Scala cruise

Snorkelling in some of the Scala features

Alessandro Candolini

October 5, 2018

Itinerary

1. Let's meet Scala
2. Tools
3. The Play framework
4. Learn more

Let's meet Scala

Figure 1: Martin Odersky, the creator of Scala

Scala programming language is:

- general purpose
- strongly statically typed (with type inference)
- compiling primarily to JVM 8+ bytecode[1]
- interoperable with Java 8+

[1] `scala.js` compiles to JS; `scala native` targets LLVM; jdk 9+ compatibility might require support; there is a Scala REPL.

Scala supports

- OOP
- FP

```
val value : Int  = 2;
val text : String = "hello";
```

```
val value : Int  = 2
val text : String = "hello"
```

```
val value = 2
val text = "hello"
```

```
val value = 2
val text = "hello"

print(text)
print(s"$value-$text") // string interpolation
print(s"${value*2}")
```

```
var value = 2
val text = "hello"

value = 4
text = "hello" // !!! forbidden
```

```
def square(x : Int) : Int = {
  return x*x
}
```

```
def square(x : Int) = x*x
```

```
def abs(x : Int) =
  if (x >= 0) // if statement
    x
  else
    -x
```

```
def id(x : Int) : Int = {
  if (x >= 0) { // if control flow  (imperative)
    print(s"$x")
  }
  x
}

id(-2)
id(2)
```

```
def square(x : Int) = x*x

square(2+3+4) // <-- how is this evaluated?
```

```
def div(x : Double) : Double = 1/x

div(1) // 1
div(2) // 0.5
div(0) // java.lang.ArithmeticException: / by zero
```

```
def div(x : Double) : Double =
  if ( x != 0 )
     1/x
  else
     null // !!!
```

```
def div(x : Double) : Option[Double] =
  if ( x != 0 )
     Some[Double](1/x)
  else
     None
```

```
def div(x : Double) : Option[Double] =
  if ( x != 0 )
     Some(1/x)
  else
     None
```

```
def div(x : Double) =
  if ( x != 0 )
     Some(1/x)
  else
     None
```

```
def square(x : Double) = x*x

def div(x : Double) =
  if ( x != 0 ) Some(1/x) else None

square(div(3)) // !! Type mismatch

val twice = 2*div(3) // !! Type mismatch
```

```
def div(x : Double) =
  if ( x != 0 ) Some(1/x) else None

val twice : Option[Double]
   = div(3).map( value => 2*value)
```

```
def div(x : Double) =
  if ( x != 0 ) Some(1/x) else None

div(3).map(2*_)
```

```
def square(x : Int) = x*x

def div(x : Int) =
  if ( x != 0 ) Some(1/x) else None

div(0).map(value => square(value))
div(0).map(square)
```

```
def div(x : Double) =
  if ( x != 0 ) Some(1/x) else None

def someFunction(x : Double) : Option[Double] = ???

div(0).flatMap(anotherFunction)
```

Crash course in Monads

All told, a monad is just a monoid in the category of endo-functors

S. Mac Lane

Monads are return types at guide you through the happy path

E. Mejier

Monads are like burritos

Brent Yorgey

```
def f(a:A) : Future[B]
def g(b:B) : Future[C]

val a : A = ???

f(a).flatMap(g)
```

- Option[Option[A]] $\rightarrow$ Option[A]
- Future[Future[A]] $\rightarrow$ Future[A]
- List[List[A]] $\rightarrow$ List[A]

Compare this to *ad-hoc* Kotlin syntax for nullable types

```
var value : Int? = 0

value?.let { it }
```

```scala
// value classes
class ProductId(val id: Int) extends AnyVal

// case classes
case class ProductDetails(
  id: ProductId,
  description: String
) // notice: no val/var
```

```
def fetchIds() : Future[Seq[ProductId]] = ???

def fetchDetails(ids : Set[ProductId])
    : Future[Map[ProductId, ProductDetails] = ???

def hydrate() =
    fetchIds().flatmap( ??? )
```

There is much more than that in terms of language features:

- higher order functions
- currying
- different evaluation strategies (call by value vs call by name)
- immutable collections
- pattern matching
- algebraic data types (sealed traits, etc)
- variance and contravariance
- value classes
- for comprehension
- implicits ...

In additionI to language features, there is the Scala ecosystem of libraries:

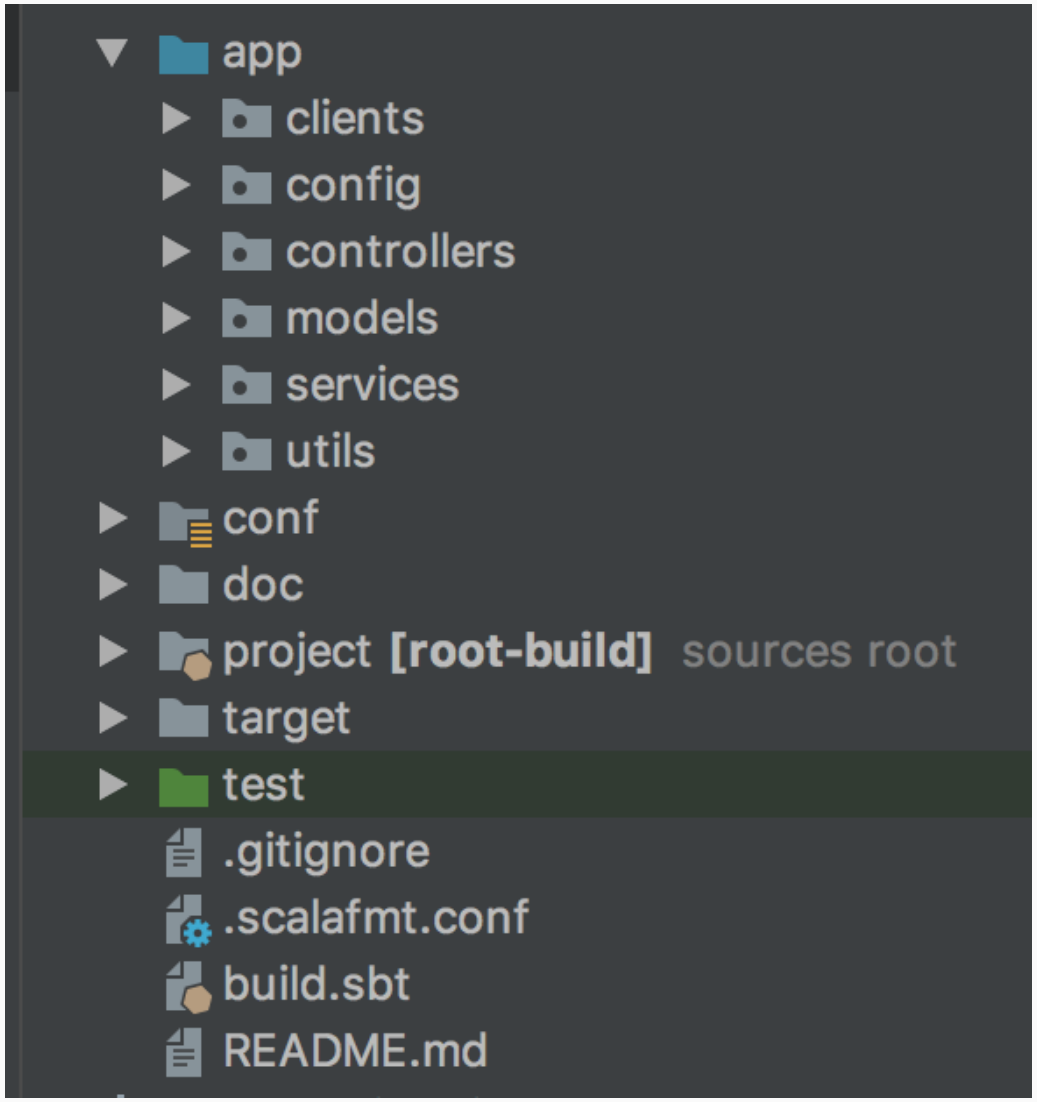
- Akka HTTP
- Akka actors
- scalaz
- cats
- etc

Tools

- IDE: intellij with Scala plugin
- Build tool: sbt
- Tests: scalatest vs specs2

The Play framework

Structure of a Play project



Routes file

```
GET    /fetch/:id
         controllers.Controller.fetch(id: Int)
```

A taste of contract driven development:

- apiary / blueprint
- dredd testing (local / staging)

Controllers: keep them simple

```
@Singleton // JSR 330, guice by default
class ProductClient @Inject()(wsClient: WSClient)
  (implicit ex: ExecutionContext) { // implicits !!

  def fetch() =  // using  Play wsclient
    wsClient.url("http://test.com").get()

}
```

There is much more than this:

- Json serialization/deserialization
- From futures to HTTP response codes (200, non-200)
- etc

Learn more

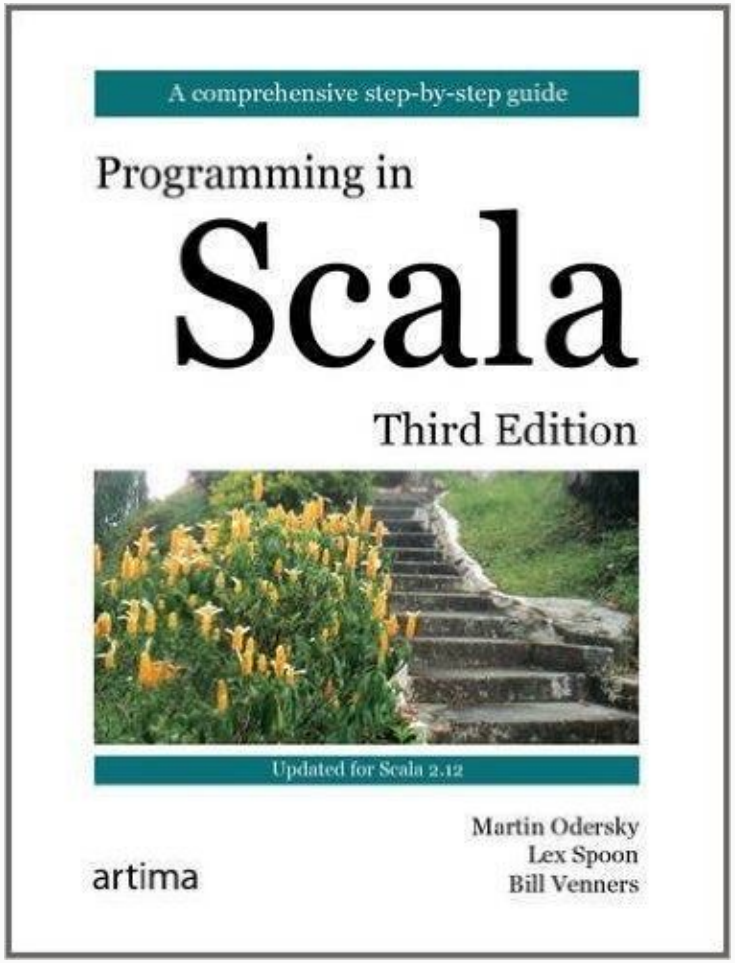


Figure 2: The reference guide

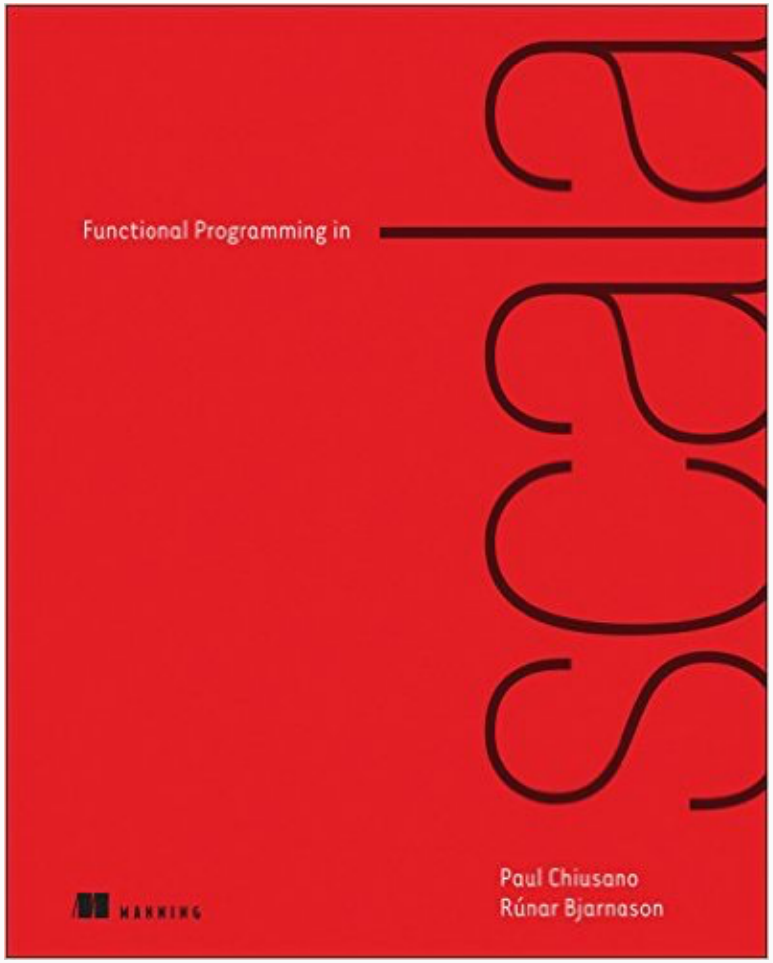


Figure 3: Gym to master FP in Scala

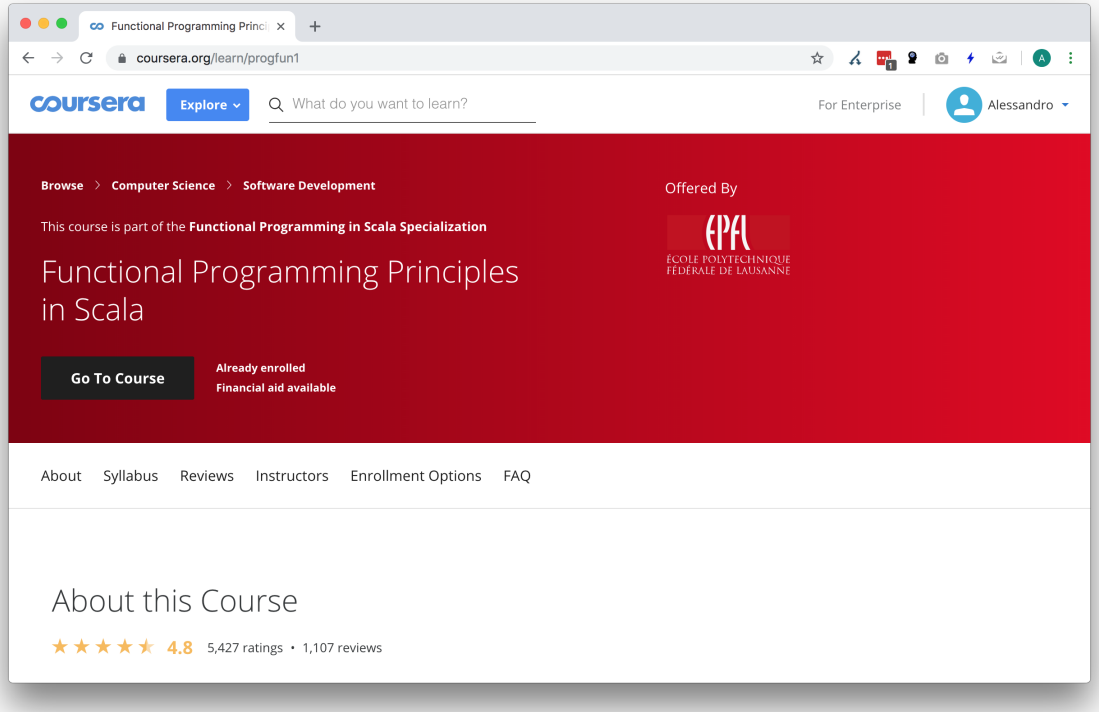


Figure 4: Functional programming principles in Scala MOOC by Martin Odersky

Online exercises:

- `https://www.scala-exercises.org/`
- `http://www.scalakoans.org/`

If you are interested in FP in Scala with attention to mathematical details...

Sergei Winitzki's *Functional programming in the mathematical spirit* video tutorials .

"Long and difficult, yet boring explanations given in excruciating detail." (quoting from the posts)

First lecture here:

- `https://www.youtube.com/watch?v=0Ld79Lnzx_o`

Questions?