

# Programmation fonctionnelle sur la JVM :



ou ?



Igor Laborie  
Expert Web & Java

 @ilaborie

---

 igor@monkeypatch.io

---

<Monkey Patch/>

## Langages pratiqués

1. Java
2. JavaScript (TypeScript, CoffeeScript)
3. Kotlin, Scala

## Notions dans

Go , Python , Racket , Rust , SML , Swift , ...



- I. ➔ Langages fonctionnels
- II. ➔ Partie I
- III. ➔ Partie II
- IV. ➔ Remarques sur la performance
- V. ➔ Conclusion

# LANGAGES FONCTIONNELS

“

Fooling around with alternating current (AC) is just a waste of time. Nobody will use it, ever.

*Edison, 1889 inventeur, scientifique, fondateur de General Electric*

“

There is no reason anyone would want a computer in their home.

*Ken Olson, 1977 cofondateur DEC*

“

I predict the Internet will soon go spectacularly supernova and in 1996 catastrophically collapse.

*Robert Metcalfe, 1995 inventeur ethernet, fondateur de 3com*

## Paradigmes

- programmation impérative
- programmation orientée objet
- programmation fonctionnelle
- programmation logique
- ...

“

On peut adopter donc un style de programmation fonctionnelle avec la plupart des langages. Les caractéristiques des langages peuvent rendre cela plus ou moins facile (voir obligatoire)

Mais alors, c'est quoi un langage fonctionnel ?

“

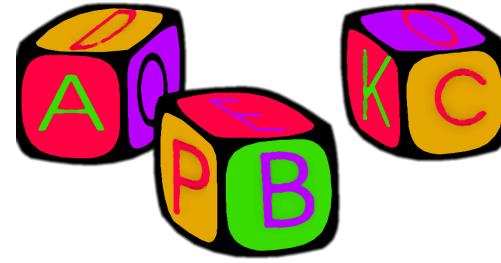
Il n'y a qu'un langage fonctionnel : le  $\lambda$ -calcul

# Fellowship

#8



Developpeur  
fonctionnel



Developpeur  
impératif ou OO



Photo by Vasilios Muselimis

“

Programation fonctionnelle  $\Rightarrow$  Typage statique

## Typage dynamique

Lisp  
Scheme  
Racket  
Clojure  
...

## Typage statique

(1958) ML (1973)  
(1975) Haskell (1990)  
(1994) OCaml (1996)  
(2007) Scala (2004)

# PARTIE I

@ilaborie #Java #Kotlin #Scala



```
class Utils {  
  
    public static String doSomething(String arg) {  
        throw new RuntimeException("Not yet implemented");  
    }  
  
    public static Function<String, String> asValue = Utils::doSomething;  
  
    public static Function<String, String> aLambda = (String s) -> {  
        throw new RuntimeException("Not yet implemented")  
    };  
  
}
```



# Functions en Scala, Kotlin

#12

```
package object apackage {  
  
    def doSomething(arg: String): String = ???  
  
    val asValue: String ⇒ String = doSomething  
  
    val aLambda: String ⇒ String = (s: String) ⇒ ???  
}
```



```
fun doSomething(arg: String): String = TODO()  
  
val asValue: String → String = ::doSomething  
  
val aLambda: String → String = { s: String → TODO() }
```



```
class Test{  
    int sum = 0;  
  
    public void compute() {  
        var arr = List.of(1, 2, 3, 4, 5);  
        for (int i : arr) {  
            sum += i;  
        }  
        System.out.println(sum);  
    }  
}
```



- ! Évitez les fonctions avec effet de bord !
- C'est un nid à bugs.
- ⇒ Évitez les fonctions qui retournent `void`, ou qui n'ont pas de paramètres.

# Pas d'effet de bord

#14

```
int sum = List.of(1, 2, 3, 4, 5)
    .stream() // Stream<Integer>
    .reduce(0, (acc, i) -> acc + i);
System.out.println(sum);
```



```
val sum = List(1, 2, 3, 4, 5)
    .foldLeft(0) { (acc, i) => acc + i } // or .sum
println(sum)
```



```
val sum = listOf(1, 2, 3, 4, 5)
    .fold(0) { acc, i -> acc + i } // or .sum()
println(sum)
```



# Instruction vs Expression

#15

```
IntPredicate isEven = n -> {
    if (n % 2 == 0) {
        return true;
    } else {
        return false;
    }
};

List.of(1, 2, 3, 4, 5)
    .forEach(i ->
        System.out.println(" " + i + " is even? " + isEven.apply(i))
    );
```



# Expressions

#16

```
IntFunction<Boolean> isEven = n ->  
    (n % 2 == 0)? true : false;
```



```
val isEven = (n: Int) =>  
    if (n % 2 == 0) true else false
```



```
val isEven = { n: Int →  
    if (n % 2 == 0) true else false  
}
```



“

The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as sites of higher level behaviors more appropriate for use as dynamic components.

--Alan Key, The Early History Of Smalltalk

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public void translateX(int offset) {  
        this.x += offset;  
    }  
  
    // + Getters  
    // + Setters  
    // + equals & hashCode  
    // + toString  
}
```



```
public class Point {  
    private final int x;  
    private final int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public Point translateX(int offset) {  
        return new Point(this.x + offset, this.y);  
    }  
  
    // + Getters  
    // + equals & hashCode  
    // + toString  
}
```



# Scala, Kotlin Immutable

#20

```
case class Point(x: Int, y: Int) {  
  
    def translateX(offset: Int): Point =  
        this.copy(x = x + offset)  
  
    // generated: Getters, equals & hashCode, toString, ...  
}
```



```
data class Point(val x: Int, val y: Int) {  
  
    fun translateX(offset: Int): Point =  
        this.copy(x = x + offset)  
  
    // generated: Getters, equals & hashCode, toString, ...  
}
```



Comment fait-on avec les structures de données ?

```
public class List<T> {  
    private final T[] array;  
  
    public List(T[] elements) {  
        this.array = Arrays.copyOf(elements, elements.length);  
    }  
  
    public List<T> add(T element) {  
        var newElts = Arrays.copyOf(this.array, this.array.length + 1);  
        newElts[this.array.length] = element;  
        return new List<>(newElts);  
    }  
}
```



On peut utiliser ➔ Eclipse Collections , ➔ Vavr , ...

# Java High Order function

#22

```
var digits = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);  
  
IntPredicate isEven = n -> (n % 2 == 0);  
  
List<Integer> evenDigits = digits.stream()  
    .mapToInt(i -> i)  
    .filter(isEven)  
    .boxed()  
    .collect(Collectors.toList());
```



```
UnaryOperator<IntPredicate> not = predicate -> i -> !predicate.test(i)  
IntPredicate isOdd = not.apply(isEven); // isEven.negate()  
var oddDigits = // ...
```



# Scala, Kotlin High Order function

#23

```
type Predicate[E] = E => Boolean  
  
def not[E](p: Predicate[E]): Predicate[E] = e => !p(e)  
  
val isEven: Predicate[Int] = n => (n % 2 == 0)  
val isOdd = not(isEven)  
  
var evenDigits = digits.filter(isEven)  
val oddDigits = digits.filter(isOdd)
```



```
typealias Predicate<E> = (E) -> Boolean  
  
fun <E> not(p: Predicate<E>): Predicate<E> = { e -> !p(e) }  
  
val isEven: Predicate<Int> = { n -> n % 2 == 0 }  
val isOdd = not(isEven)  
  
var evenDigits = digits.filter(isEven)  
val oddDigits = digits.filter(isOdd)
```



```
public static List<Event> notFunErrors1(List<Event> events, int size) {  
    List<Event> result = new ArrayList<>();  
    for (int i = 0; i < result.size(); i++) {  
        Event event = events.get(i);  
        if (event.isError()) {  
            result.add(event);  
        }  
        if (result.size() ≥ size) {  
            return result;  
        }  
    }  
    return result;  
}
```

```
public static List<Event> notFunErrors2(List<Event> events, int size) {  
    List<Event> result = new ArrayList<>();  
    for (Event event: events) {  
        if (event.isError()) {  
            result.add(event);  
        }  
        if (result.size() >= size) {  
            return result;  
        }  
    }  
    return result;  
}
```

```
public static List<Event> funErrors(List<Event> events, int size) {  
    return events.stream()  
        .filter(Event::isError)  
        .limit(size)  
        .collect(Collectors.toList());  
}
```

```
def notFunErrors(events: List[Event], size: Int): List[Event] = {  
    for {  
        event ← events  
        if event.isError  
    } yield event  
}.take(size)  
  
def funErrors(events: List[Event], size: Int): List[Event] =  
    events  
    .filter(_.isError)  
    .take(size)
```

```
fun notFunErrors(events: List<Event>, size: Int): List<Event> {   
    val result = mutableListOf<Event>()  
    for (event in events) {  
        if (event.isError) {  
            result.add(event)  
        }  
        if (result.size ≥ size) {  
            return result  
        }  
    }  
    return result  
}  
  
fun funErrors(events: List<Event>, size: Int): List<Event> =  
    events  
        .filter { it.isError }  
        .take(size)
```

```
public static int factorialFor(int n) {  
    int acc = 1;  
    for (int i = 2; i <= n; i++) {  
        acc *= i;  
    }  
    return acc;  
}
```



```
public static int factorialRec(int n) {  
    return (n <= 1) ? 1 : n * factorialRec(n - 1);  
}
```



```
public static int factorialTailRec(int n) {  
    return factorialTailRecAux(1, n);  
}  
private static int factorialTailRecAux(int acc, int n) {  
    return (n <= 1) ? acc : factorialTailRecAux(acc * n, n - 1);  
}
```



```
public static int factorialStream(int n) {  
    return IntStream.rangeClosed(1, n)  
        .reduce(1, (acc, i) -> acc * i);  
}
```



# Scala, Kotlin Tail Recursion

#31

```
def factorialTailRec(n: Int): Int = {  
    @tailrec  
    def aux(acc: Int, n: Int): Int =  
        if (n ≤ 1) acc else aux(acc * n, n - 1)  
  
    aux(1, n)  
}
```



```
tailrec fun factorialTailRec(n: Int): Int {  
    fun aux(acc: Int, n: Int): Int =  
        if (n ≤ 1) acc else aux(acc * n, n - 1)  
    return aux(1, n)  
}
```



```
speakers.filter(speaker -> speaker.xp > 10 &&  
                speaker.getLanguages().contains("Java"));
```



```
speakers.filter(speaker -> speaker.xp > 10) // is experimented  
          // is love Java  
          .filter(speaker -> speaker.getLanguages().contains("Java"));
```



```
Predicate<Speaker> isExperimented = speaker -> speaker.xp > 10;   
Predicate<Speaker> isLoveJS = speaker -> speaker.getLanguages().contains("Ja  
  
speakers.filter(isExperimented)  
          .filter(isLoveJS);
```

```
speakers.filter(isExperimented.and(isLoveJS));
```



- 😊 la notation Lambda des `@FunctionalInterface`
- 😊 API Stream
- 😊 `Function#compose`, `Function#andThen`
- 😊 / 😱 API `java.util.function.*`
- 😤 immutable, trop lourd de mettre les `final`
- 😢 pas de collections immutables ⇒ bibliothèques
- 😢 pas de `tailrec`
- 😢 trop de *boilerplate*
- 💣 eviter les effets de bord ⇒ 💕 tests

- 😍 syntaxe plus expressif
- 😊 API plus riche
- 😎 tailrec, data ou case classes, ...
- 💪 typage de Scala très puissant

# PARTIE II

@ilaborie #Java #Kotlin #Scala



- Function as First Class Citizen
- High Order Function
- Referential Transparency
- Idempotent

- Curryfication,
- Memoïsation,
- Algebraic Data Type,
- Pattern Matching,
- Functor, Monïd, Monad, ...

“

Transformation d'une fonction de plusieurs arguments en une chaîne de fonctions d'un seul argument qui donnera le même résultat lorsqu'il est appelé en séquence avec les mêmes arguments.

$$f(x, y, z) = g(x)(y)(z)$$

- Viens de [W Moses Schönfinkel](#) et [W Haskell Curry](#)
-  sens des arguments

```
IntBinaryOperator mult = (a, b) -> a * b;  
  
// Curry  
IntFunction<IntFunction> curriedMult = b -> a -> a * b;  
  
// Usage  
IntFunction identity = a -> mult.applyAsInt(a, 1);  
  
IntFunction dbl = curriedMult.apply(2);
```



## ► Vavr Currying

```
type IntFun = (number) => number; TS

const stupidMemoizer = (fun: IntFun): IntFun => {
    const cache: number[] = [];
    return (n: number) => {
        const cached = cache[n];
        if (typeof cached === 'number') {
            return cached;
        }
        return (cache[n] = fun.call(null, n));
    }
};

const fibonacci: IntFun = stupidMemoizer(n => {
    switch (n) {
        case 1 :
            return 1;
        case 2 :
            return 1;
        default:
            return fibonacci(n - 2) + fibonacci(n - 1);
    }
});

console.log('fibonacci(15)', fibonacci(15));
```

```
public static int fibonacci(int n) {  
    switch (n) {  
        case 1 return 1;  
        case 2: return 1;  
        default:  
            return fibonacci(n - 2) + fibonacci(n - 1);  
    }  
}  
  
public static IntUnaryOperator stupidMemoizer(IntUnaryOperator func) {  
    Map<Integer, Integer> cache = new HashMap<>();  
    return n -> cache.computeIfAbsent(n, func::applyAsInt);  
}  
  
public static void main(String[] args) {  
    var fibo = stupidMemoizer(Memo::fibonacci);  
    System.out.println(fibo.applyAsInt(15));  
}
```



- 💎 purtée ⇒ on peut mettre en cache !
- 💤 Caffeine
- 💣 cache avec des objets mutables

```
type schoolPerson = Teacher  
| Director  
| Student(string);
```

RE



Derive4J

- 😞 on peut utiliser des *abstract class* ou des *enum* ne sont pas des vraiment des ADT.

# Pattern Matching

#43

```
let greeting = stranger =>
  switch (stranger) {
    | Teacher => "Hey professor!"
    | Director => "Hello director."
    | Student("Richard") => "Still here Ricky?"
    | Student(anyOtherName) => "Hey, " ++ anyOtherName ++ "."
  };
```

RE

- `if-elseif-else` Hell avec `instanceof`
- ➡ Pattern Matching de Vavr
- Un jour peut être avec ➡ Project Amber
  - ➡ JEP 325: Switch Expressions
  - ➡ JEP 305: Pattern Matching

# Pattern matching en Scala, Kotlin

#45

```
val greeting = stranger match {  
    case Teacher          => "Hey professor!"  
    case Director         => "Hey director."  
    case Student("Richard") => "Still here Ricky?"  
    case Student(name)     => s"Hey, $name."  
}
```



```
val greeting = when (stranger) {  
    is Teacher          -> "Hey professor!"  
    is Director         -> "Hey director."  
    Student("Richard") -> "Still here Ricky?"  
    is Student          -> "Hey, ${stranger.name}."  
}
```



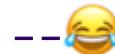
“

A monad is just a monoid in the category of endofunctors, what's the problem?



“

Généralisation aux catégories de la notion de morphisme.



```
interface Functor<A> {  
    Functor<B> map(mapper: Function<A, B>);  
    // avec associativité  
}
```

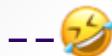


```
interface EndoFunctor<A> {  
    EndoFunctor<A> map(mapper: UnaryOperator<A>);  
}
```



“

C'est un magma associatif et unifère, c'est-à-dire un demi-groupe unifère.



```
interface SemiGroup {  
    SemiGroup concat(SemiGroup other);  
    // this.concat(x.concat(y)) = this.concat(x).concat(y)  
}
```



```
interface Monoid extends SemiGroup {  
    static Monoid neutral = ???;  
    // monoid.concat(neutral) = monoid, neutral.concat(monoid) = monoid  
}
```



```
interface Monad<A> extends Functor<A> {  
    Monad<B> flatMap(mapper: Function<A, Monad<B>>);  
}
```



“

J'ai toujours pas compris !

-- 😳, 😰

- C'est un ~~objet~~ 🍔
- qui a des méthodes simples comme par exemple `map` ou `flatMap`
- qui doivent respectées des règles (axioms)
- ce qui garenti une haute composabilité.
- `Option<V>`, `Either<A,B>`, `Try<S,E>`, `Future<V>`, ...



- Mais on n'a pas envie d'implémenter `map`, `flatMap` pour chaque Monades.
- Si on définissait des types plus abstrait pour cela ?

⇒ High Order Kind (Higher-Kinded Types, Higher-Order Types, ...)

# REMAQUES SUR LA PERFORMANCE

## Performance en quoi ?

- temps d'exécution (minimum, maximum, moyen, première exécution) ?
- consommation de mémoire ?
- consommation d'énergie ?
- ...

“

Douter de toutes les mythes et légendes

- Faire attention aux bonnes structures de données (complexité algorithmique, Data oriented Design)
- Eviter les IO (disque, réseau), c'est l'occasion de faire de la FRP, ➔ Le Manifeste Réactif
- Code bien testé
- Privilégier la lisibilité du code à une (hypothétique) optimisation de performance
- Mettre en cache n'est pas toujours la bonne solution

“

tous les leviers sont bon, y compris le langage

- Définir le seuil désiré
- Effectuer des mesures
- Isoler la zone à optimiser (la plus petite possible)
- Commenter pourquoi on n'a perdu de la lisibilité
- Suivre l'évolution des performances

# CONCLUSION

@ilaborie #Java #Kotlin #Scala



- 🌾 depuis Java 8 on a `java.util.function`, et `java.util.stream`
  - 💅 écosystème dans ce domaine, plutôt à la mode (Rx, Vavr)
- 
- 😢 API horrible en Java (`XXXFunction`, `CompletableFuture`, `Collectors.toList`)
  - il manque des structures plus *lazy* voir [▶ Lazy Java](#)

- ➡ Project Amber
- ➡ Project Valhalla
- ➡ JMH
- ➡ Vavr
- `java.util.concurrent.Flow`, 🐱 RxJava, ➡ Reactor
- ➡ <https://drboolean.gitbooks.io/mostly-adequate-guide-old/content/>
- 🐱 Awesome Java # FP

-  [Kotlin](#)
-  [Scala](#)
-  [Clojure](#)
- ...

- 🧽 plus simple
- ✅ Plus facile à tester
- 🐛 moins de bugs
- 🦎 plus évolutif
- 🔄 applicable sur tous les (bon) langages
- 🎓 apprendre

Questions ?

