clé doit redéfinir la méthode `equals`
 - La clé doit implémenter l'interface `Comparable<T>`

Map : Example

30

```
1  Map<String, String> airport = new HashMap<>();
2
3  airport.put("LHR", "Londres Heathrow");
4  airport.put("GVA", "Aéroport international de Genève");
5  airport.put("ORY", "Aéroport de Paris-Orly");
6
7  String orlyFullName = airport.get("ORY");
8
```

Map : La classe TreeMap

31

`TreeMap` implémente l'interface `Map` et `navigableMap`. La map est triée selon l'ordre naturel de ses clés ou un comparateur fourni lors de sa création. C'est un moyen efficace de trier et de stocker les paires clé-valeur.

- Caractéristiques :
 - La classe `TreeMap` contient des valeurs basées sur la clé.
 - Elle ne contient que des éléments uniques
 - Elle ne peut pas avoir de clé nulle mais peut avoir plusieurs valeurs nulles
 - Elle n'est pas synchronisée
 - La classe `TreeMap` maintient l'ordre croissant.

TreeMap : Example

32

```
1  import java.util.*;
2  public class Main{
3
4      public static void main(String args[]){
5
6          TreeMap<Integer, String> map = new TreeMap<Integer, String>();
7          map.put(21, "Pierre");
8          map.put(22, "Paul");
9          map.put(23, "Python");
10
11          System.out.println("La taille du map est: "+ map.size());
12
13          //Parcourir les éléments du Map
14          for(Map.Entry mp:map.entrySet()){
15              System.out.println(mp.getKey()+" "+mp.getValue());
16          }
17
18          map.remove(22, "Paul");
19          System.out.println("Après la suppression: "+map);
20      }
21  }
```