FUNCTIONAL PROGRAMMING SEEMS SUPER COOL!

Málaga Scala Developers

OOP / FP

Composition; Abstraction =>	Objects Interfaces Classes	First-class functions Higher-order f. Pure functions
Data Structures =>	Mutable	Immutable
Side effects =>	Embedded in the objects	No side effects or represented with types
Style =>	Imperative	Declarative
Evaluation =>	Strict/Eager	Nonstrict/Lazy

(OOP + _) => ???

Composition; Abstraction =>	Objects Interfaces Classes	First-class functions Higher-order f. Pure functions
Data Structures =>	Mutable	Immutable
Side effects =>	Embedded in the objects	No side effects or represented with types
Style =>	Imperative	Declarative
Evaluation =>	Strict/Eager	Nonstrict/Lazy

Object Oriented

(OOP + _) => ???

Composition; Abstraction =>	Objects Interfaces Classes	First-class functions Higher-order f. Pure functions
Data Structures =>	Mutable	Immutable
Side effects =>	Embedded in the objects	No side effects or represented with types
Style =>	Imperative	Declarative
Evaluation =>	Strict/Eager	Nonstrict/Lazy
	Object Oriented	**Functional Oriented**

(OOP + FP) => ???

Composition; Abstraction =>	Objects Interfaces Classes	First-class functions Higher-order f. Pure functions
Data Structures =>	Mutable	Immutable
Side effects =>	Embedded in the objects	No side effects or represented with types
Style =>	Imperative	Declarative
Evaluation =>	Strict/Eager	Nonstrict/Lazy

Object Oriented

Functional Oriented

SCALA

Scala?! Again new language… but **why**? Java is **not** good enough?

- Java Developer

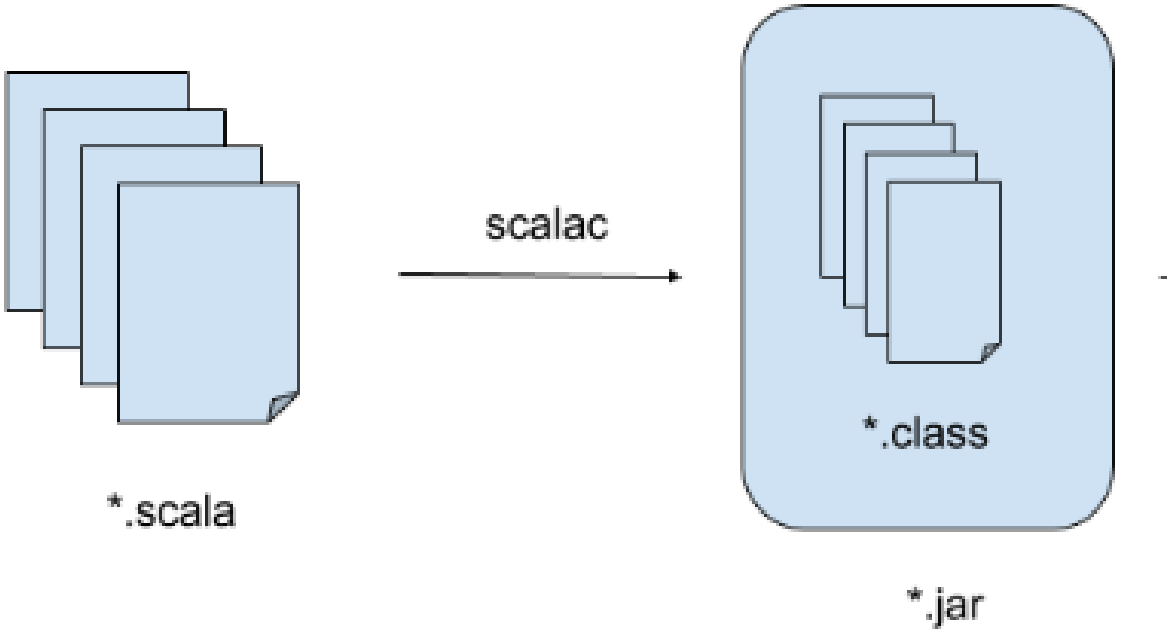
```java
public class Programmer {

        private String name;
        private String language;
        private String favDrink;

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getLanguage() {
                return language;
        }
        public void setLanguage(String language) {
                this.language = language;
        }
        public String getFavDrink() {
                return favDrink;
        }
        public void setFavDrink(String favDrink) {
                this.favDrink = favDrink;
        }
}
```

- Scala Developer

```scala
class ScalaProgrammer(
  var name:String,
  var language:String,
  var favDrink:String
  )
_
```

Great syntax + OOP + FP = Paradise?

¡¿How it is works?!



TODO:

1. Write Scala class → class ScalaProgrammer (var name : String, var language: String)
2. Compile it → scalac ScalaProgrammer.scala
3. See the result of scalac → use javap JDK tools → javap -p ScalaProgrammer.class

Scala syntax in 1/2 steps

```
def max (x: Int, y: Int) : Int = {
   if (x  >  y)
      x
   else
      y
}
```

```
Integer max (Integer x, Integer y){
   If (x > y)
      return x;
   else
      return y;
}
```

Scala syntax in 2/2 steps

```
object MyModule{

   def abs(n: Int) : Int = {
      if (n < 0) -n
      else n
   }

   private def printAbs(x : Int) : String = {
      val message = "Absolute value of %d is %d"
      msg.format(x, abs(x))
   }

   def main(args: Array[String]) : Unit = {
      println(printAbs(-10))
   }

}
```

FP => First-class functions High Order Functions

Functions that can take other functions as a argument and return another function

```
scala> def divideBy2(n: Int): Int = n / 2
```

```
scala> def plusOne(f: Int => Int): Int => Int =
   f andThen (f=> f + 1)
```

```
scala> def divideBy2Plus1 = plusOne(divideBy2)
```

- TODO =>
  How write divideBy2 literaly? <=> lambda as an anonymous function; compose

FP => First-class functions
High Order Functions

Let’s try implement

High Order and **Pure function**

FP => First-class functions
High Order Functions

Let’s try implement

High Order and **Pure function**

in

JAVA API

FP => Immutable

The contents of the object can't be altered

```
val immutableVal = 1
```

```
final Integer immutable = 1;
```

- TODO =>
  Mutable VS Immutable
  Set[A] = {good_state(x,y) ,good_state'(x',y')}
  bad_state(x',y) ∉ Set[A] ; bad_state(x,y') ∉ Set[A]

FP => Lazy initialization

Initialization deferred until first time used

```
val lazy soLazy = 1
```

```
Integer a = null;
//somewhere in code
if (a == null){
    a = 1;
} else return a;
```

- TODO =>

```
object Demo{
        lazy val x   ={ println("initializing x");
        "done"}
        }
```

FP => Tail recursion

Bye bye... stack overflow

```
def tailRecursion(x :List[Int], sum : Int) : Int ={
   if (x.isEmpty) sum
      else{
         val addition : Int = if (sum < 99) 1 else 0
         tailRecursion(x.tail, sum + addition)
      }
}
```

A tail-recursive function will not build a new stack frame for each call
all calls are execute in a single frame.

- `TODO =>`
  `define an lazy tail recursion function`

TAKE AWAYS

- f: (input: InputType) => OutputType

TAKE AWAYS

- f: (input: InputType) => OutputType
- Immutability – state changing as a new object

TAKE AWAYS

- f: (input: InputType) => OutputType
- Immutability – state changing as a new object
- Tail recursion and lazy inicialization

TAKE AWAYS

A little functional thinking :)

```
def foo[A,B,C] (a: A, f: (A, B) => C): B => C =
???
```

¿Quién quiere explicar?

TAKE AWAYS

A little functional thinking :)

```
def foo[A,B,C] (a: A, f: (A, B) => C): B => C =
(b: B) => f(a,b)
```

¡GRACIAS!