

Programación Funcional en Scala

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

Jesús López González
jesus.lopez@hablapps.com

Programación Funcional en Scala
Habla Computing

Cursos ETSII-URJC 2015

- 1 Objects and Case Classes
- 2 Pattern Matching
- 3 Tipos algebraicos de datos (ADTs)
- 4 Ejemplo: Plegando Listas
- 5 Takeaways

Objects

Scala tiene objetos *singleton*, que podemos entender como una clase que tiene una única instancia. Se declaran utilizando la palabra reservada **object**. ¡Ya vimos un ejemplo cuando creamos nuestro primer programa en Scala!

Listing 1: Hola Mundo en Scala

```
object HolaMundo extends App {  
  println("¡Hola Mundo!")  
}
```

Companion Objects

Un *companion object* es un objeto que tiene el mismo nombre que un tipo de datos, dónde suelen declararse ciertos métodos que crean o trabajan con valores de dicho tipo de datos.

Listing 2: Companion Object para la clase Alumno

```
class Alumno(val nombre: String, val apellidos: String, val
    edad: Int)

object Alumno {
    def crear(nombre: String, apellidos: String, edad: Int):
        Alumno =
        new Alumno(nombre, apellidos, edad)
}
```

Método apply

El método *apply* es un método especial que te permite tener una sintaxis más agradable cuando una clase o un objeto tienen un uso principal. Para nuestro ejemplo, el uso principal del companion object es el de crear objetos.

Listing 3: Usando el método apply como constructor

```
class Alumno(val nombre: String, val apellidos: String, val
    edad: Int)

object Alumno {
    def apply(nombre: String, apellidos: String, edad: Int):
        Alumno =
        new Alumno(nombre, apellidos, edad)
}
```

Método unapply

En la programación funcional, al igual que existen *constructores*, que recolectan las partes del objeto para proceder a su creación, también existen *extractores*, que dado un objeto instanciado, nos devuelve las partes que se utilizaron en su construcción. En Scala, es tan habitual utilizar el método *apply* para construir objetos, que se utilizó el nombre *unapply* como método especial que implementa la extracción.

Listing 4: Declaración de apply y unapply en el object Alumno

```
class Alumno(val nombre: String, val apellidos: String, val
  edad: Int)

object Alumno {
  def apply(nombre: String, apellidos: String, edad: Int):
    Alumno =
    new Alumno(nombre, apellidos, edad)

  def unapply(alumno: Alumno): Option[(String, String, Int)] =
    Some((alumno.nombre, alumno.apellidos, alumno.edad))
}
```

Case Classes

*“Las **case classes** son clases regulares que exportan los parámetros de sus constructores y ofrecen un mecanismo de decomposición recursiva mediante **pattern matching**.”* En definitiva son clases que generan automáticamente un companion object y una implementación para los métodos apply y unapply, entre otras muchas cosas.

Listing 5: Alumno en su versión case class

```
case class Alumno(nombre: String, apellidos: String, edad: Int)
```


Listing 6: Jugando con la case class Alumno

```
scala> case class Alumno(nombre: String, apellidos: String,  
    edad: Int)  
defined class Alumno  
  
scala> val pepe = Alumno("Pepe", "Pérez", 40)  
pepe: Alumno = Alumno(Pepe,Pérez,40)  
  
scala> val Alumno(nm, ap, ed) = pepe  
nm: String = Pepe  
ap: String = Pérez  
ed: Int = 40
```

- 1 Objects and Case Classes
- 2 Pattern Matching
- 3 Tipos algebraicos de datos (ADTs)
- 4 Ejemplo: Plegando Listas
- 5 Takeaways

¿Qué es el Pattern Matching?

Scala trae consigo un mecanismo genérico de *pattern matching* (coincidencia de patrones). Permite hacer matching con cualquier tipo de dato bajo una política *first-match*.

Listing 7: Pattern Matching sobre un Int

```
scala> def f(i: Int): String = i match {  
  |   case 1 => "uno"  
  |   case 2 => "dos"  
  |   case _ => "otro"  
  | }  
f: (i: Int)String  
  
scala> f(1)  
res0: String = uno  
  
scala> f(3)  
res1: String = otro
```

Patrones

El verdadero poder del Pattern Matching en Scala viene por la gran expresividad que permiten sus patrones. Los extractores (case classes) nos permiten explotar muy bien dichos patrones.

Listing 8: Ejemplos de patrones

```
scala> case class Identificacion(nombre: String, apellidos:
      String)
defined class Identificacion

scala> case class Alumno(id: Identificacion, edad: Int)
defined class Alumno

scala> def esCandidato(alumno: Alumno): Boolean = alumno match {
      | case Alumno(Identificacion("pepe", "perez"), 40) => true
      | case Alumno(id@Identificacion("jose", _), _) => true
      | case Alumno(_, edad) if (edad <= 35) => true
      | case _ => false
      | }
esCandidato: (alumno: Alumno)Boolean
```

Listing 9: Implementación del método esCandidato

```
scala> def esCandidato(alumno: Alumno): Boolean = alumno match {  
  |   case Alumno(Identificacion("pepe", "perez"), 40) => true  
  |   case Alumno(Identificacion("jose", _), _) => true  
  |   case Alumno(_, edad) if (edad <= 35) => true  
  |   case _ => false  
  | }  
esCandidato: (alumno: Alumno)Boolean
```

Listing 10: Ejercicio: ¿Quiénes son candidatos?

```
val jose = Alumno(Identificacion("jose", "garcia"), 40)  
val pepe = Alumno(Identificacion("pepe", "perez"), 30)  
val ana = Alumno(Identificacion("ana", "marquez"), 25)
```

- 1 Objects and Case Classes
- 2 Pattern Matching
- 3 Tipos algebraicos de datos (ADTs)**
- 4 Ejemplo: Plegando Listas
- 5 Takeaways

¿Qué es un ADT?

Un ADT es un tipo de datos definido por uno o varios constructores, donde cada uno de ellos puede contar con cero o más argumentos. Así, decimos que el tipo de datos es la suma (o unión) de sus constructores y que cada constructor es el producto de sus argumentos, de ahí el nombre. Ejemplos de ADT podrían ser *Option*, *List*, *Tree*, etc.

- 1 Objects and Case Classes
- 2 Pattern Matching
- 3 Tipos algebraicos de datos (ADTs)
- 4 Ejemplo: Plegando Listas
- 5 Takeaways

- 1 Objects and Case Classes
- 2 Pattern Matching
- 3 Tipos algebraicos de datos (ADTs)
- 4 Ejemplo: Plegando Listas
- 5 Takeaways