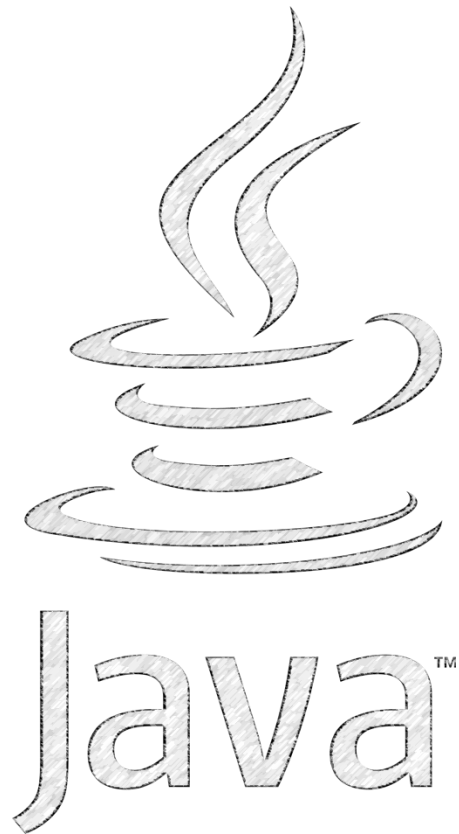


Programmation Orientée Objets avec Java

Prof. Yassin Rekik et Prof. Stéphane Malandain



Chapitre 8

Interfaces fonctionnelles
Programmation déclarative
Fonction d'ordre supérieur

- Interfaces fonctionnelles
- Notation flèche
- Programmation déclarative
- Fonction d'ordre supérieur

Les interfaces fonctionnelles

- Formellement, une interface fonctionnelle est une interface ne présentant qu'une seule méthode abstraite.
- Des instances d'interface fonctionnelles communément employées peuvent être obtenues à partir du package `java.util.functions`.
- Illustrons ce concept au travers d'un exemple utilisant une interface fonctionnelle commune : le Prédicat.

Un exemple : le prédicat

5

- Nous définissons une liste d'utilisateurs en utilisant une classe `Person`.
- Notre but sera d'extraire une liste d'utilisateurs étant majeurs à partir de la liste complète d'utilisateurs.

```
public class Person {  
    private String name;  
    private int age;  
}
```

Un exemple : le prédicat

6

```
List<Person> userList = new ArrayList<>();  
userList.add(new Person("Charles Quint", 15));  
userList.add(new Person("Luc Martin", 18));  
userList.add(new Person("Marc Durand", 22));
```

```
List<Person> adultList = new ArrayList<>();
```

```
for(Person user: userList) {  
    if(user.getAge() >= 18) {  
        adultList.add(user);  
    }  
}
```

```
// adultList vaut [Person("Luc Martin", 18), Person ("Marc Durand", 22)]
```

Un exemple : le prédicat

7

Le code précédent présente quelques problèmes :

- Il est peu élégant
- Il faut décomposer la boucle pour comprendre ce qu'elle fait
- Dans ce cas précis, si on voulait modifier la collection sur laquelle on itère, cela amène de nombreux problèmes

Malgré le fait que ce code produit le résultat escompté dans une seconde liste, il ne respecte pas les principes de **programmation déclarative**.

Il est possible de faire mieux en utilisant une interface fonctionnelle

Un exemple : le prédicat

8

default boolean

```
removeIf(Predicate<? super E> filter)
```

Removes all of the elements of this collection that satisfy the given predicate.

La méthode `removeIf` des collections (JavaDoc)

- La méthode `removeIf` nous permet de retirer des éléments d'une liste en filtrant les éléments qui répondent à une condition.
- Cette condition peut être variabilisée dans ce qu'on appelle un `Prédicat`, qui est l'une des interfaces fonctionnelles les plus communes.

Un exemple : le prédicat

9

Le code précédent peut être réduit à :

```
userList.removeIf( CONDITION )
```

Puisque la méthode `removeIf` modifie la liste en retirant les éléments répondant `true` à `CONDITION`

Un exemple : le prédicat

10

La déclaration d'un prédicat peut se faire comme suit :

```
Predicate<Integer> isAChild = i -> i < 18;
```

La variable `isAChild` contient alors la condition que nous aurions mise dans un `if`.

Au moment de la définition du `prédicat`, la valeur à tester n'est pas encore connue, elle est désignée ici par `i`

Un exemple : le prédicat

11

Un `Integer` arbitraire peut être testé en utilisant la méthode `test` du prédicat :

```
Predicate<Integer> isAChild = i -> i < 18;  
  
resultOne = isAChild.test(10)  
// True  
  
resultOne = isAChild.test(20)  
// False
```

Un exemple : le prédicat

12

Le code précédent peut être remplacé par :

```
Predicate<Person> isAChild = p -> p.getAge() < 18;  
  
userList.removeIf(isAChild);  
  
// userList vaut [Person("Luc Martin", 18), Person("Marc Durand", 22)]
```

Généralisation et autre
interfaces fonctionnelles communes

Si l'on raisonne sur le type, le prédicat est en fait une fonction qui prend un type donné et qui retourne un booléen. Autrement dit, une fonction avec la signature :

```
Predicate<T> T -> boolean;
```

Le prédicat peut être employé en utilisant la méthode `test` tel que :

```
Boolean test(T t)
```

Certaines méthodes comme `removeIf` prennent directement un `Predicate` en paramètre.

Généralisation des interfaces fonctionnelles

15

Voici les 4 interfaces fonctionnelles les plus communes à connaître :

java.util.function	signature	utilisation
Predicate<T>	T -> boolean	boolean test(T t)
Consumer<T>	T -> ()	void accept(T t)
Supplier<T>	() -> T	T get()
Function<T,R>	T -> R	R apply(T t)

- La définition des interfaces fonctionnelles utilise un sucre syntaxique appelé `expression lambda`. L'implémentation derrière cela sera décrite plus loin.
- Leur utilisation permet un code plus court, plus “propre” et souvent plus lisible.
- De plus, certaines méthodes comme `removeIf` n'acceptent qu'une interface fonctionnelle précise.

Implémentations concrètes

Le supplier

18

Supplier

@FunctionalInterface

```
public interface Supplier<T> {  
    T get();  
}
```

Le `supplier` est une interface fonctionnelle simple représentant une opération qui fournit une valeur à chaque appel.

Exemple :

```
Supplier<String> s = () -> myFormatFunction(LocalDate.now());  
String currentTime = s.get()
```

Exemple 2 : un nombre aléatoire à chaque appel

```
public class SupplierEx1 {  
  
    public static void main(String[] args) {  
  
        Supplier<Double> random = () -> Math.random();  
  
        System.out.println("Random value: " + random.get());  
        System.out.println("Random value: " + random.get());  
        System.out.println("Random value: " + random.get());  
    }  
}
```

Le consumer

20

Consumer

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> {
            accept(t);
            after.accept(t);
        };
    }
}
```

Le consumer est une interface fonctionnelle représentant une opération qui accepte un paramètre d'entrée et ne renvoie rien.

Exemple :

```
Consumer<Person> printName = p -> System.out.println(p.getName());
userList.forEach(printName)
```

```
public class ConsumerEx1 {  
  
    public static void main(String[] args) {  
  
        // Create a Consumer object directly  
        Consumer<String> greeter = name -> System.out.println("Hello " + name);  
  
        greeter.accept("Tran"); // Hello Tran  
    }  
}
```

La fonction

22

Function interface

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }

    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

Function est une interface fonctionnelle représentant un opérateur qui accepte une valeur d'entrée et en renvoie une autre.

Exemple :

```
Function<Integer, Double> half = a -> a / 2.0;
result = half.apply(10);
```

La fonction

23

```
import java.util.function.Function;

public class FunctionEx1 {

    public static void main(String[] args) {

        Function<String, Integer> func = (text) -> text.length();

        int length = func.apply("Function interface tutorial");

        System.out.println("Length: " + length);
    }
}
```

La fonction

24

```
public class FunctionEx2 {  
  
    public static void main(String[] args) {  
  
        Function<String, String> func = text -> text.toUpperCase();  
  
        List<String> list = Arrays.asList("Java", "C#", "Python");  
  
        List<String> newList = map(func, list);  
  
        newList.forEach(System.out::println);  
    }  
  
    public static <T,R> List<R> map(Function<T,R> mapper, List<T> list) {  
        List<R> result = new ArrayList<R>();  
  
        for(T t: list) {  
            R r = mapper.apply(t);  
            result.add(r);  
        }  
        return result;  
    }  
}
```


Notation fléchée

- Nous avons vu les interfaces fonctionnelles en employant une notation appelée `expression lambda`.
- Cette expression fléchée permet l'expression des `lambdas` en allégeant considérablement la syntaxe.
- Les opérations effectuées derrière peuvent tout à fait être faites 'à la main'.
- Pour rappel, la définition d'une interface fonctionnelle est une interface avec une seule méthode abstraite.

Interface fonctionnelle: Une implémentation complète

27

```
1  public interface HelloInterface {
2      public void HelloToYou(String s);
3  }
4
5  public class HelloImplementation implements HelloInterface {
6
7      public HelloImplementation() { }
8
9      public void HelloToYou(String s) {
10         System.out.println("Hello to " + s);
11     }
12 }
13
14 HelloInterface h = new HelloImplementation();
15 h.HelloToYou("Charles !");
```

Interface fonctionnelle: avec une classe anonyme

28

```
1  @FunctionalInterface
2  public interface Hello {
3      public void HelloToYou(String s);
4  }
5
6  Hello h = new Hello() {
7      public void HelloToYou(String s) {
8          System.out.println("Hello to " + s);
9      }
10 }
11
12 h.HelloToYou("Encore Charles !");
13
```

Interface fonctionnelle: avec une lambda

29

```
1  @FunctionalInterface
2  public interface Hello {
3      public void HelloToYou(String s);
4  }
5
6  Hello h = (String s) -> System.out.println("Hello to " + s);
7  h.HelloToYou("Et toujours Charles !");
8
```

- L'exemple précédent n'est en fait rien d'autre qu'un `Consumer` fait "à la main".
- Les interfaces connues telles que par exemple le `Consumer` respectent cette implémentation.
- Elles sont simplement là pour nous simplifier la vie en évitant la déclaration de l'interface dans le code.
- En utilisant un `Consumer`, nous évitons de déclarer l'interface `Hello`.

```
Consumer<String> helloToYou = s -> System.out.println("Hello to " + s);
```

Il est possible de s'affranchir de l'utilisation de variables pour les interfaces fonctionnelles en utilisant les expressions `lambda` :

```
// Predicate<Person> isAChild = (Person p) -> p.getAge() < 18;
```

```
userList.removeIf(p -> p.getAge() < 18);
```

Il est important de raisonner sur les types pour déterminer de quelle interface il s'agit.

Ces trois notations sont équivalentes :

```
List<Integer> numberList = ...
```

```
numberList.forEach((Integer i) -> System.out.println(i));
```

```
numberList.forEach(i -> System.out.println(i));
```

```
numberList.forEach(System.out::println); // Référence de la méthode  
// Généralisable en classe-name::méthode
```



```
List<Person> people = ...  
  
people.filter( p -> p.isWorking() )  
    .map( p -> p.getEmail() )  
    .forEach( email -> sendEmail("Good work buddy!", email));
```

- La notation fléchée est un sucre syntaxique
- En réalité, quand le compilateur rencontre cette notation :
 - Crée une classe anonyme avec une unique méthode
 - Cette classe implémente une interface fonctionnelle connue
 - Le comportement de sa méthode est décrit par l'expression `lambda`

Notation fléchée : Bonnes pratiques (1/2)

35

- Les lambdas doivent rester simples et self-explanatory
- Un de leur atout principal : le compromis entre utilité et lisibilité
- Eviter les `()`, `{}` et `return` inutiles
- Eviter de spécifier les types quand ce n'est pas ambigu

```
(String a, String b) -> a.toLowerCase() + b.toLowerCase();  
// Pourrait être  
(a, b) -> a.toLowerCase() + b.toLowerCase();
```

Notation fléchée : bonnes pratiques (2/2)

36

- Evitez de faire des blocs de code importants dans les lambdas

```
Foo foo = parameter -> {  
    int a,b,c,d,e,f,g,h,i,j,k,l.....  
    // Code complexe de plusieurs lignes ...  
    return result;  
};
```

// Pourrait être

```
Foo foo = MaClasse::MaFonctionComplexe;
```

- Ce n'est pas parce que l'on peut faire tout et n'importe quoi avec une lambda qu'il faut le faire !
- Utilisez des références de méthodes quand cela rend le code plus lisible

```
a -> a.toLowerCase();  
// Pourrait être  
String::toLowerCase;
```

Programmation déclarative

Les paradigmes : impératif vs. déclaratif

- Nous avons une chaîne de caractère et nous souhaitons :
 - Remplacer les occurrences du caractère '_' par un espace
 - Supprimer les espaces en début et fin de chaîne
 - Passer toute la chaîne en minuscule

Exemple : approche impérative

40

```
private static String maFonctionImperative(String input) {
    StringBuilder out_builder = new StringBuilder();
    boolean start = true;
    int endOfTrailingSpaces = input.length() - 1;
    while (input.charAt(endOfTrailingSpaces) == ' '
        && endOfTrailingSpaces-- > 0);

    for (int i = 0; i <= endOfTrailingSpaces; i++) {
        char c = input.charAt(i);
        if (c == ' ') {
            if (start) continue;
            out_builder.append(c);
        } else if (c == '_') {
            out_builder.append(' ');
        } else if (c >= 'A' && c <= 'Z') {
            out_builder.append((char) (c + 32));
        } else {
            out_builder.append(c);
        }
        start = false;
    }
    return out_builder.toString();
}
```


Exemple : approche déclarative

41

```
private static String maFonctionDeclarative(String input) {  
    return input.replace('_', ' ') // 1  
                .trim() // 2  
                .toLowerCase(); // 3  
}
```

- La programmation impérative décrit comment faire les choses
 - Par exemple, pour dessiner : Commence par dessiner 4 pattes, puis le corps,
- La programmation déclarative décrit les choses que l'on veut faire
 - Par exemple, pour dessiner : Dessine un mouton.

- Avantages
 - Le code est en général plus court
 - Le code est plus intelligible (lire le code permet de comprendre ce que l'on veut faire)
 - Le code est bien plus simple à optimiser (grâce à l'abstraction)
- Dé-avantages
 - Il est difficile d'optimiser le code par rapport à ses caractéristiques propres (à cause de l'abstraction)
 - Parfois un peu complexe à comprendre pour les non-initiés
 - L'apprentissage de la programmation commence généralement par de la programmation impérative et le changement de paradigme peut parfois être compliqué.

Les fonctions d'ordre supérieur (HOF)

Qu'est-ce que c'est ?

45

Une fonction d'ordre supérieur (Higher-Order Function) est une fonction qui prend en paramètre et/ou retourne une(des) autre(s) fonction(s).

En Java, nous n'allons pas pouvoir travailler avec des fonctions directement. Nous nous servirons plutôt d'interfaces fonctionnelles.

Exemple de fonction d'ordre supérieur en Java:

```
Predicate<Integer> getTester(Supplier<Integer> generator);
```

HOF: Les piliers de la programmation déclarative en java

La méthode `forEach` permet d'appliquer un `Consumer` sur chacun des éléments d'une collection.

Sa signature est la suivante :

```
<T> void forEach(Collection<T> collec,  
                  Consumer<T> action);
```

Soit la collection d'entiers suivante :

```
Collection<Integer> collec = List.of(0, 1, 2, 3, 4, 5);
```

On souhaite afficher ses éléments, avec un élément par ligne :

```
forEach(collec, System.out::println);
```

La fonction `System.out::println` est un `Consumer<Integer>` qui prend un entier en paramètre et qui ne retourne rien.

forEach - implémentation

49

Une implémentation possible serait :

```
private static <T> void forEach(Collection<T> collec,  
                                Consumer<T> action) {  
    for (T element : collec) {  
        action.accept(element);  
    }  
}
```

La fonction `map` permet de transformer une collection en appliquant une fonction unaire sur chacun des éléments de la collection.

```
<T, U> Collection<U> map(Collection<T> collec,  
| | | | | | Function<T, U> mapper);
```

Je souhaite appliquer une fonction $f(x) = (x/2) + 1$ à ma collection. Avec `map`, je peux faire :

```
Collection<Integer> collec = List.of(0, 1, 2, 3, 4, 5);  
Collection<Double> collec2 = map(collec, i -> i / 2.0 + 1.0);
```

On va donc transformer notre collection d'entiers en collection de double grâce à notre `lambda` qui est une `Function<Integer, Double>` qui prend un entier et qui retourne un double.

map - implémentation

51

Une implémentation possible serait :

```
private static <T, U> Collection<U> map(Collection<T> collec,
                                         Function<T, U> mapper) {
    Collection<U> out = new ArrayList<U>();
    for (T element : collec) {
        out.add(mapper.apply(element));
    }
    return out;
}
```

La méthode `reduce` permet de réduire une collection en appliquant itérativement un opérateur binaire.

Sa signature est la suivante :

```
<T> T reduce(Collection<T> collec,  
             T identity,  
             BinaryOperator<T> accumulator);
```

Contrairement aux fonctions précédentes, nous avons un paramètre supplémentaire `T identity`. Il s'agit de l'élément neutre de l'opérateur `accumulator`.

Pour sommer les éléments d'une collection de double, je peux faire :

```
Collection<Double> collec = List.of(1.0, 0.5, 0.25, 0.125, 0.0625);  
Double sum = reduce(collec, 0.0, (lhs, rhs) -> lhs + rhs);  
// ou alors  
sum = reduce(collec, 0.0, Double::sum);
```

Ici, nous utilisons l'opérateur addition de doubles, et son élément neutre 0.

reduce - implémentation

54

Une implémentation possible serait :

```
private static <T> T reduce(Collection<T> collec,  
                             T identity,  
                             BinaryOperator<T> accumulator) {  
    T result = identity;  
    for (T element : collec)  
        result = accumulator.apply(result, element);  
    return result;  
}
```

La méthode `flatMap` permet d'appliquer une fonction sur chaque élément de la collection comme pour le `map`. La spécificité du `flatMap` vient de la fonction à fournir.

La signature est la suivante :

```
<T, U> Collection<U> flatMap(Collection<T> collec,  
| | | | | | | Function<T, Collection<U>> mapper);
```

La fonction `mapper` prend un élément de la collection et retourne une collection. Après avoir appliqué la fonction sur tous les éléments, les collections obtenues sont concaténées.

flatMap - exemple 1/2

56

Prenons le texte sur plusieurs lignes contenu dans la variable `my_text`. Je veux récupérer une collection contenant chaque mot du texte. Une solution possible consiste à séparer le texte en collection de lignes. Ensuite, on peut passer une fonction qui transforme une ligne en collection de mots à notre `flatMap`.

```
String my_text = """
    Bonjour
    Je suis un texte sur
    plusieurs lignes
    Aurevoir et à bientôt""";
Collection<String> lines = Arrays.asList(my_text.split("\n"));
Collection<String> words = flatMap(lines,
    l -> Arrays.asList(l.split(" ")));
```


Décomposons les étapes :

1. `my_text.split("\n")`, transforme le texte en collection, ex : `my_text -> {"bonjour", ... , "Aurevoir et à bientôt"}`.
2. `flatMap` effectue deux actions
 1. `l -> l.split(" ")`, transforme une ligne en collection de mots, ex : `"Je suis un texte sur" -> {"je", "suis", "un", "texte", "sur"}`
 2. Concaténation des collections, on a `{{"bonjour", ..., {"Aurevoir", "et", "à", "bientôt"}}` -> `{"bonjour", ..., "Aurevoir", "et", "à", "bientôt"}`

On peut observer à l'étape 2.2 qu'on passe d'une `Collection<Collection<String>>` à une `Collection<String>`

flatMap - implémentation

58

Une implémentation possible serait :

```
private static <T, U>
    Collection<U> flatMap(Collection<T> collec,
        Function<T, Collection<U>> mapper) {
    Collection<U> out = new ArrayList<U>();
    for (T element : collec) {
        out.addAll(mapper.apply(element));
    }
    return out;
}
```