Programación Funcional en Scala

– Tema 2 (Parte I) –
Introducción a la Programación Funcional

Jesús López González
jesus.lopez@hablapps.com

Programación Funcional en Scala
Habla Computing

Cursos ETSII-URJC 2015

Índice

¿Qué es la Programación Funcional?

Efectos de Lado

Se dice que una función tiene efectos de lado si, además de devolver un valor, modifica algún estado o interactúa visiblemente con el mundo externo.

Ejemplos

- Modificar una variable global
- Invocar un *setter*
- Lanzar una excepción
- Imprimir una traza por la consola
- Actualizar un dato en una BBDD
- Invocar un servicio web externo

¿Qué es la Programación Funcional?

Paradigma Funcional y Pureza

La programación funcional se basa en una única premisa: programar con *funciones puras*. Una función pura es aquella que no tiene efectos de lado y que para una misma entrada siempre devolverá la misma salida (mapeo de valores).

Listing 1: Ejemplo de función pura

```
def add(a: Int, b: Int): Int = a + b
```

¿Qué es la Programación Funcional?

Transparencia Referencial y Modelo de Sustitución

Se dice que una expresión es *referentially transparent* si tras reemplazarla por el valor que devuelve, no se aprecia ningún cambio en el comportamiento del programa. Esta cualidad activa un nuevo modo de evaluar nuestros programas: *el modelo de sustitución*. Y es que cuando contamos con la transparencia referencial podemos razonar sobre nuestros programas como si se tratasen de expresiones algebraicas.

Listing 2: Aplicando el modelo de sustitución sobre una expresión

```
def add(a: Int, b: Int): Int = a + b

add(add(1, 2), add(add(4, 5), 6))
add(3, add(add(4, 5), 6))
add(3, add(9, 6))
add(3, 15)
18
```

¿Qué es la Programación Funcional?

Listing 3: ¿Es una función pura?

```
var lastAdd: Int = 0

def add2(a: Int, b: Int): Int = {
  val res = a + b
  lastAdd = res
  res
}

// Puedes apoyarte en el siguiente programa...

add2(1, 2)
doSomething(lastAdd)
```

¿Qué es la Programación Funcional?

Listing 4: ¿Y esta función?

```
def add3(a: Int, b: Int): Int = {
  println(s"Adding $a and $b")
  a + b
}
```

Listing 5: ¿Y ésta otra?

```
def add4(a: Int, b: Int): Int = {
  var res = a + b
  res = res - 1
  res = res + 1
  res
}
```

¿Qué es la Programación Funcional?

¿Por qué preocuparnos por el Paradigma Funcional?

- Eficiencia / Escalabilidad (multi-cores, Big Data)
- Modularidad (reusabilidad, composición...)
- Testability (property-based testing)
- Comprensibilidad (estructura, funcionamiento, depuración...)

¿Qué es la Programación Funcional?

¿Y por qué preocuparnos por Scala?

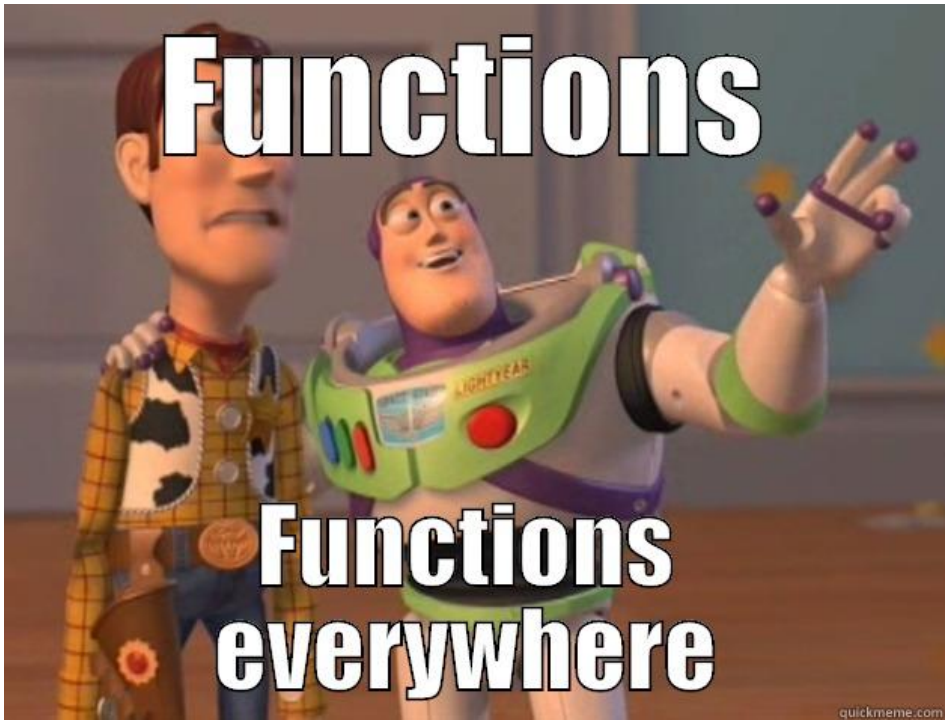
- Multiparadigma e Impuro (*Scala as a better Java*)
- Lenguaje JVM estáticamente tipado
- Más que un lenguaje, un ecosistema (Play, Akka, Spark...)

Listing 6: Hola Mundo en Scala

```
object HolaMundo {
  def main(args: Array[String]): Unit = {
    println("¡Hola Mundo!")
  }
}
```

Índice

Funciones en Scala



Funciones en Scala

Definición de Función

Una **función** es un dispositivo computacional componible que permite exclusivamente transformar valores de entrada en valores de salida.

Listing 7: Nuestra ya conocida función add

```
def add(a: Int, b: Int): Int = a + b
```

Funciones en Scala

Funciones como ciudadanos de primera clase

En Scala, es posible tratar una función como si de un valor se tratase. Por tanto, una función puede ser utilizada para definir el valor de una variable o para pasarse como parámetro a otras funciones (ver diapositiva 14).

Listing 8: Funciones como ciudadanos de primera clase

```
scala> def add(a: Int, b: Int): Int = a + b
add: (a: Int, b: Int)Int

scala> val addV1: (Int, Int) => Int = add
addV1: (Int, Int) => Int = <function2>

scala> addV1(1, 2)
res0: Int = 3

scala> val addV2 = add _
addV2: (Int, Int) => Int = <function2>

scala> addV2(1, 2)
res1: Int = 3
```

Funciones en Scala

Funciones de orden superior

Una función de orden superior es aquella que cumple al menos una de las siguientes condiciones:

- Toma al menos una función como entrada
- Devuelve una función como salida

Listing 9: Funciones de orden superior

```
scala> def add(a: Int, b: Int): Int = a + b
add: (a: Int, b: Int)Int

scala> def mul(a: Int, b: Int): Int = a * b
mul: (a: Int, b: Int)Int

scala> def reduce(pair: (Int, Int), f: (Int, Int) => Int): Int =
     |   f(pair._1, pair._2)
reduce: (pair: (Int, Int), f: (Int, Int) => Int)Int

scala> reduce((3, 2), add)
res0: Int = 5

scala> reduce((3, 2), mul)
res1: Int = 6
```

Funciones en Scala

Funciones anónimas o expresiones lambda

En Scala, el trabajo con funciones es tan frecuente que surge la necesidad de contar con funciones anónimas que puedan definirse *inline* con una sintaxis ligera. Estas funciones también son conocidas como expresiones lambda.

Listing 10: Uso de funciones anónimas

```
scala> val add: (Int, Int) => Int = (a: Int, b: Int) => a + b
add: (Int, Int) => Int = <function2>

scala> val add2: (Int, Int) => Int = (a, b) => a + b
add2: (Int, Int) => Int = <function2>

scala> val add3: (Int, Int) => Int = _ + _
add3: (Int, Int) => Int = <function2>

scala> def reduce(pair: (Int, Int), f: (Int, Int) => Int): Int =
    f(pair._1, pair._2)
reduce: (pair: (Int, Int), f: (Int, Int) => Int)Int

scala> reduce((5, 3), (a, b) => a * b)
res0: Int = 15
```

Variadic Function

Hay funciones que pueden invocarse con un número variable de argumentos. Tales funciones son conocidas como *variadic function* y pueden llegar a resultar muy útiles en diversos contextos (ver diapositiva 20).

Listing 11: Declaración y uso de una función variadic

```
scala> def add(xs: Int*) = xs.fold(0)(_ + _)
add: (xs: Int*)Int

scala> add(1, 2)
res0: Int = 3

scala> add(1, 2, 3, 4, 5)
res1: Int = 15

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

scala> add(l:_*)
res3: Int = 15
```

Función con múltiples listas de argumentos

Hasta ahora todas las funciones que hemos visto tienen una única lista de argumentos de entrada. Scala nos permite que podamos desplegar varias listas. Esto, que a priori puede resultar de poca utilidad, tiene aplicación en ciertas situaciones:

- En el paso de parámetros implícitamente (que veremos en el T3)
- Para ayudar al compilador con la inferencia de tipos
- Si queremos utilizar la técnica de *currying*, que veremos en el siguiente apartado.

Listing 12: Función con múltiples listas de argumentos

```
scala> def add(a: Int)(b: Int)(c: Int, d: Int): Int =
     |   a + b + c + d
add: (a: Int)(b: Int)(c: Int, d: Int)Int

scala> add(1)(2)(3, 4)
res4: Int = 10
```

Currying

La currificación consiste en transformar una función que utiliza múltiples argumentos (o más específicamente una n-tupla como argumento) en una función que utiliza un único argumento.

Listing 13: Currying nativo en Scala

```
scala> def add(a: Int)(b: Int)(c: Int, d: Int): Int =
     |   a + b + c + d
add: (a: Int)(b: Int)(c: Int, d: Int)Int

scala> val f = add(1) _
f: Int => ((Int, Int) => Int) = <function1>

scala> val g = f(2)
g: (Int, Int) => Int = <function2>

scala> g(3, 4)
res0: Int = 10
```

Ejercicio 1

Implementa la función *def curry*, cuya signature se muestra en la siguiente sesión REPL.

Listing 14: Implementando nuestra propia función de currificación

```
scala> def curry(f: (Int, Int) => Int): Int => Int => Int = ???
curry: (f: (Int, Int) => Int)Int => (Int => Int)

scala> def add(x: Int, y: Int) = x + y
add: (x: Int, y: Int)Int

scala> val f = curry(add)
f: Int => (Int => Int) = <function1>

scala> f(1)(2)
res0: Int = 3
```

Composición de Funciones

De manera informal, podemos definir la composición de funciones como la aplicación de una función al resultado de otra.

Listing 15: Composición nativa de funciones show y double

```
scala> val double: Int => Int = x => x * 2
double: Int => Int = <function1>

scala> val show: Int => String = x => x.toString
show: Int => String = <function1>

scala> val showDouble = show compose double
showDouble: Int => String = <function1>

scala> showDouble(2)
res0: String = 4
```

Ejercicio 2

Implementa la función *compose*, cuya signatura se muestra en la siguiente sesión REPL.

Listing 16: Uso de nuestra propia función de composición (prefija)

```
scala> def compose(f: Int => String, g: Int => Int): Int =>
    String = ???
compose: (f: Int => String, g: Int => Int)Int => String

scala> val showDouble = compose(show, double)
showDouble: Int => String = <function1>

scala> showDouble(4)
res0: String = 8
```

Función Polimórfica

Las funciones en Scala pueden estar parametrizadas con valores y con tipos. Ya sabemos que los parámetros valor se definen entre paréntesis. Por su parte, los parámetros tipo se declaran entre corchetes. Los parámetros tipo nos permiten abstraer un comportamiento recurrente. De esta manera, una única implementación podría ser suficiente para cubrir un gran número de casos.

Listing 17: Función polimórfica add

```
scala> def add[A](x: A, y: A)(f: (A, A) => A): A = f(x, y)
add: [A](x: A, y: A)(f: (A, A) => A)A

scala> add(1, 2)(_ + _)
res0: Int = 3

scala> add("hola", "mundo")(_ + _)
res1: String = holamundo

scala> add(List(1, 2, 3), List(4, 5, 6))(_ ++ _)
res2: List[Int] = List(1, 2, 3, 4, 5, 6)
```

Ejercicio 3

Implementa las funciones *curry* y *compose* de forma polimórfica y demuestra su correcto funcionamiento con varios escenarios. ¿Cuántos parámetros tipo te hacen falta para implementar cada una de estas funciones?

Funciones en Scala