Scala Intro

Twitter: @Scalabcn
Slack: scalabcn.slack.com

Victor Dubé @duferdev

Software engineer

Rodrigo Molina @rgermanm

Software engineer

Scala (Wikipedia description)

- Born on 2003-2004
- Born in EPFL (Switzerland)
- Father: Martin Odersky
- JVM language...but also Javascript! (scala.js)
- Type system: static, strong, inferred
- Multi-paradigm: Functional, OO, imperative, concurrent
- Current version: 2.12.7

Why Scala?

Why scala?

- Successful combination of FP and OOP.
- Sophisticated object and type system (flexibility in defining abstractions)
- Excellent basis for concurrent, parallel, and distributed computing. (Complex domains!)
- **Scala**ble (Small scripts -> Large distributed apps)
- Big and rich ecosystem base (Akka, Spark, Scalaz, Cats…)
- Compatible with Java

The central drive behind Scala is to make life easier and more productive for the developer

--Martin Odersky--

Scala “bad parts”

- breaks the “there is only one best way to do something” principle.
- High learning curve.
- Compiler

```
def doSomething():Unit = {...}
def doSomething:Unit = {...}
def doSomething() = {...}
def doSomething = {...}
def doSomething() {...}
def doSomething {...}

MyClass.doSomething
MyClass.doSomething()
MyClass doSomething()
MyClass.doSomething(param)
MyClass doSomething(param)
MyClass doSomething param
```


THIS IS
SCALA!
makeameme.org

Expression-oriented language

- Empower expressions over statements (EOP)
- (nearly) everything evaluates to a value (Control structures, side-effects methods...) for composability

```
val num: Int = if (true) 1 else 2
val unit: Unit = println("Hello guys!")
val nums: Seq[Int] = for(i ← 1 to 10) yield i

num  1: Int
unit  (): Unit
nums  Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10): scala.collection.Seq
```

Everything is an object

```
1 + 2 * 3 / 23  1: Int

// Same as
1.+(2.*(3)./(23))  1: Int
```

How??

Any method can be an operator

```
case class Euro(value: Int) {
  def add(other: Euro): Euro =
    Euro(this.value + other.value)

  def add(other1: Euro, other2: Euro): Euro =
    Euro(this.value + other1.value + other2.value)
}

// Method call
val amount = Euro(1).add(Euro(1)) // Euro(2)

// Operator
val twoEuros = Euro(1) add Euro(1) // Euro(2)
val tenEuros = twoEuros add (twoEuros, Euro(6)) //Euro(10)
```

Operator notations

```
case class Euro (value: Int) {
  def + (other: Euro): Euro =
    Euro(this.value + other.value)

  // only '+', '-', '!' or '~' supported
  def unary_- : Euro = Euro(-this.value)

  def asString: String = s"$value€"
}

val oneEuro = Euro(1)
// Infix notation
val twoEuros = oneEuro + oneEuro // Euro(2)
// Prefix notation (Same as twoEuro.unary_-)
val minusTwoEuros = -twoEuros // Euro(-2)
// Postfix notation
val twoEurosAsSring = twoEuros asString // 2€
```

Operator precedence

Scala decides precedence based on the first character of the methods used in operator notation.

Table 5.3 · Operator precedence

(all other special characters)
* / %
+ -
:
= !
< >
&
^
|
(all letters)
(all assignment operators)

-> End with =

```
a + b ^? c ?^ d less a ==> b | c
```

```
((a + b) ^? (c ?^ d)) less ((a ==> b) | c)
```

Use parentheses to clarify and override precedence

Operator associativity

```
case class Euro(value: Int) {
  // ...

  // Ends with : ⇒ right to left
  def +:(value: Int): Euro =
    Euro(this.value + value)
}

val oneEuro = Euro(1)

//Same as oneEuro.::(1)
val twoEuros = 1 +: oneEuro // Euro(2)
```

```
Left to right
a + b + c = (a + b) + c

Right to left
a +: b +: c = a +: (b +: c)
```

Functions are objects!

and you will learn it in 5 seconds...

```
Function0[R] // aka () ⇒ R
Function1[P1, R] // aka P1 ⇒ R
Function2[P1, P2, R] // aka (P1, P2) ⇒ R
   ...
Function22[P1, P2, ... , P22, R] // aka (P1, P2, ... , P22) ⇒ R
```

Java 8 Style

```
Consumer, Supplier, Predicate, Function, Operator, BiConsumer, UnaryOperator, BinaryOperator, BiOPredicate, BiFunction ...
```

But why java, why???

Functions are objects!

so first class citizens...

```
val f = new Function0[Int] {
  def apply() = 3
}

val sum3 = new Function1[Int, Int] {
  def apply(i: Int) = i + 3
}

val sum3Bis = (i: Int) ⇒ i + 3
// or val sum3Bis: Int ⇒ Int = i ⇒ i + 3

f()  3: Int
sum3(1)  4: Int
sum3Bis(1)  4: Int
```

Functions are objects!

so they have cool methods...

```
def sum(a: Int, b: Int, c: Int): Int = a + b + c

// Function value from method
val sumFunc: (Int, Int, Int) ⇒ Int = sum
// or f2 = sum _
val sumCurried: Int ⇒ Int ⇒ Int ⇒ Int = sumFunc.curried
val sumTupled: ((Int, Int, Int)) ⇒ Int = sumFunc.tupled
val sumFuncBis: (Int, Int, Int) ⇒ Int = Function.untupled(sumTupled)
// Partially applied function
val sum3: Int ⇒ Int = sumCurried(2)(1)

val tuple = (1,2,3)
sumTupled(tuple)  6: Int
sum3(2)  5: Int
```

High order functions

functions that take functions as arguments or return functions

```scala
def map[A, B](list: List[A], f: A ⇒ B): List[B] =
  for (element ← list) yield f(element)

val numbers: List[Int] = List(1, 2, 3, 4)

map(numbers, (i: Int) ⇒ i + 0.2d)  List(1.2, 2.2, 3.2, 4.2): scala.collection.immutable.List
```

```scala
// Recursive version without pattern matching
def map[A, B](list: List[A], f: A ⇒ B): List[B] =
  if (list.isEmpty) Nil
  else f(list.head) :: map(list.tail, f)
```

```scala
//Tail recursive version without pattern matching
def map[A, B](list: List[A], f: A ⇒ B): List[B] = {
  @tailrec
  def mapRec(acc: List[B], list: List[A], f: A ⇒ B): List[B] =
    if (list.isEmpty) acc
    else mapRec(acc :+ f(list.head),  list.tail, f)

  mapRec(List.empty[B], list, f)
}
```

High order functions

and collections API has plenty of them out of the box...

```
Collection[A]

//Method        //HOF                      //Result
map[B]          (A⇒B)                      Collection[B]
flatMap[B]      (A ⇒ Collection[B])         Collection[B]
reduce          ((A, A) ⇒ A)               A
foldLeft[B]     ((B, A) ⇒ B)               B
foldRight[B]    ((A, B) ⇒ B)               B
groupBy[K]      (A ⇒ K)                    Map[K, Collection[A]]
forAll          (A ⇒ Boolean)              Boolean
foreach         (A ⇒ Unit)                 Unit
takeWhile       (A ⇒ Boolean)              Collection[B]
dropWhile       (A ⇒ Boolean)              Collection[B]
span            (A ⇒ Boolean)              (Collection[B], Collection[B])
....
```

Methods are cool

Named parameters + default values

```
def greet(name:String, greeting:String = "Hello", exclamation:Boolean = false):String =
  s"$greeting $name" + (if(exclamation) "!" else "")

greet("John Snow")  Hello John Snow: java.lang.String
greet("John Snow", "You know nothing")  You know nothing John Snow: java.lang.String
greet("John Snow", greeting = "You know nothing")  You know nothing John Snow: java.lang.String
greet("Cersey", exclamation = true)  Hello Cersey!: java.lang.String
```

Multiple parameter lists

```
def map[A, B](l: A*)(f: A ⇒ B): List[B] = {
  for (element ← l.toList) yield f(element)
}

map(1, 2, 3 ,4) { i ⇒
  i + 0.2d
}
```

by-value vs. by-name parameters

```
def foo(a: ⇒ String, b: String) = {println(a); println(b); println(a); println(b);}

foo({println("Resolving a"); "printing a"}, {println("Resolving b"); "printing b"})  (): Unit
```

```
Resolving b
Resolving a
printing a
printing b
Resolving a
printing a
printing b
```

Lazy evaluation

```
final class Cons[+A](hd: A, tl: ⇒ Stream[A]) extends Stream[A] {
  ...
}
```

lazy val / var

```
val eagerResource: Int = init("eager")

lazy val lazyResource: Int = init("lazy")

def init(from: String): Int = {
  println(s"doing something expensive ... from $from call")
  0
}

println("Resources usage")  (): Unit

eagerResource  0: Int
lazyResource  0: Int
lazyResource  0: Int
```

```
doing something expensive ... from eager call
Resources usage
doing something expensive ... from lazy call
```

Immutability-first language

- val vs var

```
// aka final int foo = 1 in java
val foo = 1
foo = foo + 1
```

```
reassignment to val
```

```
var foo = 1
foo = foo + 1  (): Unit
foo  2: Int
```

- Function parameters are vals by default
- Scala Predef (object accessible in all scala compilation units) exposes immutable structures by default

Why immutability?

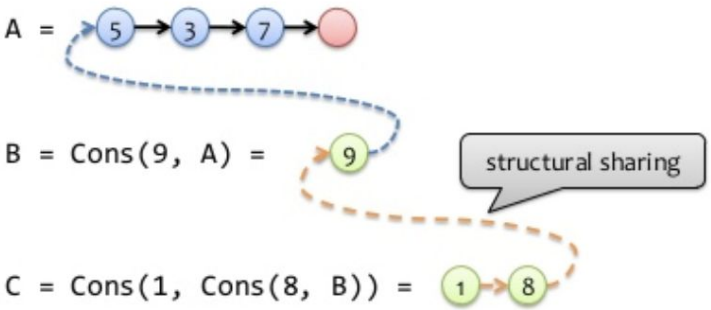
- It's easy to reason about the code locally
- Achieve code that has fewer defects and is easier to maintain
- It allows parallelism without fearing any thread safety issues (Writing mutable parallel code is hard and very error prone)

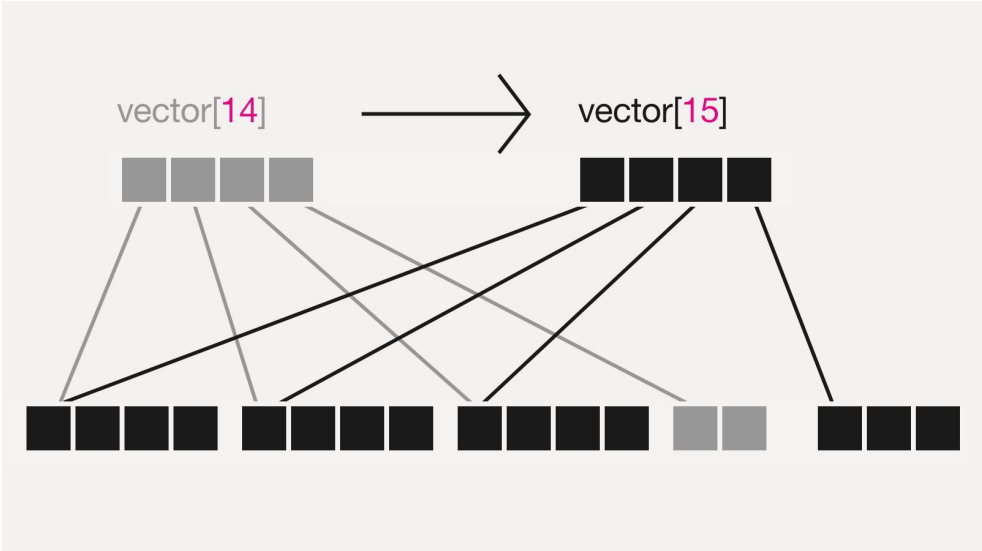
"Prefer vals, immutable objects, and methods without side effects. Reach for them first. Use vars, mutable objects, and methods with side effects when you have a specific need and justification for them."

--programming in scala--

Persistent data structures support multiple versions

Structural sharing





Collections performance

	head	tail	apply	update	prepend	append	insert
immutable							
`List`	C	C	L	L	C	L	-
`Stream`	C	C	L	L	C	L	-
`Vector`	eC	eC	eC	eC	eC	eC	-
`Stack`	C	C	L	L	C	L	L
`Queue`	aC	aC	L	L	C	C	-
`Range`	C	C	C	-	-	-	-
`String`	C	L	C	L	L	L	-

	lookup	add	remove	min
immutable				
`HashSet` / `HashMap`	eC	eC	eC	L
`TreeSet` / `TreeMap`	Log	Log	Log	Log
`BitSet`	C	L	L	eC[1]
`ListMap`	L	L	L	L

More objects...

```
object Configuration {
  val host = "localhost"
  val port = 8080
  def senderAddress(userType: Int) = ???
}

Configuration.host  localhost: java.lang.String
Configuration.port  8080: Int
```

- Singleton objects
- Static members
- Can extend other classes or traits

Classes

```
class MyClass(_a: String, _c: Boolean) {
  val a = _a
  private var c = _c
  def setC(other: Boolean): Unit = this.c = c
}

val myClass = new MyClass("a", true)

myClass.a  a: java.lang.String
myClass.c //NOP
myClass.setC(false)  (): Unit
```

```
class MyClass(val a: String, private var c: Boolean)
```

Companions forever

```scala
class Configuration(private val foo: List[String]) {
  val port = Configuration.port
}

object Configuration {
  val host = "localhost"
  private val port = 8080
  def merge(config: Configuration) = host :: config.foo
}
```

- Same file
- Share visibility
- implicits related with the CC in the CO

Case classes

```
case class MyClass(a: String, c: Boolean)
```

```
class MyClass(val a: String, val c: Boolean) {
  def ==(other: MyClass): Boolean = ??? //Equals
  def copy(a: String = this.a, c: Boolean = this.c): MyClass = new MyClass(a, c)
  override def hashCode: Int = ???
  override def toString: String = ???
}

object MyClass {
  def apply(a: String, c: Boolean): MyClass = new MyClass(a, c)
  def unapply(myClass: MyClass): Option[TupleN[ ... ]] = ???
}
```

Traits

- Like java 8 interfaces but with instance members
- Mixins (bring functionality, not just a contract)
- Can extend other classes or traits

```
abstract class A {
  val message: String
}
trait B extends A {
  val message = "I'm an instance of class B"
}
trait C extends A {
  def loudMessage = message.toUpperCase()
}
class D extends B with C

val d = new D
println(d.message)  // I'm an instance of class B  (): Unit
println(d.loudMessage)  // I'M AN INSTANCE OF CLASS B  (): Unit
```

Traits (Construction)

```
trait TBase                                {print("TBase⇒")}
trait T1 extends TBase                     {print("T1⇒")}
trait T2 extends TBase                     {print("T2⇒")}
trait T3 extends T1 with TBase             {print("T3⇒")}
class A extends T1 with T2 with T3 {print("A")}

val a = new A  //TBase⇒T1⇒T2⇒T3⇒A
```

TBase => T1 => ~~TBase~~ => T2 => ~~TBase~~ => ~~T1~~ => ~~TBase~~ => T3 => A

Traits (Linearization)

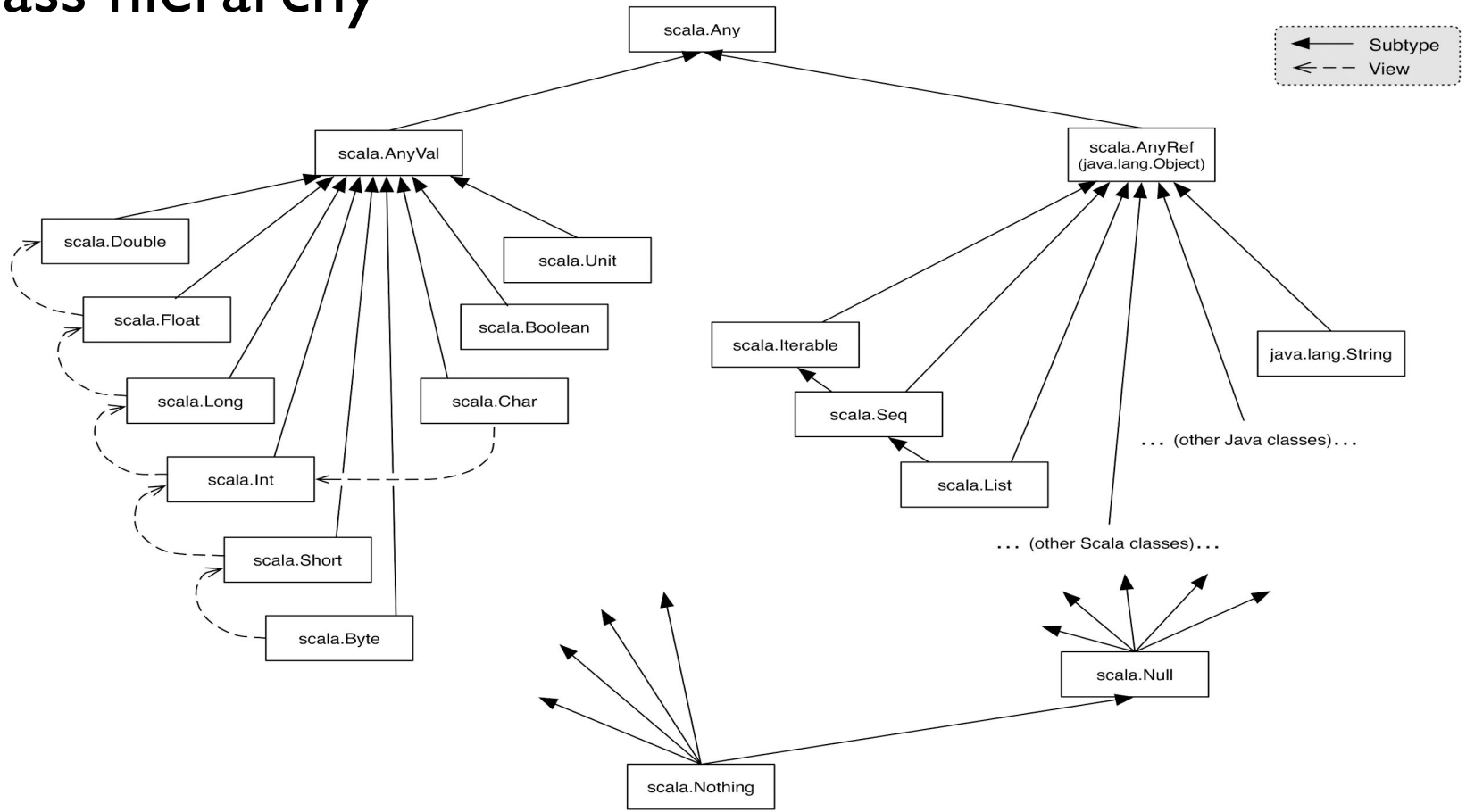
flatten the inheritance hierarchy graph

```
trait TBase                             {def foo: Unit = {print(s"⇒TBase")}}
trait T1 extends TBase                  {override def foo: Unit = {print(s"⇒T1");super.foo}}
trait T2 extends TBase                  {override def foo: Unit = {print(s"⇒T2");super.foo}}
trait T3 extends T1 with TBase          {override def foo: Unit = {print(s"⇒T3");super.foo}}
class A extends T1 with T2 with T3      {override def foo: Unit = {print(s"A");super.foo}}

val a = new A().foo  // A⇒T3⇒T2⇒T1⇒TBase
```

A => T3 => ~~TBase~~ => ~~T1~~ => ~~TBase~~ => T2 => ~~TBase~~ => T1 => TBase

Class hierarchy



Pattern matching

- Mechanism for checking a value against a pattern
- Can match against:
  - Values
  - Variables
  - Types
  - Sequences
  - Tuples
  - RegExp
  - Case classes
- Sometimes compiler can check for exhaustiveness
- Can combine matching with guards

Pattern matching

```scala
case class Person(id: String, tuple:(Int, Boolean), age: Int)

val persons = List(
  Person("xrl_23", (1, true), 25),
  Person("xrl_24", (1, true), 30),
  Person("xrl_25", (2, false), 50)
)

persons match {  first person is xrl_23: java.lang.String
  case Person("xrl_23", _, _) :: xs ⇒ "first person is xrl_23"
  case Person(id, _, _) :: xs if id == "xrl_23" ⇒ "first person is xrl_23"
  case List(_, Person(a, (_, true), _), _*) ⇒ s"match person $a with true value"
  case List(p1, p2, p3, p4) ⇒ "match a list of 4 people"
  case x :: Person(_, (_, true), age) :: Nil if age > 25 ⇒ "2nd person of a 2 length list true and > 25"
  case list @ (x :: Nil) ⇒ s"1 person list: $list"
  case Nil ⇒ "match Empty list constant"
  case xs ⇒ "else"
  //case _ ⇒ "another else, this is unreachable code. Compiler complains"
}
```

Pattern matching everywhere

```scala
val t = ("Victor & Rodrigo", true)
val(name, likesScala) = t
s"It's $likesScala that $name like scala"
// It's true that Victor & Rodrigo like scala

// Partial function
val print3: PartialFunction[Int, Unit] = {
  case 3 ⇒ println(3)
}
```

Amazing type constructors

- List[T]
- Option[T]
- Try[T, Throwable]
- Either[A, B]
- Future[T]

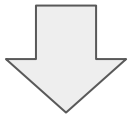
Types gide you. Compiler protects you from yourself!

Amazing type constructors (Option)

Avoid nulls! use Options!

Don’t try this at home

```
def findUser(userId: UUID): User = null
val user = findUser(UUID.randomUUID())
val result = s"Hello ${user.name}" //BOOOOOOOM
```



```
val users = List()
def findUser(userId: UUID): Option[User] = users.find(p ⇒ p.userId == userId)
val user = findUser(UUID.randomUUID())
val result =user.map(user ⇒ s"Hello ${user.name}").getOrElse("Who are you??")
```

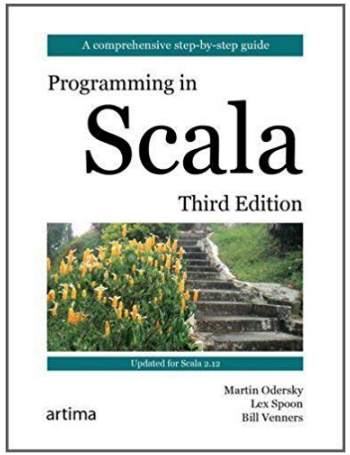
Amazing type constructors (Either)

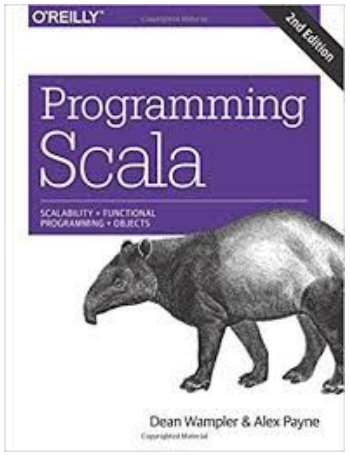
- Can be either A or B
- Is right biased

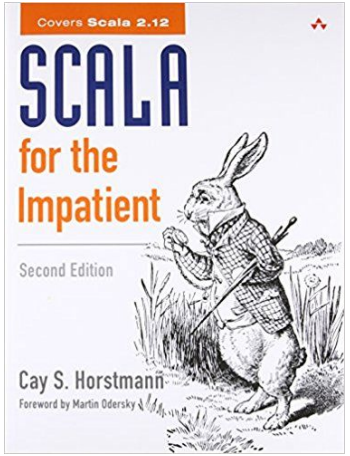
```
def findUser(userId: UUID): Either[UserNotFoundException, User]
val result = findUser(UUID.randomUUID()).map(_.name) match {
  case Right(name) ⇒ s"Hello $name"
  case Left(_) ⇒ s"Who are you??"
}
```

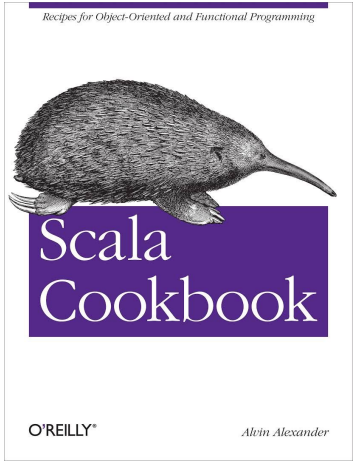
Next steps

- Functional programmint principles in scala (https://es.coursera.org/learn/progfun1)
- Scala exercices (https://www.scala-exercises.org/std_lib/asserts)









Play with the examples!

Find them at https://github.com/rodrigo-molina/intro-to-scala

Questions???

Want more???

Bonus track: Implicits!

Compiler doesn't give up!!

- Implicit conversion to an expected type

  Sees X but needs Y then look for conversion (X=>Y)

- Converting the receiver

  If X doesn't have some behaviour, then look for conversion (X=>Something that have that behaviour)

- Implicit parameters

  Pass implicit parameters if there are no specific ones

Implicit conversion (parameters)

```
implicit def double2Int(x: Double): Int = x.toInt

def foo(x: Int) = ???
foo(3.5)

//aka
foo(double2Int(3.5))
```

Implicit conversion (methods)

```
case class Person(name: String)
case class Employee(name: String) {
  def work: Unit = ()
}

implicit def personToEmployee(p: Person): Employee = Employee(p.name)
val p = Person("Victor").work
```

Implicit parameters

```
def foo(implicit a: Int) = s"$a received"
implicit val b: Int = 2

foo  //2 received
```

Implicit classes

- Extension methods with wrappers (ad-hoc polymorphism)

```
case class Person(name: String)

implicit class WorkerOps(value: Person) {
  def work(): Unit = ()
}

Person("Victor").work()
```