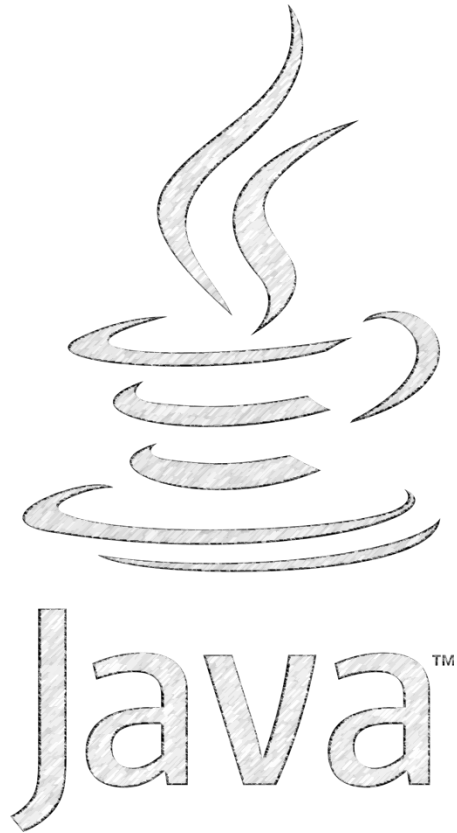


Programmation Orientée Objets avec Java

Stéphane Malandain, Yassin Rekik



Chapitre 4 part 2

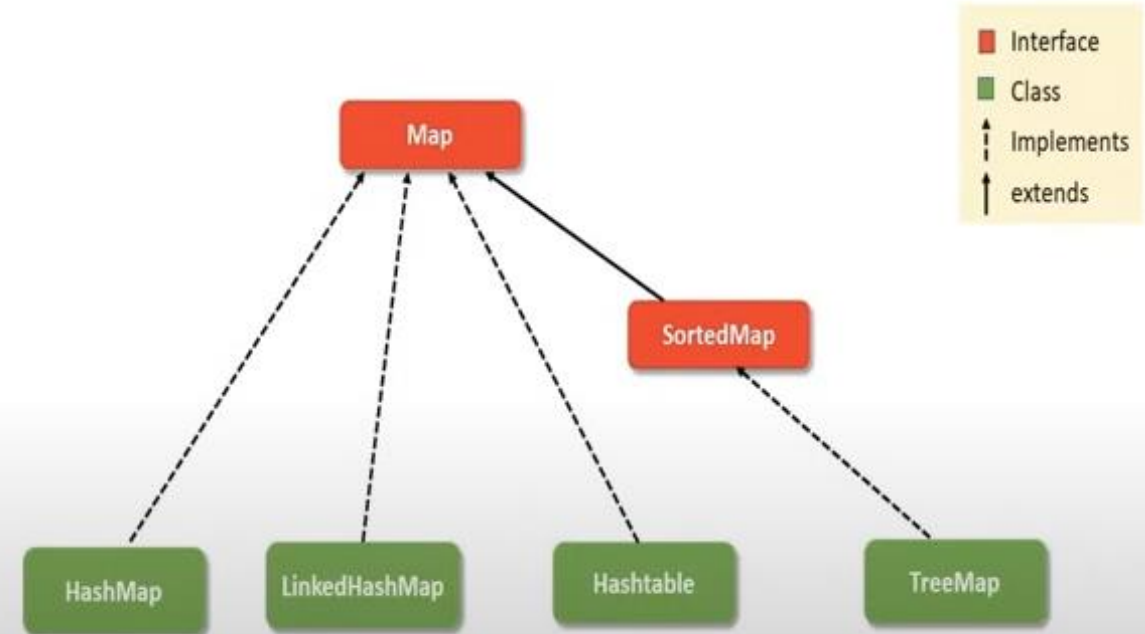
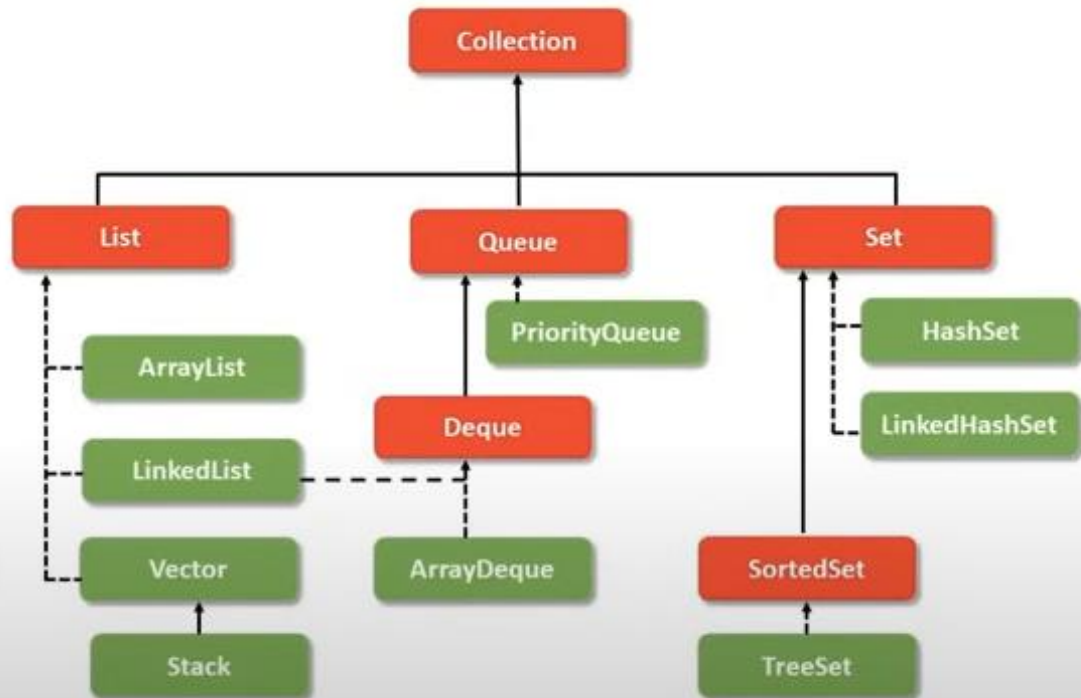
Bien choisir sa collection

Introduction

- Bien choisir une collection est essentiel pour optimiser les performances et assurer le comportement correct de notre application
- Java fournit un ensemble de classes très riches avec les interfaces `List`, `Map` et `Set`
- Elles sont adaptées à différentes utilisations avec des caractéristiques propres.

Principales Interfaces

3



Les listes : ArrayList et LinkedList

4

- Les listes sont des collections ordonnées qui autorisent les éléments dupliqués
 - Les **ArrayList** sont des tableaux dynamiques qui offrent un accès aléatoire très rapide en $O(1)$
 - Idéal pour un scénario où l'accès par index est essentiel
 - Les **LinkedList** sont des listes chaînées qui offrent un temps d'accès constant pour l'insertion et la suppression mais un temps d'accès aléatoire moins rapide en $O(n)$
 - Les `ArrayList` sont couramment préférées dans la plupart des cas à cause d'un temps d'accès aléatoire bien plus intéressant.
 - Utilisez les `LinkedList` quand vous avez spécifiquement besoin d'insertions et de suppressions fréquentes à des positions arbitraires.

Les listes : ArrayList et LinkedList

5

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class ListExample {
5      Run | Debug
6      public static void main(String[] args) {
7
8          List<String> arrayList = new ArrayList<>();
9          arrayList.add(e:"Pomme");
10         arrayList.add(e:"Banane");
11         arrayList.add(e:"Orange");
12         arrayList.get(index:1);
13         arrayList.remove(index:2);
14         arrayList.remove(o:"Pomme");
15
16         List<String> linkedList = new LinkedList<>();
17         linkedList.add(e:"Pomme");
18         linkedList.add(e:"Banane");
19         linkedList.add(e:"Orange");
20         linkedList.get(index:1);
21         linkedList.remove(index:2);
22         linkedList.remove(o:"Pomme");
23     }
24 }
```

Les queues : Queue et Deque

6

- Les queues (files) sont des collections qui permettent de gérer les données dans un ordre spécifique.
 - Une **Queue** est une collection ordonnée où l'ajout de nouveaux éléments se fait à une extrémité (tail) de la queue et la suppression à l'autre extrémité (head) (FIFO)
 - Les queues sont essentielles dans des scénarios où vous devez maintenir un ordre spécifique dans le traitement des éléments.
 - L'interface `Deque` permet de définir une autre stratégie comme le LIFO. Elle offre deux points d'accès pour l'ajout et la suppression des éléments. Si vous avez besoin d'un retrait rapide de n'importe quel point de la collection, la `Deque` brille par sa polyvalence
 - La classe `PriorityQueue` étend directement `Queue` et fonctionne sur une stratégie LIFO mais avec les éléments ordonnés par ordre de priorité.

Les queues : Queue et Deque

7

```
1  import java.util.*;
2  public class ArrayDequeExample {
    Run | Debug
3      public static void main(String[] args) {
4
5          Deque<String> deque = new ArrayDeque<String>();
6
7          deque.add(e:"One");
8          deque.addFirst(e:"Two");
9          deque.addLast(e:"Three");
10
11         for (String str : deque) {
12             System.out.println(str);
13         }
14     }
15 }
```

Les queues : Queue et Deque

8

```
1  import java.util.*;
2  public class PriorityQueueExample {
    Run | Debug
3      public static void main(String[] args) {
4
5          PriorityQueue<String> priorityQueue =
6              new PriorityQueue<>(Comparator.comparingInt(String::length));
7
8          priorityQueue.add(e: "Banane");
9          priorityQueue.add(e: "Pomme");
10         priorityQueue.add(e: "Grenade");
11
12         while (!priorityQueue.isEmpty()) {
13             System.out.println(priorityQueue.remove());
14         }
15     }
16 }
```


Les ensembles : Set et SortedSet

9

- Les `set` sont des collections qui refusent les doublons et qui contiennent au plus un élément `null`. Les `set` se basent sur la méthode `equals` de chaque objet de la collection.
 - Plusieurs implémentations de l'interface `Set` : `HashSet`, `LinkedHashSet` et `TreeSet`
 - Les `HashSet` utilisent une table de hachage pour stocker les éléments, permettant ainsi d'être certain que les éléments sont uniques. Ils offrent des opérations d'ajout, de recherche et de suppression efficaces en $O(1)$ (dépend de la fonction de hachage)
 - Chaque objet inséré doit fournir une implémentation des méthodes `hashCode()` et `equals()`.
 - La méthode `hashCode()` permet de calculer le code de hachage de l'objet et détermine l'emplacement de stockage de l'objet dans la table de hachage,
 - La méthode `equals()` permet de vérifier l'égalité entre deux objets lors de l'ajout ou de la recherche afin d'être sûr qu'il n'y ait pas de doublons.

Les ensembles : HashSet

10

```
1  import java.util.*;
2  public class HashSetExample {
    Run | Debug
3      public static void main(String[] args) {
4
5          HashSet panier = new HashSet<>();
6
7          panier.add(e: "Banane");
8          panier.add(e: "Pomme");
9          panier.add(e: "Grenade");
10         panier.add(e: "Mozzarella");
11         panier.add(e: "Pain");
12
13         // affiche les éléments dans l'ordre du HashSet
14         // Banane, Pain, Mozzarella, Grenade, Pomme
15         panier.stream().forEach(System.out::println);
16     }
17 }
```

- La classe `LinkedHashSet` possède une liste doublement chaînée qui lui permet de maintenir les éléments dans leur ordre d'insertion. Cette caractéristique ralentie légèrement les performances.
- Cette classe est très utile lorsque vous avez besoin d'un ensemble sans doublons et que vous voulez maintenir l'ordre d'insertion des éléments.
- Les éléments dans un `TreeSet` sont triés selon leur ordre naturel, ou à l'aide d'un `Comparator` fourni lors de la création du `TreeSet`.
- L'ordre naturel est défini par l'interface `Comparable` que les objets stockés doivent implémenter.

Les ensembles : LinkedHashSet

12

```
1  import java.util.*;
2  public class SetExample {
3      public static void main(String[] args) {
4          //
5          LinkedHashSet<String> panier = new LinkedHashSet<>();
6          //
7          panier.add("Banane");
8          panier.add("Pomme");
9          panier.add("Grenade");
10         panier.add("Mozzarella");
11         panier.add("Pain");
12         //
13         // affiche les éléments dans l'ordre d'insertion
14         panier.stream().forEach(System.out::println);
15     }
16 }
```

Les ensembles : TreeSet

13

```
1  import java.util.*;
2  public class TreeSetExample {
3      public static void main(String[] args) {
4         
5              TreeSet<String> panier = new TreeSet<>();
6         
7              panier.add("Banane");
8              panier.add("Pomme");
9              panier.add("Grenade");
10             panier.add("Mozzarella");
11             panier.add("Pain");
12         
13             // affiche les éléments dans l'ordre alphabétique
14             panier.stream().forEach(System.out::println);
15         }
16     }
```

- Un `set` est recommandé dans les situations où l'unicité des éléments est cruciale.
- Conseils pour une utilisation optimale :
 - Choisir l'implémentation adaptée : La sélection entre `HashSet`, `LinkedHashSet` et `TreeSet` doit être basée sur les exigences de votre application en tenant compte de l'importance de l'ordre des éléments et des performances des opérations de manipulation de données.
 - Implémenter correctement les méthodes `hashCode()` et `equals()`.
 - Considérer les performances : `TreeSet` fournit un ordre trié, mais ses performances peuvent être inférieures à celles de `HashSet` pour de grandes collections.

hashCode() et equals()

15

```
1  import java.time.LocalDate;
2
3  public class Student {
4      String surname;
5      String name;
6      String secondName;
7      LocalDate birthday;
8
9      public Student(String surname, String name, String secondName, LocalDate birthday ){
10         this.surname = surname;
11         this.name = name;
12         this.secondName = secondName;
13         this.birthday = birthday;
14     }
15
16     @Override
17     public int hashCode(){
18         return (surname + name + secondName + birthday).hashCode();
19     }
20     @Override
21     public boolean equals(Object other_) {
22         Student other = (Student)other_;
23         return (surname == null || surname.equals(other.surname) )
24             && (name == null || name.equals(other.name))
25             && (secondName == null || secondName.equals(other.secondName))
26             && (birthday == null || birthday.equals(other.birthday));
27     }
28 }
29
```

hashCode() et equals()

16

```
1  import java.time.LocalDate;
2  import java.time.Month;
3
4  public class HashCodeExample {
5      Run | Debug
6      public static void main(String[] args) {
7          Student sarah1 = new Student(surname:"Sarah",name:"Connor", secondName:"Jane", LocalDate.of(year:1970, Month.JANUARY, dayOfMonth:01));
8          Student sarah2 = new Student(surname:"Sarah",name:"Connor", secondName:"Jane", LocalDate.of(year:1970, Month.JANUARY, dayOfMonth:01));
9          Student sarah3 = new Student(surname:"Sarah",name:"Connor", secondName:"Jane", LocalDate.of(year:1959, Month.FEBRUARY, dayOfMonth:28));
10         Student john = new Student(surname:"John",name:"Connor", secondName:"Kyle", LocalDate.of(year:1975, Month.FEBRUARY, dayOfMonth:28));
11         Student johnny = new Student(surname:"John",name:"Connor", secondName:"Kyle", LocalDate.of(year:1985, Month.FEBRUARY, dayOfMonth:28));
12
13         System.out.println(sarah1.hashCode());
14         System.out.println(sarah2.hashCode());
15         System.out.println(sarah3.hashCode());
16         System.out.println(john.hashCode());
17         System.out.println(johnny.hashCode());
18         System.out.println(sarah1.equals(sarah2));
19     }
20 }
```

```
malandai@MacBook16 testcours % java HashCodeExample
1051737769
1051737769
1258712792
282015947
2024826282
true
```


les tableaux associatifs : Map

17

- Les `map`, ou dictionnaires, sont des structures de données qui associent des clés uniques à des valeurs.
- Avantage principal : un accès rapide aux valeurs en utilisant les clés.
 - Plusieurs implémentations de l'interface `Map` : `HashMap`, `HashTable`, `LinkedHashMap` et `TreeMap`
 - `HashMap` est l'implémentation de `Map` la plus utilisée. Elle stocke les informations par paires clé-valeur. Les clés permettent d'accéder aux valeurs de la liste. Le stockage est fait par hachage pour garantir un accès rapide.
 - Structure idéale pour des recherches rapides.
 - `HashTable` : Similaire à `HashMap` mais synchronisée pour fournir un environnement thread-safe. Performances légèrement inférieures. Notez que l'ordre des clés diffère.

les tableaux associatifs : Map

18

```
1  public class Champignon {
2      String nom;
3      int taille;
4      boolean estComestible;
5
6      public Champignon(String nom, int taille, boolean estComestible ){
7          this.nom = nom;
8          this.taille = taille;
9          this.estComestible = estComestible;
10     }
11
12     @Override
13     public String toString() {
14         return nom + " : " + taille + "cm" + (estComestible ? "" : " non ") + "comestible";
15     }
16 }
17
```

les tableaux associatifs : Map

19

```
1  import java.util.HashMap;
2
3  public class HashMapExample {
    Run | Debug
4      public static void main(String[] args) {
5
6          HashMap<Integer, Champignon> bois = new HashMap<>();
7
8          bois.put(key:1,new Champignon(nom:"Bolet", taille:7, estComestible:true));
9          bois.put(key:3,new Champignon(nom:"Amanite", taille:10, estComestible:false));
10         bois.put(key:4,new Champignon(nom:"Girolle", taille:8, estComestible:true));
11         bois.put(key:2,new Champignon(nom:"Lépiote", taille:12, estComestible:false));
12         // Affiche Amanite : 10cm non comestible
13         System.out.println(bois.get(key:3));
14         // Affiche 1,2,3,4
15         bois.keySet().stream().forEach(System.out::println);
16     }
17 }
```

les tableaux associatifs : Map

20

```
1  import java.util.Hashtable;
2
3  public class HashtableExample {
4      Run | Debug
5      public static void main(String[] args) {
6          Hashtable<Integer, Champignon> bois = new Hashtable<>();
7
8          bois.put(key:1,new Champignon(nom:"Bolet", taille:7, estComestible:true));
9          bois.put(key:3,new Champignon(nom:"Amanite", taille:10, estComestible:false));
10         bois.put(key:4,new Champignon(nom:"Girolle", taille:8, estComestible:true));
11         bois.put(key:2,new Champignon(nom:"Lépiote", taille:12, estComestible:false));
12         // Affiche Lépiote : 12cm non comestible
13         System.out.println(bois.get(key:2));
14         // Affiche 4,3,2,1
15         bois.keySet().stream().forEach(System.out::println);
16     }
17 }
```

les tableaux associatifs : Map

21

- `LinkedHashMap` et `TreeMap`
 - `LinkedHashMap` est une extension de `HashMap`. Elle maintient en plus une liste des entrées dans l'ordre dans lequel elles ont été insérées.
 - Utile pour les applications qui nécessitent un ordre d'itération prévisible.
 - `TreeMap` est idéale quand il s'agit de maintenir les éléments triés selon l'ordre naturel de leurs clés ou selon un ordre spécifié par un comparateur.
 - Utile par exemple pour des applications qui nécessitent un tri constant des données.

les tableaux associatifs : Map

22

```
1  import java.util.LinkedHashMap;
2
3  public class LinkedHashMapExample {
4      Run | Debug
5      public static void main(String[] args) {
6
7          LinkedHashMap<Integer, Champignon> bois = new LinkedHashMap<>();
8
9          bois.put(key:1,new Champignon(nom:"Bolet", taille:7, estComestible:true));
10         bois.put(key:3,new Champignon(nom:"Amanite", taille:10, estComestible:false));
11         bois.put(key:4,new Champignon(nom:"Girolle", taille:8, estComestible:true));
12         bois.put(key:2,new Champignon(nom:"Lépiote", taille:12, estComestible:false));
13         // Affiche Lépiote : 12cm non comestible
14         System.out.println(bois.get(key:2));
15         // Affiche 1,3,4,2
16         bois.keySet().stream().forEach(System.out::println);
17     }
```

les tableaux associatifs : Map

23

```
1  import java.util.TreeMap;
2
3  public class TreeMapExample {
4      Run | Debug
5      public static void main(String[] args) {
6          TreeMap<Integer, Champignon> bois = new TreeMap<>();
7
8          bois.put(key:1,new Champignon(nom:"Bolet", taille:7, estComestible:true));
9          bois.put(key:3,new Champignon(nom:"Amanite", taille:10, estComestible:false));
10         bois.put(key:4,new Champignon(nom:"Girolle", taille:8, estComestible:true));
11         bois.put(key:2,new Champignon(nom:"Lépiote", taille:12, estComestible:false));
12         // Affiche Lépiote : 12cm non comestible
13         System.out.println(bois.get(key:2));
14         // Affiche 1,2,3,4
15         bois.keySet().stream().forEach(System.out::println);
16     }
17 }
```