Collection

Rémi Forax

Battery Included

Il y a deux philosophies pour les librairies inclues avec un langage

- “Minimaliste”, comme en C,
  une seule librairie (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

Collection API

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

- Nommée, avec une **classe public** accessible
- Anonyme, accessible à travers une **méthode public static**

List et Map

Les deux interfaces les plus utilisées sont

- List, qui définit une liste d'élements
  - 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 modfiables

java.util.List

List.of(), Arrays.asList(), ArrayList

Il y a 3 implantations principales

- List.of() pour les listes non modifiables

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

- Arrays.asList() qui voit un tableau comme une liste

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

- 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()/size()/get()

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

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

list.**size**() renvoie le nombre d'élements

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

list.**get**(index) renvoie le n-ième élement (à partir de 0)

```
List.of("hello", "list").get(0)  // hello
```

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

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);
list.equals(list)  // 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) {
  System.out.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, cf plus tard)

java.util.Arrays.asList()

Une liste qui “wrap” un tableau

- La liste a une taille fixe (celle du tableau)
- Le contenu de la liste est modifiable

  Si on modifie la liste, le tableau est modifié et vice-versa

  Arrays.**asList**(E...)

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

  Arrray.**asList**() avec un tableau d’objet, ne copie pas les valeurs

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

  Array.**asList**() accepte **null**

List.set()

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

```
var list = Arrays.asList("hello");
list.set(0, "bonjour");
list.get(0)  // bonjour
```

Arrays.asList() est une vue

```
var array = new String[] { "hello" };
var list = Arrays.asList(array);
list.set(0, "bonjour");
array[0]  // bonjour
```

java.util.ArrayList

Liste modifiable et extensible dynamiquement

- La liste s'agrandit toute seule
- Le contenu de la liste est modifiable

On créé une liste vide et on ajoute les élements

```
var list = new ArrayList<String>();  // on doit indiquer le type
list.add("hello");
list.add("list");
```

ArrayList accepte **null**

```
list.add(null);   // ok
```

List.add()/List.remove()

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

```
list.add("hello")
```

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

```
list.remove("hello")
```

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

```
list.remove(0)  // les éléments sont décalés à gauche
```

Liste non modifiable

set(), add() et remove() sont des opérations optionnelles, elles peuvent lever UnsupportedOperationException

- List.of() est non modifiable
  - List.of("hello").set(0, "list") // lève UOE
  - List.of("hello").add("list") // lève UOE
  - List.of("hello").remove("hello") // lève UOE
- Arrays.asList() a une taille fixe
  - Arrays.asList("hello").add("list") // lève UOE
  - Arrays.asList("hello").remove("hello") // lève UOE

Non modifiable <-> Modifiable

List.**copyOf**(collection)

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

new ArrayList<>(collection)

Prend une collection en paramètre et copie tous les élements dans une ArrayList (modifiable)

contains()/indexOf()/lastIndexOf()

list.**contains**(element) renvoie si l'élement 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 au pire cas de parcourir tous les élements

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

java.util.Map

Map.of(), HashMap, LinkedHashMap

Il y a 3 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");
```

- LinkedHashMap pour les Map modifiables, extensibles et utilisant l’ordre d’insertion

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

Map.of() / Map.ofEntries()

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

Map.**of**(key, value, key, value, ...)

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

Map.**ofEntries**(Map.Entry<K,V>... entries)

```
var two = Map.ofEntries(
  Map.entry("key", "value"),
  Map.entry("key", "value")
);
```

La méthode statique Map.entry() prend une clé et une valeur et renvoie un couple clé/valeur typé Map.Entry (Entry est une interface rangée dans Map)

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, defaut) 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() vs Map.getOrDefault()

map.**get**(clé) est équivalent à
map.getOrDefault(clé, **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 NPE

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 Maps 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
```

keySet(), entrySet(), values()

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()) {
  System.out.println(key);
} // bus
  // car
```

keySet(), entrySet(), values()

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();
  System.out.println(key + " " + value);
} // bus yellow
  // car red
```

Exemple de Map.values()

```
var map = Map.of("bus", "yellow", "car", "red");
for(var value: map.values()) {
  System.out.println(value);
} // yellow
  // red
```

Key.equals()/Key.hashCode()

Toutes les méthodes sauf isEmpty() et size() nécesitent 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
```

ajouter les méthodes equals() et hashCode() dans Author résout 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**(clé, valeur) 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**(clé) supprime le couple clé/valeur correspondant à la clé

```
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 élements est sauvegardé dans une liste chainée

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

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

Map non modifiable

put() et remove() sont des opérations optionnelles, elles peuvent lever UnsupportedOperationException

Map.of() est non modifiable

- Map.of("hello", 5).put("list", 4) // lève UOE
- Map.of("hello", 5).remove("hello") // lève UOE

Non Modifiable <-> Modifiable

Map.**copyOf**(map)

Prend une map en paramètre et copie tous les élements dans une Map non modifiable (la même que Map.of())

new HashMap<>(map)
new LinkedHashMap<>(map)

Prend une map en paramètre et copie tous les élements dans une HashMap/LinkedHashMap (modifiable)

Itérateur

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 créé sur un List/Set/Deque pour le parcourir

- Pour créer un Iterator, on utilise la méthode **iterator**() (qui créé un nouvel iterateur à chaque appel)

Exemple d’utilisation

Un Iterator possède 2 méthodes principales

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

```
var list = List.of("hello", "iterator");
var iterator = list.iterator();   // créé un nouvel itérateur
while(iterator.hasNext()) {   // tant qu’il reste des élements
  var element = it.next();      // obtenir l’élement et
                                // passer au suivant
  System.out.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) {  // créé un itérateur et
                          // boucle dessus
  System.out.println(element);
}
```

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 CME

Iterator.remove()

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

Utiliser iterator.remove() est sûre

```
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

Methode publique et interface

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

```
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

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

Autres collections

Deque et Set

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()

Set représente les ensembles sans doublon

Implantations: Set.of(), HashSet() et LinkedHashSet(), équivalent à Map<E, Boolean>

Collection *Legacy*

Anciennes collections

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

Remplacées en 1998, car trop lentes

pas utilisées sauf sur d’anciennes pages Web :)

Les boites (*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;
  }
}
```

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 élement

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

Attention à **null** !

```
int value = Arrays.asList(1, null, 3).get(1);  // NPE
```

Les Wrappers

Tous les types primitifs, on 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 se 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

L’API des collections

java.util.List représente les listes indexées

- List.of(element, ...) est non modifiable et non nullable
- Arrays.asList(array) a une taille fixe
- **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)