

## L3. Programmation orientée objet. Cours 4

Marie-Pierre Béal (Cours de Rémi Forax)

Collections et Maps



Collections

# Collections

# Battery Included

Il y a deux philosophies pour les bibliothèques incluses avec un langage

- “Minimaliste”, comme en C, une seule bibliothèque (stdlib) qui ne contient que les fonctions essentielles
- “Battery Included”, comme en Java ou Python, le langage vient avec beaucoup d’API différentes qui co-évoluent avec le langage

# L'API des collections

L'API des collections (`java.util`) fournit les structures de données usuelles

L'API est organisée en séparant les interfaces de leurs implantations

- Une interface indique l'ensemble des méthodes disponibles (`List`, `Set`, `Deque`, `Map`)
- Une implantation fournit chaque méthode avec une complexité particulière
- Les implantations peuvent être
  - nommées, avec une **classe publique** accessible
  - anonymes, accessibles à travers une **méthode `public static`**

# List et Map

Les deux interfaces les plus utilisées sont

- List, qui définit une liste d'éléments
  - indexée par un entier
  - ordonnée par l'ordre d'insertion
- Map, qui définit un dictionnaire de couples clé/valeur
  - indexé par une clé
  - avec plusieurs ordres possibles (dépend de l'implantation)

Chacune a des implantations modifiables et non modifiables

# Contraintes sur les éléments

Pour fonctionner correctement

- les éléments d'une List
- les clés ou valeurs d'une Map

doivent implanter les méthodes

- equals(Object)
- hashCode()
- toString()

Pas de warning, pas d'erreur si vous les oubliez, le comportement sera juste incohérent !

java.util.List

java.util.List

# List.of vs ArrayList

Il y a deux implantations principales

- List.of() pour les listes non modifiables

```
var list = List.of("hello", "collection");
```

- ArrayList qui s'agrandit dynamiquement

```
var list = new ArrayList<String>();  
list.add("hello");  
list.add("collection");
```

## java.util.List.of()

Liste non modifiable qui n'accepte pas null

- `List.of(E ...)`

```
var empty = List.<String>of(); // on doit indiquer le type
var one = List.of(42);
var three = List.of(1, 5, 8);
```

- `List.of()` avec un tableau d'objets, copie les valeurs

```
var array = new String[] { "hello", "collection" };
var list = List.of(array);
```

`list.isEmpty()`, `list.size()`

`list.isEmpty()` renvoie true si la liste est vide

```
List.of("hello").isEmpty() // false
```

`list.size()` renvoie le nombre d'éléments

```
List.of("hello").size() // 1
```

`list.get()`, `list.getFirst()`, `list.getLast()`

`list.get(index)` renvoie le nième élément (à partir de 0)

```
List.of("hello", "list", "map").get(1) // list
```

- Lève une exception `IndexOutOfBoundsException` si on sort des bornes de la liste

Depuis Java 21, les listes sont dans des classes qui implémentent l'interface `SequencedCollection`.

`list.getFirst()` renvoie le premier élément

`list.getLast()` renvoie le dernier élément

```
List.of("hello", "list").getFirst() // hello  
List.of("hello", "list").getLast() // list
```

- Lèvent une exception `NoSuchElementException` si la liste est vide

toString() / equals() / hashCode()

`list.toString()` renvoie une version textuelle

```
List.of(1, 2, 3).toString() // [1, 2, 3]
```

`list.equals(list2)` permet de comparer des listes même si ce n'est pas la même implantation

```
var list = List.of(1, 2, 3);  
IO.println(list.equals(list)); // true  
var list2 = new ArrayList<Integer>();  
list2.add(1);  
list2.add(2);  
list2.add(3);  
IO.println(list.equals(list2)); // true
```

`list.hashCode()` renvoie une valeur résumée du contenu de la liste

```
List.of(1, 2, 3).hashCode() // 30817
```

## Parcours avec for(:)

On peut utiliser la syntaxe for(:) sur les listes

```
var list = List.of(1, 2, 3);  
for(var element: list) {  
    IO.println(element);  
    // 1  
    // 2  
    // 3  
}
```

C'est la même syntaxe que pour les tableaux mais le code généré par le compilateur n'utilise pas des indices (mais un Iterator, voir plus tard).

# list.set()

`list.set(index, element)` change la valeur de l'élément à la nième case

```
var list = new ArrayList<String>();  
list.add("hello");  
list.set(0, "bonjour");  
IO.println(list.getFirst()); // bonjour
```

## `list.add()` / `list.remove()`

`list.add(element)` ajoute un élément à la fin

`list.addLast(element)` ajoute un élément à la fin (appelle `add`)

`list.addFirst(element)` ajoute un élément au début

- `list.add("hello")` // sur une liste où l'opération est supportée

`list.remove(element)` supprime le premier élément (en utilisant `element.equals()`)

- `list.remove("hello")` // sur une liste où l'opération est supportée

`list.remove(index)` supprime à l'index et renvoie l'élément

`list.removeFirst()` supprime le premier élément et renvoie l'élément

`list.removeLast()` supprime le dernier élément et renvoie l'élément

- `list.removeFirst()` // les éléments sont décalés à gauche, sur une liste où l'opération est supportée
- lèvent une exception `NoSuchElementException` si la liste est vide

## Listes non modifiables

Si la liste est non modifiable, les méthodes `set()`, `add()` et `remove(index)/remove(element)/removeFirst()/removeLast()` lèvent `UnsupportedOperationException`

`List.of()` construit une liste non modifiable

- `List.of("hello").set(0, "list") // lève  
UnsupportedOperationException`
- `List.of("hello").add("list") // lève  
UnsupportedOperationException`
- `List.of("hello").remove("hello") // lève  
UnsupportedOperationException`
- `List.of("hello").removeFirst() // lève  
UnsupportedOperationException`

## conversion liste modifiable / non modifiable

### `List.copyOf(collection)`

- Prend une collection en paramètre et copie tous les éléments dans une `List` non modifiable (la même que `List.of()`)

### `new ArrayList<>(collection)`

- Prend une collection en paramètre et copie tous les éléments dans une `ArrayList` (modifiable)

`contains()` / `indexOf()` / `lastIndexOf()`

`list.contains(element)` renvoie si l'élément est contenu (en utilisant `element.equals()`)

- `List.of("foo", "bar").contains("bar") // true`

`list.indexOf(element)` renvoie l'index du premier élément égal (en utilisant `element.equals()`) ou -1

- `List.of("foo", "foo").indexOf("foo") // 0`

`list.lastIndexOf(element)` renvoie l'index du dernier élément égal (en utilisant `element.equals()`) ou -1

- `List.of("foo", "foo").lastIndexOf("foo") // 1`

## Lenteur de contains/remove/indexOf/lastIndex

Toutes ces méthodes ont une complexité en  $O(n)$ , elles nécessitent dans le pire cas de parcourir tous les éléments.

Il peut être plus efficace d'utiliser un dictionnaire (Map) pour avoir une recherche plus rapide.

`java.util.Map`

`java.util.Map`

## Map.of(), HashMap, LinkedHashMap

Il y a trois implantations principales de Map

- Map.of() pour les Map non modifiables

```
Map.of("monday", 1, "tuesday", 2)
```

- HashMap pour les Map modifiables, extensibles et **sans ordre**

```
var map = new HashMap<String, String>();  
map.put("dog", "sad");  
map.put("cat", "happy");  
IO.println(map); // {cat=happy, dog=sad}
```

- LinkedHashMap pour les Map modifiables, extensibles et conserve l'ordre d'insertion

```
var map = new LinkedHashMap<String, Person>();
```

## Map.of()

Map non modifiable qui n'accepte pas null, ni en tant que clé, ni en tant que valeur

- `Map.of(key0, value0, key1, value1, ...)`

```
var empty = Map.<String, Person>of();  
var two = Map.of("key", "value", "key2", "value2");
```

isEmpty /size / getOrDefault

`map.isEmpty()` indique si la Map est vide

```
Map.of().isEmpty() // true
```

`map.size()` renvoie le nombre de couples

```
Map.of("dog", 10, "cat", 8).size() // 2
```

`map.getOrDefault(key, default)` renvoie la valeur associée à key ou la valeur défaut sinon

```
var map = Map.of("dog", 10, "cat", 8);  
map.getOrDefault("dog", 0) // 10  
map.getOrDefault("alien", 0) // 0
```

## map.get() versus map.getOrDefault()

map.get(key) est équivalent à map.getOrDefault(key, null)

```
var map = Map.of("dog", 10);  
map.get("dog") // 10  
map.get("alien") // null
```

Attention à map.get() car si on ne fait pas de test à null derrière, le code va planter avec une NullPointerException.

## map.containsKey() vs map.getOrDefault()

Il existe une méthode `containsKey(key)` de Map qui permet de savoir si la Map contient une clé.

Attention à ne pas écrire

```
var map = Map.of("dog", 10);
if (map.containsKey("dog")) {
    var value = map.getOrDefault("dog", null);
    ...           // ou map.get("dog")
}
```

car cela fait deux recherches dans la Map, alors que le code suivant fait une seule recherche (si on ne stocke pas null en tant que valeur)

```
var map = Map.of("dog", 10);
var value = map.getOrDefault("dog", null);
if (value == null) {
    ...
}
```

toString() / equals() / hashCode()

map.toString() renvoie une version textuelle

```
Map.of("dog", 3, "cat", 3).toString() // {cat=3, dog=3}
```

map.equals() permet de comparer des Map même si ce n'est pas la même implantation

```
var map = Map.of("fantastic", 4);  
map.equals(map) // true
```

map.hashCode() renvoie une valeur résumée du contenu de la map

```
Map.of("fantastic", 4).hashCode() // -16758988
```

## Parcours d'une Map

On ne peut pas faire une boucle sur une Map directement, mais on peut demander

- `map.keySet()` : l'ensemble des clés
- `map.entrySet()` : l'ensemble des couples clé/valeur
- `map.values()` : la collection des valeurs

Exemple de `map.keySet()`

```
var map = Map.of("bus", "yellow", "car", "red");
for(var key: map.keySet()) {
    IO.println(key);
    // bus
    // car
}
```

# Parcours des clés et des valeurs !

Exemple de `map.entrySet()`

```
var map = Map.of("bus", "yellow", "car", "red");

for(var entry: map.entrySet()) {
    // entry est de type Map.Entry<K,V>
    var key = entry.getKey();
    var value = entry.getValue();
    IO.println(key + " " + value);
    // bus yellow
    // car red
}
```

## equals() et hashCode() pour les clés

Toutes les méthodes sauf isEmpty() et size() nécessitent que les clés de la Map implantent les méthodes equals() et hashCode().

Exemple de code faux

```
class Author {
    private final String name;
    public Author(String name) { this.name = name; }
}
...
var map = new HashMap<Author, String>();
map.put(new Author("Edgar Alan Poe"), "dark");
map.getOrDefault(new Author("Edgar Alan Poe"), null)
    // null
```

Il faut jouter les méthodes equals() et hashCode() dans Author pour résoudre le problème.

# java.util.HashMap

Table de hachage modifiable et extensible

- l'ordre des couples n'est pas défini

On crée une table de hachage vide puis on ajoute les couples

```
var map = new HashMap<String, String>();  
map.put("bus", "yellow");  
map.put("car", "red");
```

# map.put()

`map.put(key, value)` ajoute un couple clé valeur

```
var map = new HashMap<String, Person>();  
map.put("Ana", new Person("Ana", 32));  
map.put("Bob", new Person("Bob", 18));  
map.size() // 2
```

avec une clé existante, la valeur est remplacée

```
map.put("Ana", new Person("Ana", 33));  
map.size() // 2
```

# map.remove()

## map.remove(key)

- supprime le couple clé/valeur correspondant à la clé et renvoie la valeur.
- ne fait rien si la clé n'est pas présente et renvoie null.

```
var map = new HashMap<String, String>();  
map.put("car", "red");  
map.remove("car");  
map.isEmpty() // true
```

# LinkedHashMap

Table de hachage modifiable et extensible

- l'ordre d'insertion des éléments est sauvegardé dans une liste chaînée

On crée une table de hachage vide puis on ajoute des couples

```
var map = new LinkedHashMap<String, String>();  
map.put("bus", "yellow");  
map.put("car", "red");
```

Itérateurs

# Itérateurs

# java.util.Iterator

A part les List, les autres collections ne sont pas indexées.

- Comment faire pour les parcourir ?

Un itérateur est un objet que l'on crée sur un List/Set/Deque pour le parcourir.

- Pour créer un Iterator, on utilise la méthode `iterator()` (qui crée un nouvel itérateur à chaque appel)

## Exemple d'utilisation

Un Iterator possède deux méthodes principales

- `hasNext()` qui renvoie vrai si il y a un élément suivant
- `next()` qui renvoie l'élément et mute l'itérateur pour le décaler vers le prochain élément

```
var list = List.of("hello", "iterator");
var iterator = list.iterator(); // crée un nouvel itérateur
while(iterator.hasNext()) { // tant qu'il reste des éléments
    var element = iterator.next(); // obtenir l'élément et
                                // passer au suivant

    IO.println(element);
    // hello
    // iterator
}
```

## Le for(:)

Le `for(:)` sur une collection est transformé en un `while` sur l'itérateur par le compilateur

```
var list = List.of("hello", "iterator");  
for(var element: list) {  
    ...  
}
```

est équivalent à

```
var iterator = list.iterator();  
while(iterator.hasNext()) {  
    var element = iterator.next();  
    ...  
}
```

## Parcours et mutation

Si l'on modifie la structure de données lors du parcours, l'itérateur pourrait pointer sur quelque chose qui n'existe plus

```
var list = new ArrayList<>(List.of(1, 2, 4, 5));
for(var value: list) { // ConcurrentModificationException
    if (value % 2 == 0) {
        list.remove(value);
    }
}
```

Il n'est pas possible en Java de changer/muter une collection pendant son parcours avec un itérateur.

- la méthode `iterator.next()` lève une `ConcurrentModificationException`

## Iterator.remove()

L'interface `Iterator` possède une méthode `remove()` qui supprime l'élément renvoyé par l'appel à `next()`.

Utiliser `iterator.remove()` est sûr.

```
var list = new ArrayList<>(List.of(1, 2, 4, 5));
var iterator = list.iterator();
while(iterator.hasNext()) {
    var value = iterator.next(); // OK
    if (value % 2 == 0) {
        iterator.remove();
    }
}
```

Collection et maintenance

# Collection et maintenance

# Méthode publique et interface

Une méthode publique ne doit pas utiliser une classe d'implantation mais l'interface correspondante.

Mauvais code

```
public class Foo {  
    public ArrayList<String> hello(HashMap<String, String> map) {  
        ...  
    }  
}
```

Problèmes de maintenance

- on ne peut pas appeler `hello()` avec une autre `Map`
- on ne peut pas changer l'implantation de `hello()` pour renvoyer une autre liste

# Méthode publique et interface

Si on utilise à la place des interfaces

```
public class Foo {  
    public List<String> hello(Map<String, String> map) {  
        ...  
    }  
}
```

Plus de problèmes de maintenance

- on peut appeler `hello()` avec une autre `Map`
- on peut changer l'implantation de `hello()` pour renvoyer une autre implantation de liste

# Méthode qui renvoie une Collection

Appeler des méthodes qui modifient (add, remove, etc) une Collection renvoyée par une méthode d'une librairie est **dangereux**.

- Une future version de la librairie peut renvoyer une implantation non modifiable.
- La javadoc de la méthode devrait indiquer si collection renvoyée est modifiable ou pas, mais c'est rarement le cas en pratique !

Autres collections

Autres collections

# Deque

Deque (double ended queue) représente à la fois les piles et les files

- L'implantation est `ArrayDeque`
  - comme pile
    - `isEmpty()`, `push(element)`, `pop()` et `peek()`
  - comme file
    - `addFirst(element)`, `addLast(element)`, `pollFirst()`, `pollLast()`, `peekFirst()`, `peekLast()`
- Les deque implémentent l'interface `SequencedCollection`.

# Set

Set représente les ensembles sans doublon

- Implantations : `Set.of()`, `HashSet()` et `LinkedHashSet()`

`add(element)` renvoie `false` si l'élément passé en paramètre est déjà présent.

`LinkedHashSet` implémente `SequencedCollection`.

Il existe aussi des `SortedSet` qui implémentent `SequencedCollection`.

# Collection Legacy

## Anciennes collections

- `Vector<E>`, remplacée par `ArrayList<E>`
- `Stack<E>`, remplacée par `ArrayDeque<E>`
- `Hashtable<K,V>`, remplacée par `HashMap<K,V>`

Donc remplacées en 1998, car trop lentes pas utilisées sauf sur d'anciennes pages Web :)

Les boîtes (wrappers)

# Les boîtes (wrappers)

# Collection et type primitif

Le type `List<int>` est interdit

- les types primitifs ne sont pas des objets
- et les types paramétrés, `List<TypeArgument>`, ne marchent qu'avec des objets

Pour résoudre ce problème

- Le package `java.lang` possède des classes immutables prédéfinies qui permettent de stocker un type primitif dans un objet

## Exemple de java.lang.Integer

La classe java.lang.Integer est définie à peu près comme cela

```
package java.lang;

public class Integer {
    private final int value;
    ...
    public static Integer valueOf(int value) {
        return new Integer(value);
    }
    public int intValue() {
        return value;
    }
    // + equals, hashCode() et toString()
}
```

## Conversion int <-> Integer

java.lang.Integer sert de wrapper au type primitif int

```
int i = ...  
Integer wrapper = Integer.valueOf(i);  
Integer wrapper = ...  
int i = wrapper.intValue();
```

Conversion automatique : auto-boxing, auto-unboxing

```
int i = ...  
Integer wrapper = i; // boxing, appelle Integer.valueOf()
```

```
Integer wrapper = ...  
int i = wrapper; // unboxing, appelle wrapper.intValue()
```

## Une liste d'entiers

List<int> n'est pas un type valide, on va donc utiliser List<Integer>

```
List<Integer> list = List.of(1, 2, 3); // boxing
```

Et pour obtenir un élément

```
int value = list.get(1); // unboxing
```

Attention à null!

```
var list = new ArrayList<Integer>();  
list.add(null); // ahhh, mais ça marche  
int value = list.getFirst(); // NullPointerException
```

# Les Wrappers

Tous les types primitifs ont un Wrapper

primitif	wrapper	boxing	unboxing
boolean	Boolean	Boolean.valueOf(value)	wrapper.booleanValue()
byte	Byte	Byte.valueOf(value)	wrapper.byteValue()
char	Character	Character.valueOf(value)	wrapper.charValue()
short	Short	Short.valueOf(value)	wrapper.shortValue()
int	Integer	Integer.valueOf(value)	wrapper.intValue()
long	Long	Long.valueOf(value)	wrapper.longValue()
float	Float	Float.valueOf(value)	wrapper.floatValue()
double	Double	Double.valueOf(value)	wrapper.doubleValue()

# Mauvaises utilisations

Les wrappers ne doivent pas être utilisés avec == et !=.  
Comme ce sont des objets, == teste les références.

```
Integer.valueOf(2_000) == Integer.valueOf(2_000) // false
```

Ne pas utiliser de wrapper en tant que type de champ (car il accepte aussi null)

```
public class Car {  
    private final Long weight; // ahhhh, devrait être "long"  
    ...  
}
```

En résumé

En résumé

# L'API des collections

`java.util.List` représente les listes indexées

- `List.of(element, ...)` est non modifiable et non nullable
- `new ArrayList<E>()` grandit dynamiquement

`java.util.Map` représente les dictionnaires

- `Map.of(clé, valeur, ...)` est non modifiable et non nullable
- `new HashMap<K,V>()` est une table de hachage dynamique sans ordre
- `new LinkedHashMap<K,V>()` est une table de hachage qui conserve l'ordre d'insertion

Dans les "`< >`", on utilise les wrappers à la place des types primitifs (Integer à la place de int)