PROGRAMMATION FONCTIONNELLE

Fabrice Legond-Aubry et Pascal Poizat

Juillet-Août 2024

- Introduction
- Pureté
- Immutabilité
- Ordre supérieur
- Programmes séquentiels
- Absence de valeur et erreurs
- Types de données algébriques
- Effets de bord
- Flux et évaluation paresseuse
- Concurrence
- Test
- Synthèse des notations

PURETÉ

FONCTIONS AU SENS GÉNÉRAL

- entrée(s) -> calcul -> sortie
- **signature** + corps

principe : s'intéresser aux signatures
plus qu'au corps des fonctions

$f : (T_1, ..., T_n) \to T$ (fonction) ou $f : \_ \bullet (T_1, ..., T_n) \to T$ (méthode)

Java

```java
public static int add(int a, int b) { return a+b; }
public static char getFirstCharacter(String s) { return s.charAt(0); }
public static int divide(int a, int b) { return a/b; }
public static void eatSoup(Soup soup) { // TODO: eat soup algorithm }
```

$add : \_ \bullet (Int, Int) \to Int$

$getFirstCharacter : \_ \bullet String \to Char$

$divide : \_ \bullet (Int, Int) \to Int$

$eatSoup : \_ \bullet Soup \to ()$

CONTEXTE OBJET

- méthodes d'instance : un contexte = un objet
- présence d'un type implicite : type du receveur

$C : \ : f : (T_1, ..., T_n) \to T$ donne $f : C \hookrightarrow (T_1, ..., T_n) \to T$

Java

```java
public class Personne {
    private String nom;
    private int age;

    public Personne(String nom, int age) {... }
    public String nom() {... }
    public int age() {... }
    pubilc void vieillir(int annees) {... }
}
```

$nom : Personne \hookrightarrow () \to String$

$age : Personne \hookrightarrow () \to int$

$vieillir : Personne \hookrightarrow int \to ()$

JAVA VS SCALA

en Java

Java

```java
public static int add(int a, int b) { return a + b; }
```

en Scala

Scala

```scala
def add(a: Int, b: Int): Int = { a + b } // ou juste a + b
```

utilisation du REPL Scala

```
➜ scala
Welcome to Scala 3.4.2 (21, Java Java HotSpot(TM) 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> def add(a: Int, b: Int): Int = { a + b }
def add(a: Int, b: Int): Int

scala> add(10, 11)
val res0: Int = 21
```

PROBLÈME(S)

add : _ • (*Int*, *Int*) → *Int*

getFirstCharacter : _ • *String* → *Char*

divide : _ • (*Int*, *Int*) → *Int*

eatSoup : _ • *Soup* → ()

les fonctions peuvent mentir !

```Java
getFirstCharacter(null) = ?
getFirstCharacter("") = ?
divide(10, 0) = ?
eatSoup(null) = ?
```

les fonctions peuvent ne rien retourner !

```Java
eatSoup(someSoup) = ?
```

principe : on veut éviter cela

ETUDE DE CAS

- panier de courses
  - un item (String) peut être ajouté au panier
  - si un livre a été ajouté on a un rabais de 5%
  - on peut connaître les items du panier

Java

```java
public class Panier {
    private List<String> items = new ArrayList<>();
    private boolean bookAdded = false;

    public void ajouter(String item) {
        items.add(item);
        if (item.equals("Livre")) {
            bookAdded = true;
        }
    }
    public int rabais() { return bookAdded ? 5 : 0; }
    public List<String> items() { return items; }
}
```

ETUDE DE CAS : PROBLÈME

Java

```java
public class Panier {
    private List<String> items = new ArrayList<>();
    private boolean bookAdded = false;

    public void ajouter(String item) {
        items.add(item);
        if (item.equals("Livre")) {
            bookAdded = true;
        }
    }
    public int rabais() { return bookAdded ? 5 : 0; }
    public List<String> items() { return items; }
}
```

ajouter : Panier ↪ String → ()

rabais : Panier ↪ () → Int

items : Panier ↪ () → List[String]

remove : List[String] ↪ String → Bool

Java

```java
Panier panier = new Panier();
panier.ajouter("Pomme");
panier.ajouter("Livre");
panier.ajouter("Citron");
panier.items()
      .remove("Livre");
panier.items() = ? / panier.rabais() = ?
```

ETUDE DE CAS : SOLUTION

vous avez probablement repéré le problème :
mauvaise gestion de l'état

principe : passer des copies
plutôt que modifier l'état sur place

Java

```java
public class Panier {
    private List<String> items = new ArrayList<>();
    private boolean bookAdded = false;

    public void ajouter(String item) {
        items.add(item);
        if (item.equals("Livre")) {
            bookAdded = true;
        }
    }
    public int rabais() { return bookAdded ? 5 : 0; }
    public List<String> items() { return new ArrayList<>(items); }
}
```

ETUDE DE CAS (V1.1)

- panier de courses
  - un item (String) peut être ajouté au panier
  - si un livre a été ajouté on a un rabais de 5%
  - on peut connaître les items du panier
  - **un item ajouté au panier peut en être retiré**

Java

```java
public class Panier {
    ...
    public void retirer(String item) {
        items.remove(item);
        if (item.equals("Livre")) {
            bookAdded = false;
        }
    }
    ...
}
```

ETUDE DE CAS (V1.1) : PROBLÈME

Java

```java
public class Panier {
    private List<String> items = new ArrayList<>();
    private boolean bookAdded = false;

    public void ajouter(String item) {
        items.add(item);
        if (item.equals("Livre")) {
            bookAdded = true;
        }
    }
    public void retirer(String item) {
        items.remove(item);
        if (item.equals("Livre")) {
            bookAdded = false;
        }
```

Java

```java
Panier panier = new Panier();
panier.ajouter("Livre");
panier.ajouter("Livre");
panier.retirer("Livre");
panier.items() = ? / panier.rabais() = ?
```

ETUDE DE CAS (V1.1) : SOLUTION

vous avez probablement repéré le problème :
mauvaise utilisation d'un champ calculé

Java

```java
public class Panier {
    private List<String> items = new ArrayList<>();

    public void ajouter(String item) {
        items.add(item);
    }
    public void retirer(String item) {
        items.remove(item);
    }
    public int rabais() { return items.contains("Livre") ? 5 : 0; }
    public List<String> items() { return new ArrayList<>(items); }
}
```

on a échangé la performance
contre la lisibilité et la maintenance

ETUDE DE CAS (V1.1) : ANALYSE

Java

```java
public class Panier {
    private List<String> items = new ArrayList<>();

    public void ajouter(String item) {
        items.add(item);
    }
    public void retirer(String item) {
        items.remove(item);
    }
    public int rabais() { return items.contains("Livre") ? 5 : 0; }
    public List<String> items() { return new ArrayList<>(items); }
}
```

- on a en réalité beaucoup de “boilerplate code” :
  - définition et sécurisation de l’état interne (`items` et `items`)
  - wrappers de modification de l’état interne (`ajouter`, `retirer`)

ETUDE DE CAS (V1.1) : NOUVELLE SOLUTION

on peut supprimer ce “boilerplate code”

Java

```java
public class Panier {
  public static int rabais(List<String> items) {
      return items.contains("Livre") ? 5 : 0;
  }
}
```

qu’en est il des besoins ?

- *un item (String) peut être ajouté au panier*
- **si un livre a été ajouté on a un rabais de 5%**
- *on peut connaître les items du panier*
- *un item ajouté au panier peut en être retiré*

ETUDE DE CAS (V1.1) : SÉPARATION

principe : séparation des besoins
en différentes fonctions éventuellement regroupées

- `String` pour les items (discutable)
- `Panier::rabais` pour le rabais
- API des collections pour l'état (ici)

Java

```java
List<String> items = new ArrayList<>();
items.add("Pomme");
items.add("Livre");
items.add("Livre");
items.remove("Livre");
items = ? / Panier.rabais(items) = ?
```

FONCTIONS IMPURES ET FONCTIONS PURES

style impératif

ajouter : *Panier* ↪ *String* → ()

retirer : *Panier* ↪ *String* → ()

items : *Panier* ↪ () → *List*[*String*]

rabais : *Panier* ↪ () → *Int*

style fonctionnel

add : *List*[*String*] ↪ *String* → *Boolean*

remove : *List*[*String*] ↪ *String* → *Boolean*

iterator : *List*[*String*] ↪ () → *Iterator*[*String*]

rabais : *Panier* • *List*[*String*] → *Int*

fonctions impures

- absence de sortie
- sortie pas f°(entrées)
- mutation d'état

principe : fonctions pures

- sortie, unique
- sortie f°(entrées)
- pas de mutation d'état

EXERCICE : REFACTORER

Java

```java
public class CalculateurPourboire {
    private List<String> noms = new ArrayList<>();
    private int pourcentage = 0;

    public void ajouterPersonne(String nom) {
        noms.add(nom);
        if (noms.size() > 5)      { pourcentage = 20; }
        else if (noms.size() > 0) { pourcentage = 10; }
    }

    public List<String> noms() { return noms; }
    public int pourcentage() { return pourcentage; }
}
```

- objectifs :
  - sortie, unique
  - sortie f°(entrées)
  - pas de mutation d'état
- techniques :
  - stocker -> calculer
  - état en paramètre
  - copies des données

EXERCICE : FONCTIONS PURES ?

Java

```java
static int    f1(String s)     { return s.length() * 3;              }
static char   f2(String s)     { return s.charAt(0);                 }
static double f3(double x)     { return x * Math.random();           }
static int    f4(int x, int y) { return x + y;                       }
static int    f5(int x, int y) { return x / y;                       }
static double f6(int x, int y) { return x / y;                       }
public int    f7(String item)  { items.add(item); return items.size();}
```

RÉSUMÉ

- signatures = quoi ? > corps = comment ?
- $f : (T_1, ..., T_n) \to T$, $f : C \bullet (T_1, ..., T_n) \to T$, $f : C \hookrightarrow (T_1, ..., T_n) \to T$
- fonctions pures
  - sortie, unique
  - sortie f°(entrées)
  - pas de mutation d'état
- qualité du code
  - responsabilité unique
  - pas d'effet de bord
  - déterminisme / transparence référentielle
  - facilité de test (on reviendra sur cela)