Introducción a Scala

David Suárez Caro (david.suarez@intelygenz.com)
Esteban Dorado Roldán (esteban.dorado@intelygenz.com)

Scala

Creado en 2003 por Martin Odersky

- Lenguaje Funcional y Orientado a Objetos Tipado estático
- Influencias: Java, Haskell Compila a JVM

Ecosistema

Página: https://www.scala-lang.org

Implementaciones:

Scala (compila a JVM), ScalaJS (compila a Javascript)

Versiones en funcionamiento:

2.9 (2011), 2.10 (2013) , 2.11 (2014)

Intérprete: scala Compilador: scalac

Construcción: sbt (http://www.scala-sbt.org)

Documentación: http://docs.scala-lang.org/

¿Conocéis algún proyecto hecho en Scala?

- Spark
- Akka
- Play framework
- ArnoIC (http://hackersome.com/p/lhartikk/ArnoldC)

Introducción

Índice:

- **Inferencia de tipos**
- **Sintaxis Básica**
- **Val/Var**
- **Ámbito de variables**
- **Estructuras de Control**
- **Clase Option**
- **Funciones Scala**
- **Programación Funcional**
- **Tipos predefinidos**
- **Todo es un objeto**
- **Funciones anónimas**
- **Classes/ Case Classes**
- **Traits**
- **Manejo de colecciones**
- **Libros**

Inferencia de tipos

- Chequeo estático de tipos
  - Tiempo de compilación
  - Objetivo: Si compila, entonces no hay error de tipos
- Sistema de inferencia de tipos
  - Muchas declaraciones de tipos = opcionales
  - Si se declara, se comprueba que está bien

```
val  x: Int =  +3          //>   x
     2                            :Int=5
val  y = 2 + 3             //>   y
                                  :Int=5

val  z:        =2 + 3      //>   error: type
     Boolean                      mismatch;
                                   found:Int(5)
```

Sintaxis Básica

- Nombres de variabes similar a Java
- Bloque de sentencias entre { } y separados por ;
- Valor del bloque = valor de último elemento
- Un bloque es una expresión

Sintaxis Básica

- Las sentencias tienen un ; al final
  - Pero en muchas ocasiones puede omitirse
  - El Sistema lo infiere

```
{ val x = 3
  x + x
}
```

≡

```
{ val x = 3 ;
  x + x
}
```

Sintaxis Básica

- Variables locales
  - Definiciones en un bloque sólo visibles dentro de él
  - Definiciones en un bloque tapan definiciones externas

```
val x = 3
def f(x: Int) = x + 1
val resultado = {
   val x = f(3)
      x * x } + x


println(resultad        //> (4*4)+3 = 19
o)
```

Sintaxis Básica

- Declaraciones de tipo después de variable

Java

```
Integer x = 0;
```

Scala

```
val x: Int =
0
```

- No es necesario return

Java

```
Integer suma(Integer a, Integer
b) {
  return a + b;
}
```

Scala

```
def suma(x:Int, y:Int) = x +
y
```

Estructuras de control

- If
- Match
- While
- For, foreach
- Try

Estructuras de control

- If es similar a otros lenguajes
  - Devuelve un valor

```
val mensaje =
    if (edad >= 18) "Puede votar"
    else "Es menor"
```

While, do...while

- Similares a Java

```
def mcd(x:Int,y:Int):Int
  ={  var a = x
   var b = y
   while(a != 0)
   {
     val temp = a
     a = b % a

     b = temp
   }
   b
}
```

While, do...while

- Los bucles While suelen ser imperativos.
  - Pueden re-escribirse mediante recursividad

```
def mcd(x:Int,y:Int):Int = {
  if (x == 0) y
  else mcd(y % x,x)
}
```

Bucles while e iteradores

- Estilo imperativo

```
def mediaEdad(personas: List[Persona]): Double = {
    var suma = 0
    val it = personas.iterator
    while (it.hasNext) {
      val persona = it.next()
      suma += persona.edad
    }
    suma / personas.length
}
```

- Pueden re-escribirse con estilo funcional

```
def mediaEdad(personas: List[Persona]): Double = {
    personas.map(_.edad).sum / personas.length
}
```

Encaje de patrones

- Expresión match

```
dia      {
match    "Sabado" => println("Fiesta")
     case
     case "Domingo => println("Dormir")
          "
     case _        => println("Programar en
 }                    Scala")
```

- Expresión match devuelve un valor

```
val mensaje = dia match {
     case "Sabado"   => "Fiesta"
     case "Domingo"  => "Dormir"
     case _          => "Programar en
     Scala"
   }
```

Bucles for

- Contienen:
  - Generadores (suelen ser colecciones)
  - Filtros (condiciones)
  - Yield: valores que se devuelven

```
def pares(m:Int,n:Int): List[Int] =
       for (i <- List.range(m,n) if i % 2 == 0) yield i
println(pares(0,10)) //>List(0,2,4,6,8,10)
```

- Expresión match devuelve un valor

```
for (i<-1 to 4)
      print("x" + i)          //> x1 x2 x3 x4
```

Excepciones

- Try…throw...catch...similar a Java

```
def divide(m:Int, n:Int) : Int = {
     if (n == 0)
       throw new RuntimeException("division por 0")
     else m / n
 }
   try {
    println("5/4 = " + divide(5,4))
         println("5/0 = " +
  divide(5,0))
   } catch {
     case e: Exception =>
      println("Error:" + e.getMessage)
   } finally {
     println("Fin")
```

Clase Option

- Option permite definir funciones parciales
  - Puede utilizarse para evitar uso de excepciones

```
def divide(m:Int, n:Int) : Option[Int] = {
 if (n == 0)
  None
 else
  Some(m / n)
}
```

```
println("5/4  " + divide(5,4))    //> Some(1)
      =
println("5/0  " + divide(5,0))    //> None
      =
```

Funciones en Scala

- Varias formas de declarar funciones

```
def suma(x:Int,y:Int) = x + y
```

```
def suma(x:Int,y:Int): Int = x + y
```

```
def suma(x:Int,y:Int): Int = {
   return x + y
}
```

- Procedimiento = función que devuelve valor de tipo Unit

```
def suma3(x:Int,y:Int)
{
   println(x + y)
}
```

```
def suma4(x:Int,y:Int):Unit = {
   println(x + y)
}
```

Programación funcional

- Funciones como valores

```
val suma = (x:Int,y:Int) => x + y
```

- Funciones de orden superior

```
def suma3(x:Int) = x + 3
def aplica2(f: Int => Int, x: Int) = f(f(x))

println(aplica2(suma3,2))                    //> 8 (2+3)+3
println(aplica2((x:Int) => x *               //> 16 (2*2)*2*2
x,2))
```

Tipos predefinidos

- Numéricos: Int, BigInt, Float, Double
- Boolean
- String
- Rangos
- Tuplas
- Regex
- Null

Números, booleanos y caracteres

- Similares a Java pero sin tipos primitivos
  - Byte, Short, Int, Long, Float, Double
  - Rango disponible mediante MinValue, MaxValue
    - Ej. Int.MinValue
  - También disponibles: BigInt, BigDecimal
    - Ej. “234”.toInt
- Boolean: valores true, false
- Char: representa caracteres

Strings

- Similares a Java
  - Comparación mediante == (equals en Java)
- Sintaxis """ para cadenas multilínea
- Numerosas utilidades en StringOps

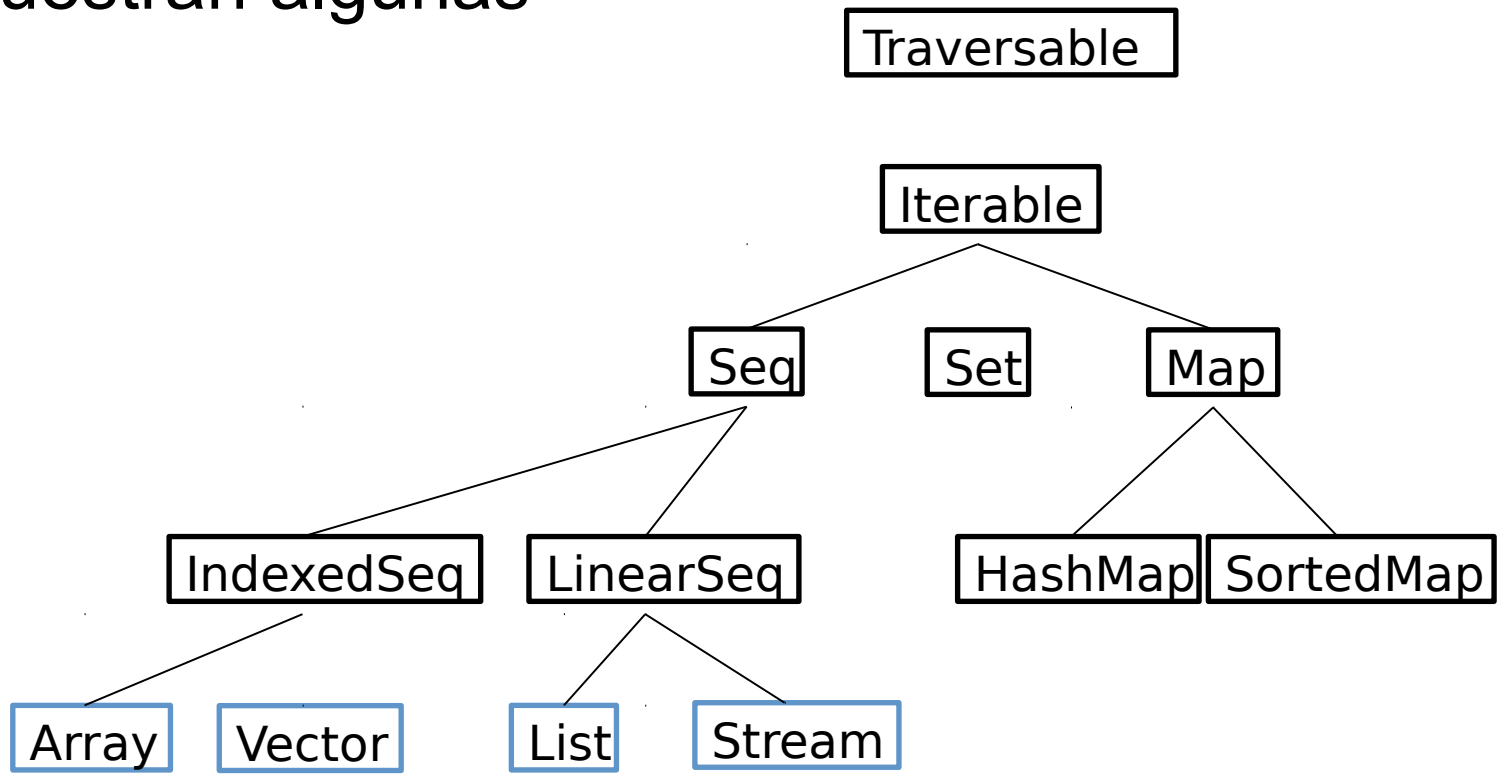
Rangos

- Range (min, max) crea rango entre min y max

```
val ceroDiez = Range(1,10)

println (ceroDiez.contains(5))                    //> true

for (i <- ceroDiez if i % 3 == 0) print(s"$i ")   //> 3 6 9
```

Colecciones

- Jerarquía de colecciones
  - Solo se muestran algunas



Listas

- Construcción básica mediante :: y Nil

```
val x = 1 :: 2 :: 3 ::
Nil  val y = List(1,2,3)

println(x ==                          //> true
y)
```

Vectores

- Operación de indexación muy rápida

```
val frutas = Vector("Peras", "Limones", "Naranjas")
println(frutas(1))                                  //>
Limones  println((frutas :+ "Kiwis").length)//> 4
```

Maps

- Arrays asociativos (Tablas Hash)

```
val notas = Map("Jose" -> 5.7, "Luis" -> 7.8)

for ((n,v) <- notas)
  println (s"${n} tiene un ${v}")
```

```
Jose tiene un 5.7
Luis tiene un 7.8
```

Todo es un objeto

- Los números son objetos:
  - 1 + 2 * 3 / x <==> (1).+(((2).*(3))./(x))
  - Por eso ningún nombre puede empezar por un numero.
- Consideraciones
  - (Double)1.+(2) != (Int)(1).+(2)
- Cosas raras
  - 1.to(10).foreach(i=>println(i))
  - For i=1 to i<=10

Callback: Las funciones son objectos

```
object Timer {
   def oncePerSecond(callback: () => Unit) {
      while (true) { callback(); Thread sleep 1000 }
   }
   def timeFlies() {
      println("time flies like an arrow...")
   }
   def   main(args:   Array[String {
     oncePerSecond(timeFlies)
   }
}
```

Funciones Anónimas

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
    println("time flies like an
    arrow..."))
  }
}
```

Clases

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
object ComplexNumbers {
  def main(args:  Array[String]) {
   val c = new Complex(1.2, 3.4)
   println("imaginary part: " +c.im())
  }
}
```

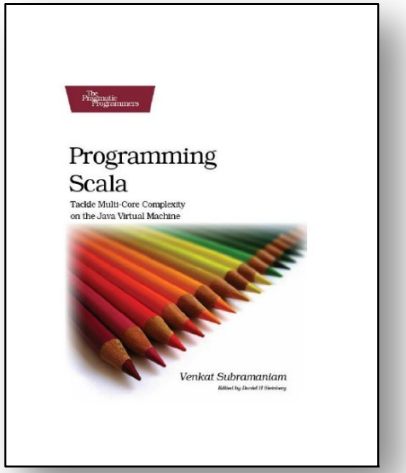
Traits

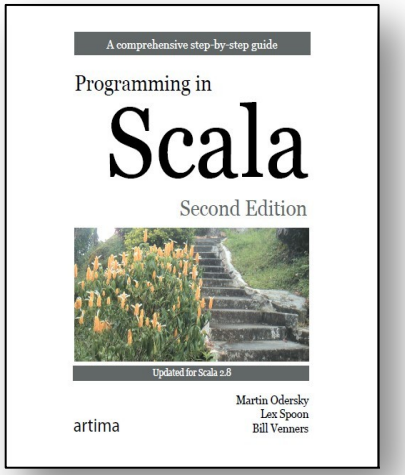
```
trait Saludador {
 def saluda(nombre: String) {
  println("Hola " + nombre + ", soy " + this.toString)
 }
}

case class Persona(nombre:String, edad:Int) extends Saludador
case class Coche(marca:String) extends Saludador

val p206= Coche(”Peugeot 206")
val javi = Persona(”Javier",31)

p206.saluda("Pepe")     //> Hola Pepe, soy Coche(Peugeot 206)
javi.saluda("Pepe")     //> Hola Pepe, soy Persona(Javier,31)
```

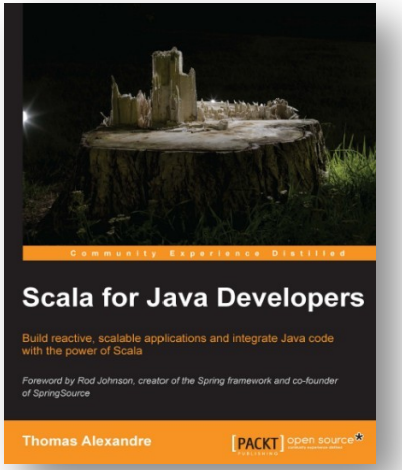
Manejo de Colecciones

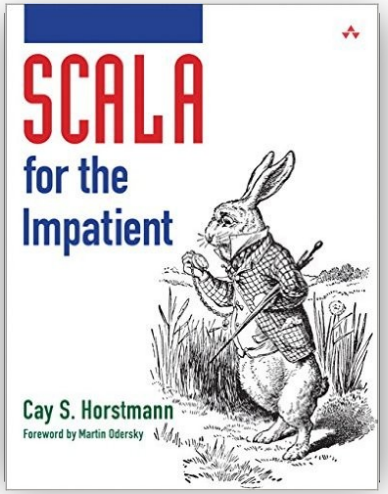
```
val list =       2, 3)
List(1,
list.foreach(println)   /// same   1, 2  3
=>
list.map( => x  +  2) /  return  a new List(3 4, 5)
x                     /  s              ,
list.map( + 2) // same
list.filter(x => x % 2 == 1)// returns a new List(1, 3)
list.filter(_ %2 == 1) // same
list.reduce((x y)   x  +   y) //   => 6
,               =>   same
list.reduce(_  _) /
+               /
list.foldLeft(  (_*_   // =>     6
1)                  )
list.foldLeft(  ( +   // =>     6
```

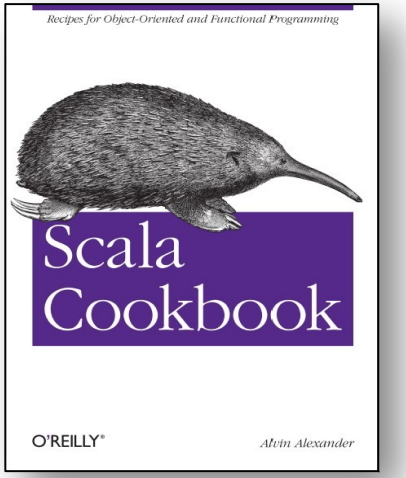
Libros

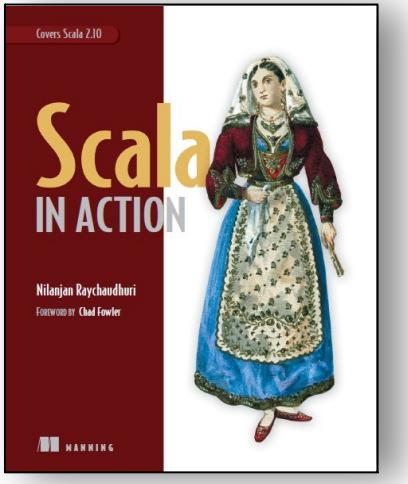

Programming Scala
Tackle Multi-Core Complexity on the Java Virtual Machine
Venkat Subramaniam

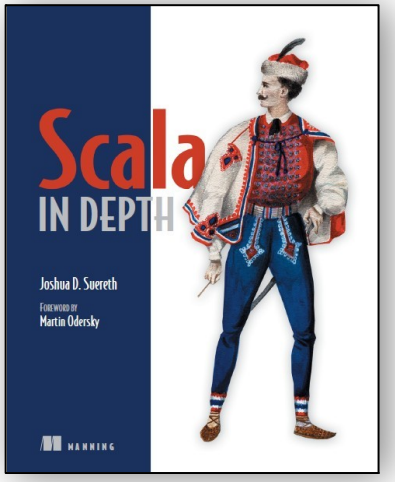

A comprehensive step-by-step guide
Programming in
Scala
Second Edition
Updated for Scala 2.8
Martin Odersky
Lex Spoon
Bill Venners
artima

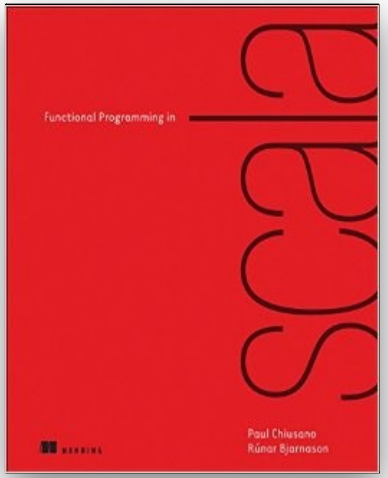

Community Experience Distilled
Scala for Java Developers
Build reactive, scalable applications and integrate Java code with the power of Scala
Foreword by Rod Johnson, creator of the Spring framework and co-founder of SpringSource
Thomas Alexandre
PACKT open source


SCALA
for the Impatient
Cay S. Horstmann
Foreword by Martin Odersky

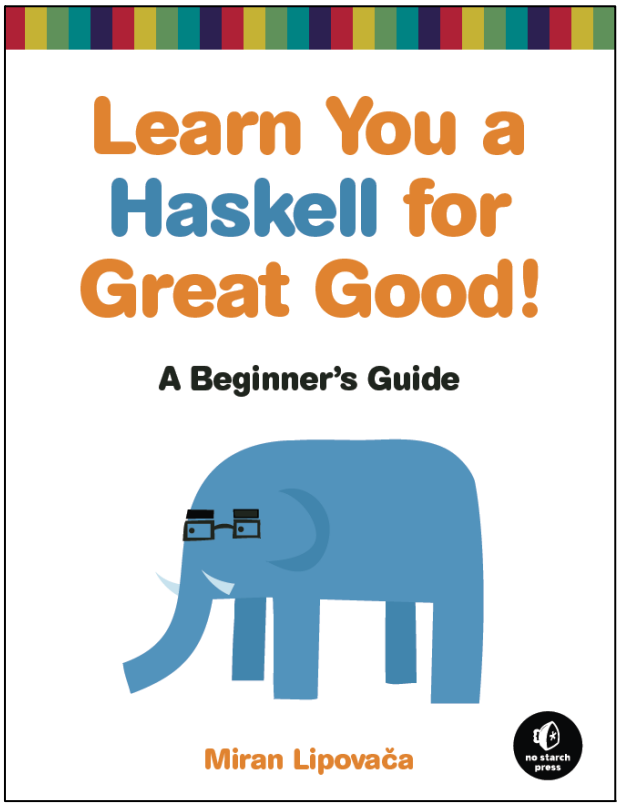

Recipes for Object-Oriented and Functional Programming
Scala Cookbook
O'REILLY
Alvin Alexander


Covers Scala 2.10
Scala
IN ACTION
Nilanjan Raychaudhuri
Foreword by Chad Fowler
MANNING


Scala
IN DEPTH
Joshua D. Suereth
Foreword by
Martin Odersky
MANNING


Functional Programming in
Scala
Paul Chiusano
Rúnar Bjarnason
MANNING

Libro Programación Funcional



http://learnyouahaskell.com/chapters

Ejercicios Scala

http://scala-exercises.47deg.com/

Instalación

En solo 3 pasos

- Descárgalo de la pagína (http://scala-lang.org/download/)
- Descomprímelo en la carpeta que tu prefieras (si es linux no es una mala idea hacerlo en la carpeta /opt)
- Añade o actualiza estas dos variables de entorno:
  - $SCALA_HOME=/opt/scala/last
  - $PATH=$PATH:$SCALA_HOME/bin
- YA ESTA!!! Solo debemos ejecutar el comando “scala” y aparecerá la consola

Hello World

```
object HelloWorld {
   def main(args: Array[String]) {
      println("Hello, world!")
   }
}
```

Compílalo!

- Guarda el texto en un fichero HelloWorld.scala
- Compila con el comando “scalac HelloWorld.scala”
- Ejecuta con el comando “scala HelloWorld”

```
object HelloWorld  extends App
   {  println("Hello, world!")
}
```

Script it!

- Se puede crear scripts para bash utilizando el interprete de la consola.
  En el caso del HelloWorld el código sería el siguiente.

```
#!/bin/sh
exec scala     "$0" "$@"
!#
object HelloWorld extends App
   {  println("Hello, world!")
}
HelloWorld.main(args)

          >./script.sh
```