BrownBagLunch

Programmation Fonctionnelle en Scala

Karol Chmist
@karolchmist
http://www.chmist.com

Programmation fonctionnelle

Fonctions pures

Immuabilité

Collections

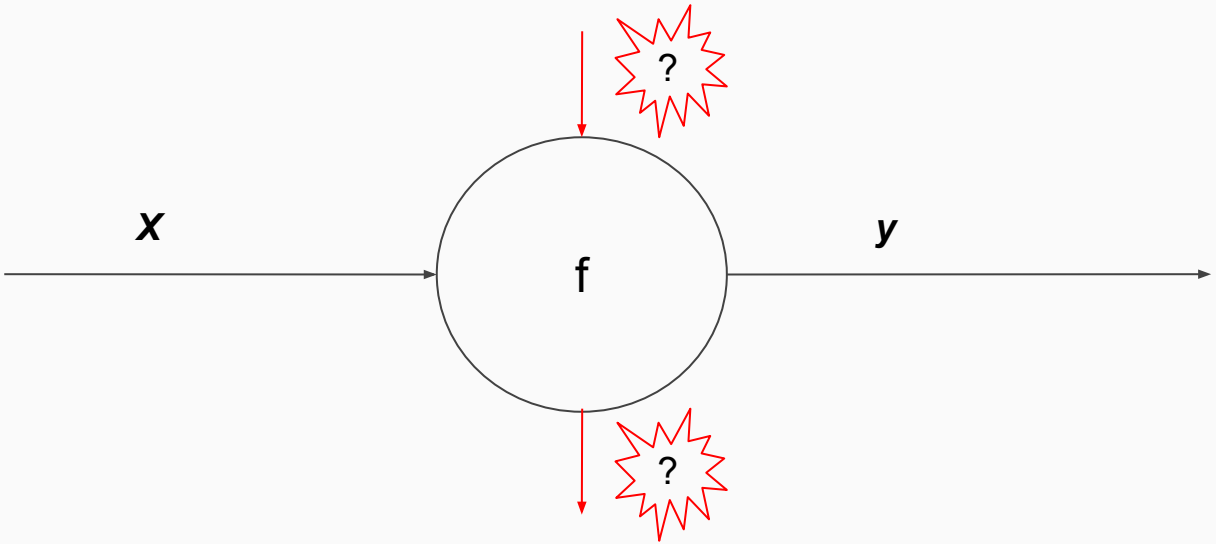
Valeurs vides

Async

Exceptions

Fonctions pures

- même résultat pour même argument
- sans effets de bord observables



Fonctions pures

```
class Person { String name; } // name modifiable

Person p = new Person();

p.name = "John";

foo(p);

bar(p);

baz(p);

p.name.equals("John"); // ???
```

```
case class Person(name: String) // constant

val p = Person(name = "John")

val a = foo(p)

val b = bar(a)

val c = baz(a)

p.name == "John" // true
```

Fonctions pures - composables

```
case class Person(name: String)

val p = Person("John")

val a: Person = foo(p)

val b: String = bar(a)
```

```
val foo: Person => Person = ???

val bar : Person => String = ???

val fooBar: Person => String = foo andThen bar

val x = fooBar(p)

val x = bar(foo(p))
```

Immuabilité

String, int -> immuables

```
List numbers = new ArrayList<>();
numbers.add(3);

class Person { String name; }

Person p = new Person();

p = null;

p.name = "Jerry";

p.name = "Gerry";
```

```
case class Person(name: String, age: Int)

val p = Person("john", 28)

// p = Person("jack", 29) -> impossible

val PPP = p.copy(name = p.name.toUpperCase)

// p.name = "jack" -> impossible

p.name == "john" // true

PPP.name == "JOHN" // true
```

Collections

```
List<Integer> numbers = new ArrayList<>();
// …

List<Integer> newNumbers = new ArrayList<>();

for (Integer n : numbers) {
   if (n > 18) {
      newNumbers.add(n * 2);
   }
}
```

Impératif

```
val numbers: Seq[Int] = Seq(43, 2, 21, 9)

val newNumbers = numbers
 .filter(n => n > 18)
 .map(n => n * 2)
```

Déclaratif

Options

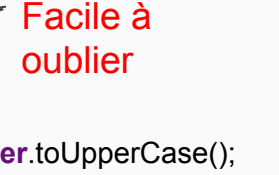
- quand valeur peut être absente
- comme liste, mais un élément au maximum possible
- oblige à traiter l’absence de la valeur
- évite null et NullPointerException

Options

```
class Project {
  String leader;
  String title;
}

if (project != null) {
  if (project.leader != null) {
    String leaderUpperCase = project.leader.toUpperCase();
    if (project.title != null) {
      return "Project "+project.title+" by "+leaderUpperCase;
    }
  }
}

return "Project doesn't exists";
```




```
case class Project(leader: Option[String], title: Option[String])

val maybeProject = Some(Project(Some("John"), None))

val maybeDescription = for {
  project <- maybeProject
  leader <- project.leader
  leaderUpperCase = leader.toUpperCase
  title <- project.title
} yield s"Project $title by $leaderUpperCase"

maybeDescription.getOrElse("Project doesn't exists")
```

Options

```
val maybeTitle = Some("matrix")

val maybeLength = maybeTitle
 .filter(title => ttitle.startsWith("m"))
 .map(_.length)

maybeLength == Some(6)
```

```
val maybeTitle = Some("matrix")

val maybeLength = for {
   title <- maybeTitle
   if title.startsWith("m")
} yield title.length

maybeLength == Some(6)
```

Futures

- calculs asynchrones et non bloquants
- contient une valeur qui n’existe pas encore
- facile à composer, parallélisme de haut niveau
- utilisation du même langage que les collection - map, filter...

Futures

```
case class Person(id: Long, name: String)

def fetchPersonAsync(id: Long) : Future[Person] = ???

val eventualPerson: Future[Person] = fetchPersonAsync(5)

val eventualBigName: Future[String] = eventualPerson.map(p => p.name.toUpperCase)
```

Pas d’exceptions

```
case class Error(reason: String)
case class Person(id: Long, name: String)

def fetchPerson(id: Long) : Either[Error, Person] = ???

val errorOrPerson: Either[Error, Person] = fetchPerson(id = 3)

val errorOrName: Either[Error, String] = errorOrPerson.right.map(p => p.name.toUpperCase)

val result: Result = errorOrName
 .fold(
   error => Results.InternalServerError(s"Error : ${error.reason}"),
   name => Results.Ok(s"Found ${name}")
 )
```

Résultat final

```
def fetchPerson(id: Long)
                : Future[Either[Error, Option[Person]]] = ???
```

```
type MyResult[A] = Future[Either[Error, A]]
```

```
def fetchPerson(id: Long) : MyResult[Option[Person]] = ???
```

```
val result: MyResult[Option[Person]] = fetchPerson(2)

val reponse: Future[Result] = result.map(
      errorOrPerson =>
          errorOrPerson.fold(errorToResult,  maybeToResult )
)

def maybeToResult[A](maybeValue: Option[A]) =
     maybeValue
        .map(p => Results.Ok(Json.toJson(p)))
        .getOrElse(Results.NotFound)

def errorToResult(error: Error) =
     Results.InternalServerError(Json.toJson(error))
```

Résumé

- langage commun (map, filter, …)
- difficile à tricher
- explicité (pas de surprise, in -> out)
- composable
- fondation très stable (mathématiques)

Questions ?

Karol Chmist
@karolchmist
http://www.chmist.com