Scala

FUNDAMENTOS TECNOLÓGICOS PARA BIG DATA

Isaac Esau Rubio Torres
2018

¿Qué es Scala?

- Es un lenguaje de programación de propósito general diseñado para expresar patrones comunes de programación de forma concisa y elegante.
- Se integran las características de los lenguajes orientados a objetos y funcional.

- Scala no es una extensión de Java, pero es totalmente interoperable con él.

- Scala se traduce a *bytecodes* Java y la eficiencia de los programas compilados por lo general es igual que Java.

Historia

- Scala fue diseñado por Martin Odersky y su grupo de la Escuela Politécnica Federal de Lausana (Suiza)

- Odersky tenía como objetivo combinar la programación funcional y la programación orientada a objetos

- Se inició en 2001 y la primera versión que se hizo pública fue en 2003. En 2006, se lanzó una segunda versión conocida como Scala v2.12

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), 2.12 (2016)

Intérprete: scala Compilador: scalac

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

Búsqueda: `http://scalex.org/`

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

Hola Mundo en Scala

```
object Saluda  {

  def main(args: Array[String]) =
       println("Hola desde Scala!")

}
```

holaMundo.scala

Varias formas de ejecutarlo:

Intérprete: scala

Compilador: scalac

```
> scala holaMundo.scala
Hola desde Scala!

> scalac
holaMundo.scala

> scala holaMundo
Hola desde Scala!
```

“Hola Mundo” en Scala

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

- **object** HelloWorld **extends** application { println("Hello world!"); }

Conceptos básicos

Lenguaje funcional y Orientado a Objetos

Diferencia entre valores y variables

```
val x = 5   // x  : Int = 5
println(x)    // 5
x = x + 1     // error: reassignment to
val
```

```
var x = 6     // x  : Int = 6
x = x + 1
println(x)
// 7
```

Chequeo e 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 = 2 + 3          // x : Int = 5 val y =
2 + 3                 // y : Int = 5 val z: Boolean =
2 + 3    // error: type mismatch;
                                  found     :  Int(5)
                                  required: Boolean
```

Sintaxis básica

Nombres de variables 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 }
```

NOTA: A veces se producen mensajes de error extraños *(unexpected ;)* Cuidado con expresiones del tipo:

```
val prueba = uno +
  dos
```

```
val prueba = uno + dos ;
```

```
val prueba
  = uno +
  dos
```

```
val prueba = uno ; + dos ;
```

Sintaxis básica

Todo son expresiones

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

Numerosas simplificaciones

( ) opcionales con métodos de 0 ó 1 argumento

```
2 + 3 == 2.+(3)
```

. opcional (operador posfijo)

```
juan.salud
a()
juan.salud
```

```
juan.saluda("Pepe")
juan saluda("pepe")
juan saluda "pepe"
```

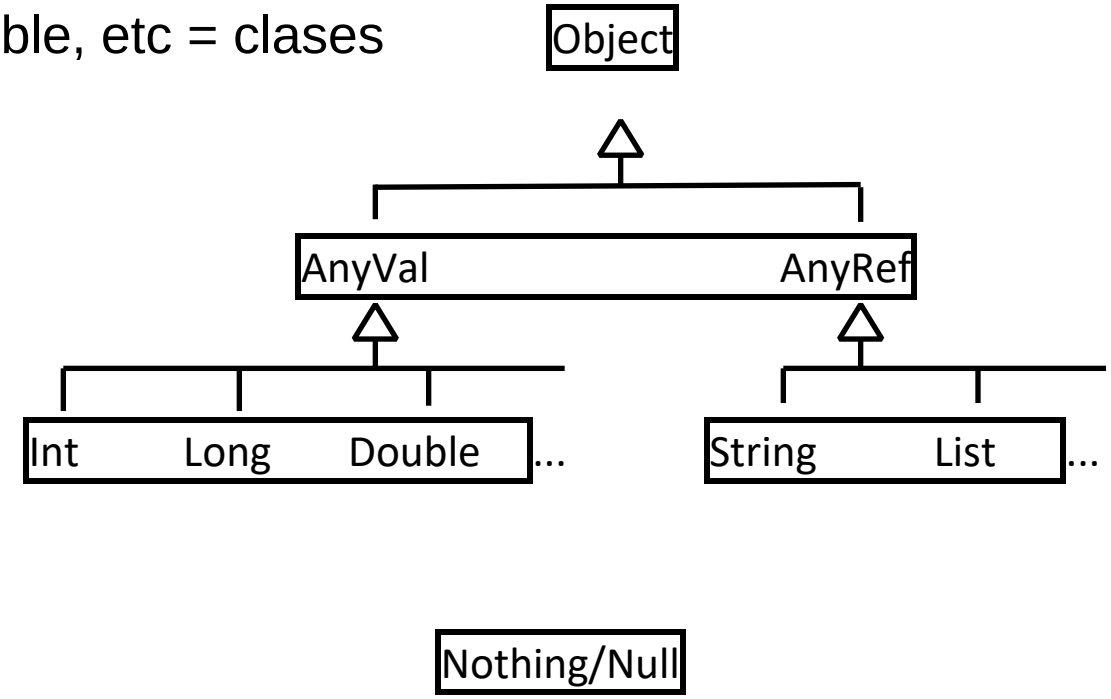
```
a juan
saluda
```

Objetos y clases

Herencia universal de **Object**

No hay tipos primitivos

Int, Boolean, Float, Double, etc = clases



Tipos predefinidos

Numéricos: Int, BigInt, Float, Double

Boolean

String

Rangos

Tuplas

Regex

Null

Any, AnyVal, AnyRef

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`

Conversión mediante `toX` Ej. `"234".toInt`

`Boolean:` valores `true`, `false`

`Char:` representa caracteres

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
```

Ámbito de variables

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(resultado)
// ???
```

¿Qué imprime?

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
}
```

NOTA: 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
}
```

NOTA: Puede re-escribirse con estilo funcional:

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

Encaje de patrones

Expresión match

```
dia   match   {   case   "Sabado"   =>
     println("Fiesta")  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)
```

Si no hayyield se devuelve Unit

Similar a bucles convencionales de otros lenguajes

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

Foreach

```
scala> val names = Vector("Bob", "Fred", "Joe", "Julia", "Kim")
names: Vector[java.lang.String] = Vector(Bob, Fred, Joe, Julia, Kim)

scala> for (name <- names) println(name)
Bob
Fred
Joe
Julia
Kim
```

```
scala> val names = Vector("Bob", "Fred", "Joe", "Julia", "Kim")
names: Vector[java.lang.String] = Vector(Bob, Fred, Joe, Julia, Kim)

scala> for (name <- names if name.startsWith("J"))
     | println(name)
Joe
Julia
```

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
println(aplica2((x:Int) => x * x,2))     // 16
```

Método map (programación funcional)

```
scala> val frutas = Seq("manzana", "platano", "naranja")
fruits: Seq[java.lang.String] = List(manzana, platano, naranja)

scala> frutas.map(_.toUpperCase)
res0: Seq[java.lang.String] = List(MANZANA, PLATANO, NARANJA)

scala> frutas.flatMap(_.toUpperCase)
res1: Seq[Char] = List(M, A, N, Z, A, N, A, P, L, A, T, A, N, O, N, A, R, A,
N, G, A)
```

```
scala> val list = List(1,2,3,4,5)
list: List[Int] = List(1, 2, 3, 4, 5)

scala> def g(v:Int) = List(v-1, v, v+1)
g: (v: Int)List[Int]

scala> list.map(x => g(x))
res0: List[List[Int]] = List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4),
List(3, 4, 5), List(4, 5, 6))

scala> list.flatMap(x => g(x))
res1: List[Int] = List(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6)
```

Método reduce (programación funcional)

```
// Sum.scala

val v = Vector(1, 10 , 100, 1000)
var sum = 0
v.foreach(x => sum += x)


// Reduce.scala

val v = Vector(1, 10, 100, 1000)
v.reduce((sum, n) => sum + n)
```

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")
 }
```

```
5/4 = 1
Error: division por 0
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
```

NOTA: La clase `Try` puede también utilizarse.

ScalaTest

Crear los siguientes métodos `par(n)` compruebe si un número es par `fact(n)` devuelve el factorial de un nº

Ejercicio

Calcular los factores primos de un número Ejemplo:

```
factores 1 = [1]
factores 2 = [1,2]
factores 3 = [1,3]
factores 4 = [1,2,2]
factores 5 = [1,5]
factores 6 = [1,2,3]
...
```

Declaración de Objetos

`object` permite definir objetos *singleton*

```
object  juan  {
  var edad = 34
  val nombre = "Juan Manuel"

  def crece() { this.edad += 1

  } def getEdad():Int =

  this.edad

  def masViejo(otro: { def getEdad(): Int}) =
this.edad > otro.getEdad }
```

NOTA
Scalaes un lenguaje basado en clases
En realidad, `object` crea un objeto singleton de una claseJuan

Clases: Estilo imperativo

Mediante `class`

```
class Persona(nombre: String) {
  private var edad:Int = 5
  def crece(): Unit = {
    this.edad += 1
  }
  def masViejo(otro: Persona): Boolean = {
    this.edad > otro.getEdad
  }
  def getEdad() = edad
}
```

Objetos mutables

edad es una variable que se actualiza

```
val juan = new Persona("Juan Manuel") juan.crece
println (juan.getEdad)

val pepe = new Persona("Jose Luis")
println(pepe.masViejo(juan))     // false
```

Clases: Estilo funcional

Objetos inmutables

Operaciones de escritura devuelven nuevos objetos

```
class Persona(nombre: String, edad: Int) {
  def crece(): Persona = {
   new Persona(nombre, this.edad + 1)
  }
  def masViejo(otro: Persona): Boolean = {
    this.edad > otro.getEdad
  }
  def getEdad() = edad
}
```

```
val juan = new Persona("Juan Manuel", 34)
println (juan.getEdad)                    //34
val juan1 = juan.crece println
(juan1.getEdad)                  // 35
println (juan.getEdad)                    //
34
```

Herencia

Mediante `extends`

```
class Usuario(nombre: String,
              edad:Int,
              email: String) extends Persona(nombre,edad)
              {
  def   login(email:   String):   Boolean   =   {
    this.email == email
  }
}
```

```
def sumaEdades(personas:List[Persona]): Int = {
    personas.map(_.getEdad).sum
}
```

```
val juan = new Persona("Juan", 34) val luis = new
Usuario("Luis", 20, "luis@mail.com")
println(sumaEdades(List(juan,luis)))                    //
54
```

Ejercicio: Figuras

Crear una clase Figura con 2 atributos (x,y) Método que permita mover la figura

Crear una clase Rect para representar Rectángulos Atributos a (ancho) y b (altura)

Crear una clase Circulo para representar Círculos Atributo r (radio)

Crear método area para calcular el area

Crear método areas que calcula el area de una lista de figuras

Strings

Similares a Java

- Comparación mediante == (equals en Java)

Sintaxis """ para cadenas multilínea

Numerosas utilidades en StringOps

http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List

Strings

Interpolación/sustitución

```
valnombre ="Ronaldo"
valedad =40 valpeso
=184.5

//      imprime      "Hola,      Ronaldo"
println(s"Hola, $nombre")

// imprime "Ronaldo tiene 40 años y pesa 184.5 kilos."
println(f"$nombre tiene $edad años y pesa $peso%.1f
kilos.")

// interpolación 'raw'
println(raw"foo\nbar")
```

```
%c  character
%d  decimal (integer) name (base 10)
%e  exponential floating-point number
%f  floating-point number
%i  integer (base 10)
%o  octal number (base 8)
%s  a string of characters
%u  unsigned decimal (integer) number
%x  number in hexadecimal (base 16)
%%  print a percent sign
\%  print a percent sign
```

Strings

Substrings

```
val s ="pokemon go"
s.substring(0,3)                // "pok"
s.substring(0,4)                // "poke"
s.substring(1,5)                // "ope "
s.substring(1,6)                // "opemon "
s.substring(0, s.length-1)      // "pokemon g"
s.substring(0, s.length)        // “pokemon go"
```

Strings Expresiones

regulares

```
// creamos una expresión regular con '.r' valnumPattern
="[0-9]+".r
valaddress ="123 Main Street" // "123 Main Street"
valmatch1=numPattern.findFirstIn(address)  // Some(123)

// creamos na expresión regular como clase
import           scala.util.matching.Regex
valnumPattern          =newRegex("[0-9]+")
valaddress ="123 Main Street Unit 639"
valmatches =numPattern.findAllIn(address) // non-empty iterator valmatches
=numPattern.findAllIn(address).toArray // Array(123, 639)
```

```
scala> val pattern = "(\\d+) ([A-Za-z]+) (\\d+)".r
pattern: scala.util.matching.Regex = (\\d+) ([A-Za-z]+) (\\d+)

scala> val pattern(dia, nes, anno) = “22 Junio
2012" dia: String = 22 mes: String = Junio anno:
String = 2012
```

Strings

Arrays a strings

```
vala =Array(1,2,3)
a.mkString                          // "123"
```

```
a.mkString(",")                 // "1,2,3"
a.mkString(" ")                 // "1 2 3"
a.mkString("(", ",", ")")       // "(1,2,3)"
```

Más ejemplos en: http://alvinalexander.com/scala/scalastring-examples-collection-cheat-sheet

Rangos

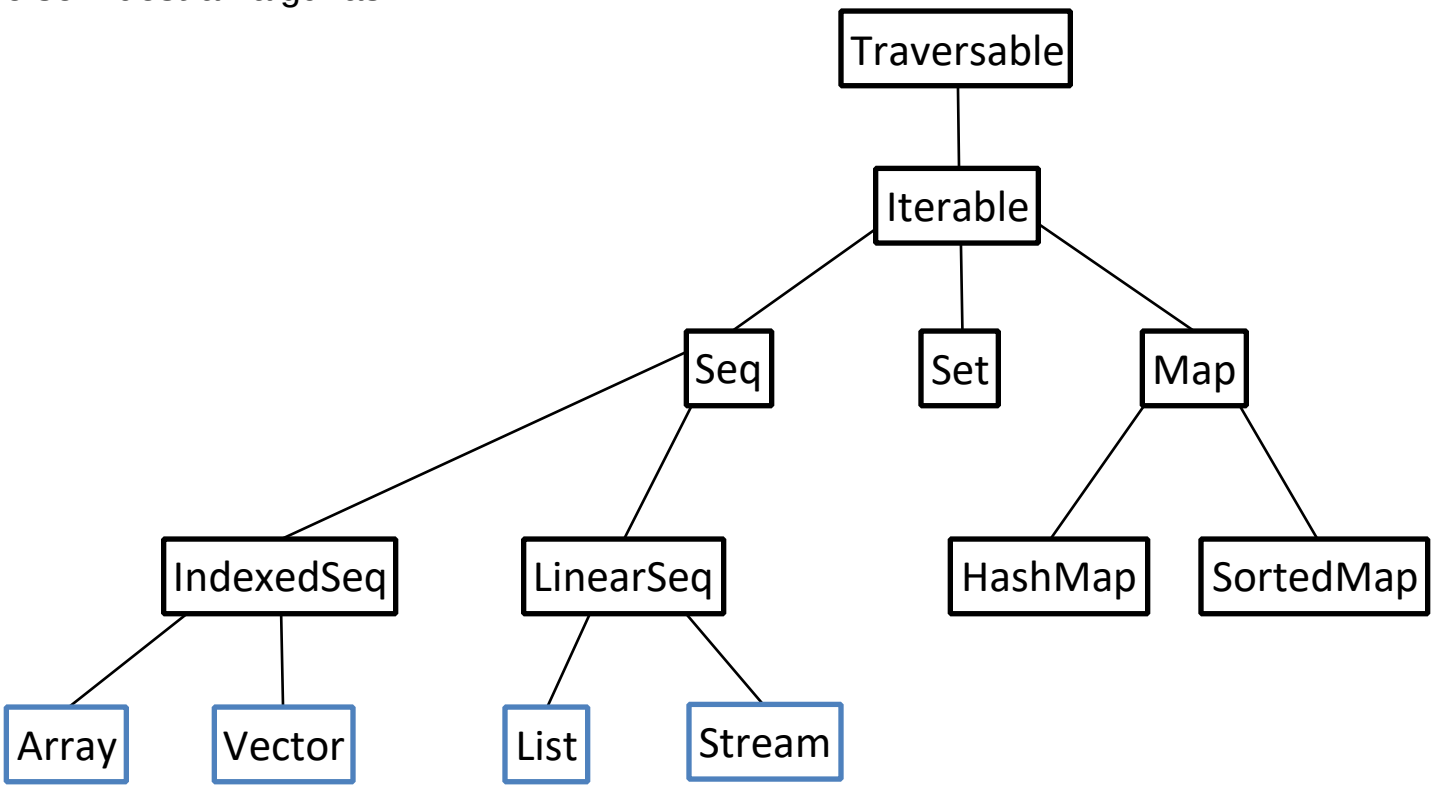
Range(min,max) crea rango entre min y max

```
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 == y)                    // true
```

Listas

```
// Lista de cadenas                        ", "naranjas", "peras")
val fruit: List[String] = List("manzana

// Lista de interos
val nums: List[Int] = List(1, 2, 3, 4)

// Lista vacía
val empty: List[Nothing] = List()
```

```
// 2D
val dim: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1)    )
```

http://www.tutorialspoint.com/scala/scala_lists.htm

Listas

```
object Test { def main(args: Array[String]){ val frutas =
   "manzanas" :: ("naranjas":: ("peras" :: Nil)) val nums = Nil
   println("Cabeza de frutas : " + frutas.head ) println("Cola de
   frutas : " + frutas.tail ) println("Frutas vacio : " +
   frutas.isEmpty ) println("Nums vacio : " + nums.isEmpty ) }
}
```

```
object Test { def main(args: Array[String]){ val frutas1 =
   "manzanas":: ("naranjas":: ("peras" :: Nil)) val frutas2 =
   "mangos" :: ("platanos":: Nil) var frutas = frutas1 ::: frutas2
   // ::: igual ++ println("frutas1 ::: frutas2 : " + frutas )
   frutas = frutas1.:::(frutas2)
      println("frutas1.:::(frutas2) : " + frutas ) frutas
      = List.concat(frutas1,frutas2)
      println("List.concat(frutas1, frutas2) : " + frutas  )
   }
}
```

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 tieneun
      5.7
Luis  tieneun
      7.8
```

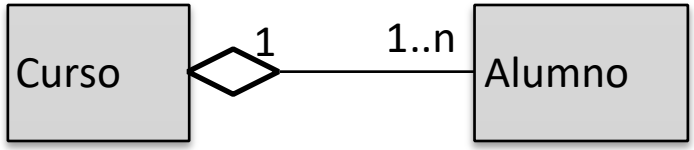
Folds

```
scala>  Array(1,2,3).foldLeft(List[Int]())((b,a)  =>  b  :+  a)
res15: List[Int] = List(1, 2, 3)

scala>  Array(1,2,3).foldRight(List[Int]())((a,b)  =>  a  :+  b)
res16: List[Int] = List(1, 2, 3)

scala> val numbers = List(5, 4, 8, 6, 2)
scala> numbers.fold(0) { (a, i) => a + i
        }
```

Ejercicio con agregación

Modelar cursos con alumnos Una clase curso compuesta por:

- Nombre del curso
- Lista de alumnos



Una clase alumno compuesta por

- id del alumno
- nota del alumno

Definir métodos de curso:

- getNota(id)
- ponNota(id,nota)
- media

Objeto acompañante objecto con mismo nombre que una clase ó trait

- Es un objeto singleton (una única instancia)
- Tiene acceso a campos/métodos privados
- Uso habitual: métodos/atributos estáticos
  - En Scala no hay `static`
- Método `apply()` puede usarse como factoría
- Método `unapply()` puede para extraer elementos

Objeto acompañante

```
class Persona(nombre:String, edad:Int) {
 def getEdad = edad def masViejo(otro:Persona) =
 this.edad > otro.getEdad
}

object Persona {
 def apply(nombre:String): Persona = new Persona(nombre,15)
def   apply(edad:Int):   Persona   =   new   Persona("Sin
nombre",edad) }
```

```
val juan = Persona("Juan")
val x   = Persona(10)

println(juan.masViejo(x))                    // true
```

Objeto acompañante

```
object Doble {
  def apply(x: Int) = x*2 def unapply(z: Int) = if
  (z%2==0) Some(z/2) else None
}
val x = Doble(21)
```

```
x match {
  case Doble(y) => println(x+" es el doble de "+y)
  case _ => println("x es impar")
}
```

case Classes

Permite simplificar declaraciones

```
case class Persona(nombre:String, edad:Int)
```

Objetos funcionales (inmutables)

Genera getters de los campos

Crea objeto acompañante (fabricar instancias sin `new`)

Genera `equals,hashCode,toString`

```
val juan = Persona("Juan", 34)  // Persona(Juan,34)
val luis = Persona("Luis", 23)  // Persona(Luis,23)

println(juan.nombre)            // "Juan"
println(juan == luis)           // false
```

Nota

case Classes permiten crearpatrón ValueObject:
http://en.wikipedia.org/wiki/Value_object

Comparación con Java

Java

Scala

```
case class Persona(nombre:String,
edad:Int)
```

```
public  class  Persona  {
  private          String
  nombre;         private
  Integer edad;

  public String  getNombre() { return nombre; }
  public Integer getEdad() { return edad; }

  @Override
  public boolean equals(Object otro) {
    if (otro == this) return true;
    if (!(otro instanceof Persona)) return false;
    Persona p = (Persona) otro;
    return p.nombre == nombre && p.edad == edad;
  }

  @Override
  public int hashCode(){ int result = 17; result = 31* result +
    (nombre !=null ? nombre.hashCode() : 0); result = 31* result
    + (edad !=null ? edad.hashCode() : 0); return result;
  }

  @Override public String
  toString() {
   return
    String.format("Persona[name=%s,birthdate=%d]", nombre,
    edad);
  }
}
```

Copia de clases

```
scala> case class Empleado(nombre: String, oficina: String, rol: String)
defined class Empleado
```

```
scala> val pepe = Empleado("Pepe", "205", "Analista")
pepe: Empleado = Empleado(Pepe,205,Analista)

scala> val maria = pepe.copy(nombre="Maria")
maria: Empleado = Empleado(Maria,205,Analista)
```

Imprimir clases

```
// Animals.scala

//    Creamos    algunas

clases:   class   Giraffe

class Bear class Hippo

// Creamos algunos
objetos:

val g1 = new Giraffe
valg2=newGiraffe
valb=newBear
valh=newHippo
println(g1)
println(g2)
println(h)
println(b)
```

```
Main$$anon$1$Giraffe@53f64158

Main$$anon$1$Giraffe@4c3c2378

Main$$anon$1$Hippo@3cc262

Main$$anon$1$Bear@14fdb00d
```

traits

Reutilización de comportamiento

```
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 r21 = Coche("Renault XXI") val
juan = Persona("Juan Manuel",34)

r21.saluda("Pepe")   //> Hola Pepe, soy Coche(Renault XXI)
juan.saluda("Pepe")  //> Hola Pepe, soy Persona(Juan
Manuel,34)
```

Ejercicio Hashes y Arrays

Corregir exámenes. Aciertos: +1, fallos: -0.25

```
[ {"pregunta" => 1, "correcta" => "a"},
  {"pregunta" => 2, "correcta" => "b"}]
```

```
[ {"alumno" => 2456,
   "respuestas" => [{ "pregunta" => 1, "respuesta" => "a"},
                    { "pregunta" => 2, "respuesta" => "b"}]},
  {"alumno" => 4321,
   "respuestas" => [{ "pregunta" => 1, "respuesta" => "b"},
                    { "pregunta" => 2, "respuesta" => "b"}]}]
```

```
[ {"alumno" => 2456, "nota" => 2},
{"alumno" => 4321, "nota" => 0.75}]
```

Variables estáticas mediante object

object puede usarse para recoger métodos y variables estáticas

```scala
class Persona(nombre:String, edad:Int)

object Persona {
 private var cuentaPersonas = 0
 def apply(nombre:String, edad:Int): Persona = {
   cuentaPersonas += 1
   new Persona(nombre,edad)
 }
 def totalPersonas(): Int = {
   cuentaPersonas
 }
}
```

```scala
val juan = Persona("Juan",23)
val luis = Persona("Luis",31)
println(Persona.totalPersonas)    // 2
```

Modularización

package = objeto especial para agrupar código

import: permite importar código

- Similar a Java, pero utiliza _ en lugar de *
- Permite declarer qué elementos se importan
  - Incluso renombrar elementos al importarlos

Templates

```
sealed trait Arbol[A]
case class Hoja[A](info:A) extends Arbol[A]
case class Rama[A](izq:Arbol[A],der:Arbol[A]) extends
Arbol[A]

def nodos[A](a: Arbol[A]): Int =
 a match { case
   Hoja(_) => 1
   case Rama(izq,der) => nodos(izq) + nodos(der)
}

def sumaNodos(a: Arbol[Double]): Double =
 a match { case
   Hoja(n) => n
   case Rama(izq,der) => sumaNodos(izq) + sumaNodos(der)
}
```

```
val a = Rama(Hoja(5.0),Rama(Hoja(5.5),Hoja(9.5)))
println(nodos(a))                                    //3
println(sumaNodos(a))                                // 20.0
```

Generación de números aleatorios

```
// ImportClass.scala

import   util.Random

val r = new Random

println(r.nextInt(10))
println(r.nextInt(10))
println(r.nextInt(10))
```

Manejo de fechas

Recomendación: nscala-time (https://github.com/nscala-time/nscala-time)

```
//    scala    -classpath    *.jar    import
org.joda.time.Days                    import
org.joda.time.Interval
importcom.github.nscala_time.time.Imports._

DateTime.now + 2.months
// devuelve org.joda.time.DateTime = 2009-06-27T13:25:59.195-07:00

DateTime.nextMonth < DateTime.now + 2.months
// devuelve Boolean = true

DateTime.now to DateTime.tomorrow
// devuelve org.joda.time.Interval = > 2009-04-27T13:47:14.840/2009-04-28T13:47:14.840

(DateTime.now to DateTime.nextSecond).millis //
devuelve Long = 1000

2.hours + 45.minutes + 10.seconds
// devuelve com.github.nscala_time.time.DurationBuilder

(2.hours + 45.minutes + 10.seconds).millis
// devuelve Long = 9910000

2.months + 3.days
// devuelve Period
```

Manejo de fechas

```
importcom.github.nscala_time.time.Imports._

importorg.joda.time.Days objectDiasParaNavidad

extendsApp {

    // 1: obtener la fecha actual
    valahora =DateTime.now

    // 2: representar la fecha de navidad valnavidad
    =(newDateTime).withYear(2015)
                            .withMonthOfYear(12)
                            .withDayOfMonth(25)

    // 3: determinar los días para navidad
    valdaysToXmas =Days.daysBetween(ahora, navidad).getDays

    // 4: mostrar el resultado
    println(s"Días para navidad = $daysToXmas")

    //  bonus:  ¿Qué  día  es  hoy  dentro  de  200  días
    valahoramas200=ahora + 200.days
    println(s"Dentro de 200 días = $ahoramas200")

}
```

Otras características

Parámetros implícitos
package objects
varianza/covarianza de tipos
tipos de rango superior
...

Referencias

Tutoriales:

http://docs.scala-lang.org/
http://twitter.github.io/scala_school/

Guía de estilo:

http://twitter.github.io/effectivescala/

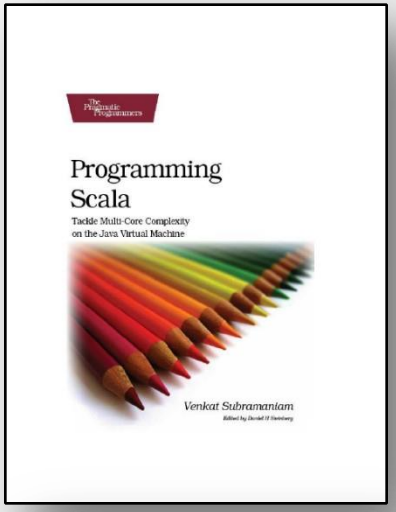
Artículo acerca del lenguaje

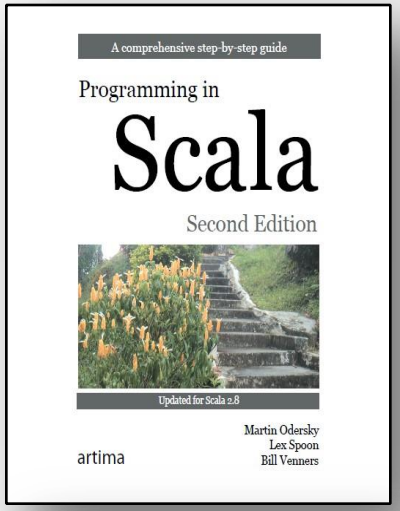
Unifying Functional and Object Oriented Programming, Martin Odersky

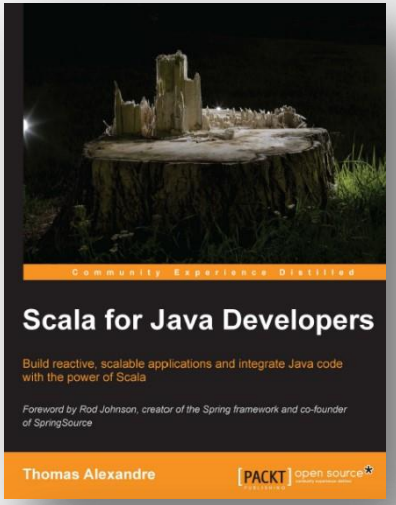
Communications of the ACM

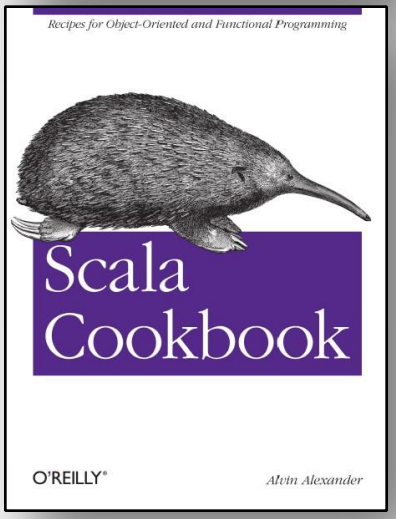
http://cacm.acm.org/magazines/2014/4/173220-unifyingfunctional-and-object-oriented-programming-with-scala/fulltext

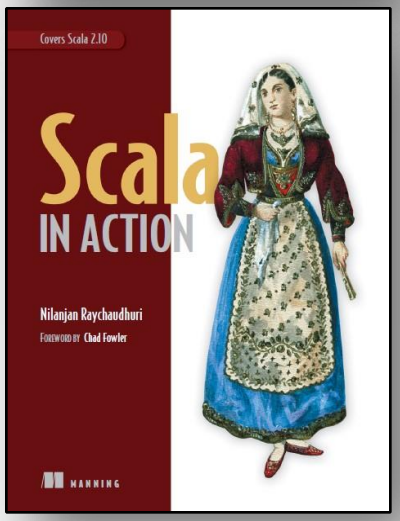
Libros

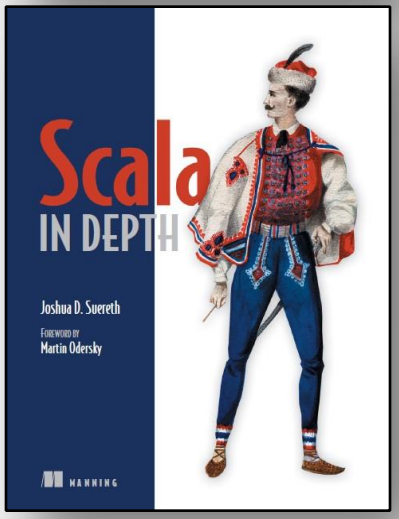

Programming
Scala
Tackle Multi-Core Complexity
on the Java Virtual Machine
Venkat Subramaniam


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


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