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

IMMUTABILITÉ

IDÉE GÉNÉRALE

La programmation fonctionnelle
consiste à programmer
en utilisant des **fonctions pures**
qui manipulent des **valeurs immuables**

ETUDE DE CAS

itinéraire de voyage, Paris à Nantes

Java

```java
List<String> planA = new ArrayList<>();
planA.add("Paris");
planA.add("Nantes");
// planA = [Paris, Nantes]
```

on veut modifier le voyage, Rennes avant Nantes

replanifie : _ • (*List*[*String*], *String*, *String*) → *List*[*String*]

Java

```java
List<String> planB = replanifie(planA, "Rennes", "Nantes");
// planB = [Paris, Rennes, Nantes]
```

Java

```java
static List<String> replanifie(List<String> plan, String nouvelleVille,
        String avantVille) {
    int index = plan.indexOf(avantVille);
    plan.add(index, nouvelleVille);
    return plan;
}
```

ETUDE DE CAS : PROBLÈME

```java
// planA = [Paris, Nantes]
List<String> planB = replanifie(planA, "Rennes", "Nantes");
// planB = ?
// planA = ?
```

replanifie : _ • (*List*[*String*], *String*, *String*) → *List*[*String*] ment !

la fonction a muté une variable, elle n'est pas pure

principe : éviter la mutabilité
rappel : fonctions pures

évite se devoir se poser des tas de questions

- la liste retournée : copie ou vue ?
- si c'est une vue, puis-je la modifier ?
- paramètres modifiés par la fonction ?
- puis-je réutiliser la liste paramètre ?

ETUDE DE CAS : SOLUTION

Java

```java
static List<String> replanifie(List<String> plan, String nouvelleVille,
        String avantVille) {
    int index = plan.indexOf(avantVille);
    List<String> copie = new ArrayList<>(plan);
    copie.add(index, nouvelleVille);
    return copie;
}
```

ici la mutation opère sur une copie locale

plus de problème, la fonction est pure

MUTATIONS DANS LES COLLECTIONS JAVA

elles sont nombreuses !

par exemple :

$$add : List[T] \hookrightarrow (Integer, T) \rightarrow ()$$

$$add : List[T] \hookrightarrow T \rightarrow Boolean$$

$$remove : List[T] \hookrightarrow Integer \rightarrow T$$

$$remove : List[T] \hookrightarrow Object \rightarrow Boolean$$

$$subList : List[T] \hookrightarrow (Integer, Integer) \rightarrow List[T]$$

plus ou moins repérables niveau signature
et avec différentes approches de la mutation

EXERCICE

- tours de pistes en sport automobile
- `totalTime`
  - temps total hors 1er tour
  - seules les listes de taille 2+ sont considérées
- `avgTime`
  - temps moyen hors 1er tour
  - seules les listes de taille 2+ sont considérées

EXERCICE : PROPOSITION

Java

```java
static double totalTime(List<Double> lapTimes) {
    lapTimes.remove(0);
    double sum = 0;
    for (double time : lapTimes) { sum += time; }
    return sum;
}
```

Java

```java
static double avgTime(List<Double> lapTimes) {
    double time = totalTime(lapTimes);
    int laps = lapTimes.size();
    return time / laps;
}
```

Java

```java
ArrayList<Double> lapTimes = new ArrayList<>();
lapTimes.add(31.0);
lapTimes.add(20.9);
lapTimes.add(21.1);
lapTimes.add(21.3);
System.out.println(totalTime(lapTimes)); // OK ?
System.out.println(avgTime(lapTimes));   // OK ?
```

EXERCICE : SOLUTION

Java

```java
static double totalTime(List<Double> lapTimes) {
    List<Double> lapTimesNoWarmup = new ArrayList<>(lapTimes);
    lapTimesNoWarmup.remove(0);
    double sum = 0;
    for (double time : lapTimesNoWarmup) { sum += time; }
    return sum;
}
```

Java

```java
static double avgTime(List<Double> lapTimes) {
    double time = totalTime(lapTimes);
    List<Double> lapTimesNoWarmup = new ArrayList<>(lapTimes);
    lapTimesNoWarmup.remove(0);
    int laps = lapTimesNoWarmup.size();
    return time / laps;
}
```

note : la solution n'est **pas DRY**

on vera une solution par la suite

ETAT MUTABLE PARTAGÉ

Shared Mutable State (SMS)

- **état :** valeur stockée en un lieu unique
- **mutable :** pouvant être modifiée sur place
- **partagé :** accessible à pls parties du programme

voir dans les exemples précédents :

- `List<String> plan`
- `List<Double> lapTimes`

base de la programmation impérative
techniques pour limiter les problèmes
(cf cours de qualité architecturale / patrons)

SOLUTIONS AU SMS

- solution utilisée pour `replanifie` :
  fonction pure + évitement mutation
- solution “OO” : encapsulation
  (perte possible confiance, attention aux fuites)
- solution “PF” : états immuables
  (confiance, pas de mutation)

en Scala

Scala

```scala
def replanifie(plan: List[String], nouvelleVille: String, avantVille: String): Li
    val index = plan.indexOf(avantVille)
    val villesAvant = plan.slice(0, index)
    val villesApres = plan.slice(index, plan.size)
    villesAvant.appended(nouvelleVille).appendedAll(villesApres)
}
```

COLLECTIONS IMMUABLES

- support natif en Scala
- support natif faible en Java
- support étendu en Java avec des librairies (vavr.io)

en Scala

Scala

```
scala> val liste1 = List("Apple", "Book")
val liste1: List[String] = List(Apple, Book)

scala> val liste2 = liste1.appended("Mango")
val liste2: List[String] = List(Apple, Book, Mango)

scala> liste1.size
val res0: Int = 2

scala> liste2.size
val res1: Int = 3
```

EXERCICE

écrire en Scala les fonctions (méthodes)

$$firstTwo : List[String] \rightarrow List[String]$$

$$lastTwo : List[String] \rightarrow List[String]$$

$$movedFirstTwoToTheEnd : List[String] \rightarrow List[String]$$

$$insertedBeforeLast : (List[String], String) \rightarrow List[String]$$

ayant les propriétés suivantes

```scala
firstTwo(List("a", "b", "c")) == List("a", "b")
lastTwo(List("a", "b", "c")) == List("b", "c")
movedFirstTwoToTheEnd(List("a", "b", "c")) == List("c", "a", "b")
insertedBeforeLast(List("a", "b"), "c") == List("a", "c", "b")
```

indice : utiliser `slice`

RÉSUMÉ

- définition de SMS : état + mutable + partagé
- la mutabilité est dangereuse
- on peut lutter contre elle
  - par l'utilisation de copies
  - par l'utilisation de valeurs immuables
- importance des API de collections immuables