Nuevas Tecnologías de la Programación

Tema 4: estructuras de control

Curso 2017-18

Índice

1. Expresiones y sentencias
2. Expresiones if-else
3. Expresiones match
4. Estructuras iterativas

Expresiones y sentencias

El término **expresión** alude a una unidad de código que devuelve un valor tras su ejecución. Varias líneas de código pueden considerarse una expresión si están agrupadas en un bloque, delimitado por llaves.

Las expresiones constituyen una de las bases de la programación funcional al hacer posible que la ejecución se base en devolución de valores y no en modificar datos ya existentes. Esto permite el uso de datos inmutables (uno de los principios básicos de la programación funcional).

Expresiones y sentencias

Esta visión de las expresiones permite considerar a las funciones como un tipo especial de expresión. La ventaja de esta forma de ver la programación radica en la reducción al máximo de los efectos laterales.

Consideremos algunas expresiones, comenzando por las más sencillas:

```
scala> "Hola"
res0: String = Hola
```

El resultado de esta expresión se almacena en el valor res0.

Expresiones y sentencias

Las expresiones también pueden usarse al declarar valores:

```
scala> val x = 5*20
x: Int = 100

scala> val cantidad = x+10
cantidad: Int = 110
```

Expresiones y sentencias

También pueden usarse bloques:

```
scala> val cantidad = { val x=5*20; x+10}
cantidad: Int = 110
```

La última sentencia del bloque es lo que devuelve la expresión. De esta forma, el valor asignado a la variable **x** es 10. El uso del punto y coma evita escribir la expresión en varias líneas.

Expresiones y sentencias

Si no se usa el punto y coma debemos separar las sentencias por líneas:

```
scala> val cantidad = {
     |     val x = 5*20
     |     x+10
     | }
cantidad: Int = 110
```

Expresiones y sentencias

También puede haber anidamiento de bloques:

```
scala> {val a=1; {val b=a*2; {val c=b+4; c}}}
res3: Int = 6
```

Expresiones y sentencias

Se denomina **sentencias** a aquellas instrucciones (o conjuntos de instrucciones) que no devuelven valor alguno, por lo que su tipo asociado es **Unit** (en escala, todo tiene tipo asociado). Un ejemplo sencillo es el siguiente:

```
scala> val x=1
x: Int = 1
```

Se aprecia que la declaración de **x** no produce devolución de valor (en el mensaje de **Scala** no aparece **res** por ningún lado).

Expresiones y sentencias

De igual forma, los bloques de sentencias no devuelven valor alguno. Suelen usarse para modificar datos existentes, hacer cambios en el estado de la aplicación, escribir datos en pantalla, actualizar bases de datos, conectar con servidores externos, etc.

Índice

1. Expresiones y sentencias
2. Expresiones if-else
3. Expresiones match
4. Estructuras iterativas

Expresiones if-else

Se trata de una estructura de decisión ya conocida y presente en todos los lenguajes de programación. En **Scala** hablamos de expresión ya que se devuelve un valor, aunque en algunas ocasiones no se use.

```
scala> if(47 % 3 > 0) println("No es multiplo de 3")
No es multiplo de 3
```

Expresiones if-else

Veamos algún ejemplo de devolución de valor:

```
scala> val x=10
x: Int = 10

scala> val y=3
y: Int = 3

scala> val maximo = if(x > y) x else y
maximo: Int = 10
```

Expresiones if-else

Puede haber situaciones en que no se considere la devolución de valor en todos los casos

```
scala> val x=3
x: Int = 3

scala> val y=10
y: Int = 10

scala> val maxX = if(x > y) true
maxX: AnyVal = ()
```

Se promociona el tipo devuelto a **AnyVal**, supertipo de todos los tipos de valores.

Índice

1. Expresiones y sentencias
2. Expresiones if-else
3. Expresiones match
4. Estructuras iterativas

Expresiones match

Se trata de otro tipo de estructura de decisión, condicional o selectiva. Tiene similitudes con la sentencia **switch** de **Java**. Se caracteriza por:

- evaluar una entrada
- seleccionar una rama de ejecución que coincide con ella
- admite una operación por defecto

Al ser una expresión, devuelve un valor. A diferencia de lo que ocurre en **Java** sólo habrá coincidencia con una entrada, por lo que no es necesario el uso de **break**.

Expresiones match

Otra diferencia esencial consiste en que en **Scala** la coincidencia puede ser más compleja que la igualdad con un simple valor y puede considerar la igualdad de tipos, expresiones regulares, rangos numéricos, contenido de estructuras, etc. Como se verá más adelante es una sentencia muy potente en **Scala**.

```
scala> val maximo = x > y match {
     | case true => x
     | case false => y
     | }
maximo: Int = 10
```

Expresiones match

Otro ejemplo:

```
scala> val error = 500
error: Int = 500

scala> val mensaje = error match {
     | case 200 => "Funcionamiento correcto"
     | case 400 => {
     |               println("Error de ejecucion")
     |               "Error 400"
     |             }
     | case 500 => {
     |               println("Error sintactico")
     |               "Error 500"
     |             }
     | }
Error sintactico
mensaje: String = Error 500
```

Expresiones match

También permite usar varias alternativas asociadas a un único bloque de sentencias:

```
scala> val dia="lunes"
dia: String = lunes

scala> val laborable = dia match {
     | case "lunes" | "martes" | "miercoles" | "jueves"
          | "viernes" => "laborable"
     | case "sabado" | "domingo" => "festivo"
     | }
laborable: String = laborable
```

Expresiones match

Si no hay coincidencia con ninguna de las alternativas (patrones) se produce un error de ejecución. Podemos verlo si en el caso anterior inicializamos la variable **dia** con el valor **LUNES**:

```
scala> val dia="LUNES"
dia: String = LUNES

scala> val laborable = dia match {
     | case "lunes" | "martes" | "miercoles" | "jueves"
          | "viernes" => "laborable"
     | case "sabado" | "domingo" => "festivo"
     | }
scala.MatchError: LUNES (of class java.lang.String)
  ... 38 elided
```

Expresiones match

En las expresiones match pueden usarse dos tipos de comodines: ligadura a variable y comodín. En la ligadura a variable se enlaza la entrada a una expresión match con una variable local. De esta forma actúa como comodín, ya que coincidirá con cualquier valor.

```
scala> val estado=mensaje match{
     | case "OK" => 200
     | case otro => {
     |                println(s"Cadena no considerada: $otro")
     |                -1
     |              }
     | }
estado: Int = 200
```

En este caso la variable **otro** coincidirá con cualquier cadena diferente a **OK**. La variable **otro** sólo tiene vida durante la ejecución de la sentencia **match**.

Expresiones match

El comodín se representa con un subrayado. El valor ligado al comodín puede usarse en la expresión a ajecutar:

```
scala> val mensaje="No autorizado"
mensaje: String = No autorizado

scala> val estado=mensaje match{
     | case "OK" => 200
     | case _ => {
     |             println(s"Imposible procesar - $mensaje")
     |             -1
     |           }
     | }
Imposible procesar - No autorizado
estado: Int = -1
```

Expresiones match

También es posible añadir condiciones a las alternativas:

```
scala> val respuesta=null
respuesta: Null = null

scala> respuesta match {
     | case s if s != null => println(s"Recibido: $s")
     | case s => println("Error: respuesta nula")
     | }
Error: respuesta nula
```

Expresiones match

Los patrones pueden incluir indicaciones de tipos:

```
scala> val x:Int=12345
x: Int = 12345

scala> val y:Any=x
y: Any = 12345

scala> y match {
     | case z:String => s"$z"
     | case z:Double => f"$z%.2f"
     | case z:Float => f"$z%.2f"
     | case z:Long => s"${z}l"
     | case z:Int => s"${x}"
     | }
res0: String = 12345
```

Índice

1. Expresiones y sentencias
2. Expresiones if-else
3. Expresiones match
4. Estructuras iterativas

Estructuras iterativas

La estructura iterativa más importante en **Scala** es la estructura **for**. Permite iterar sobre un conjunto de datos ejecutando alguna expresión en cada iteración y devolviendo una colección de todos los valores o expresiones calculadas.

Es una estructura que puede adaptarse fácilmente para incluir iteración anidada, filtrado, etc.

Estructuras iterativas

Empezamos considerando una estructura de datos llamada **Range**, que permite iterar sobre una serie de números. Se crean usando las palabras reservadas **to** o **until**, que indican en final de la serie de valores. La definición del rango se hace mediante la siguiente sintaxis:

```
<valor inicial> [to | until] <valor final> [<incremento>]
```

until no incluye el último valor.

Estructuras iterativas

Usando un rango, la sintaxis de la estructura **for** es:

```
for (<identificador> <- <rango>) [yield] [<expresion>]
```

yield indica que el resultado de cada iteración se devuelve como producto de la iteración.

Estructuras iterativas

Se recomienda usar llaves para el cuerpo de la expresión **for**.

```
scala> for(x <- 1 to 7){println(s"Dia $x")}
Dia 1
Dia 2
Dia 3
Dia 4
Dia 5
Dia 6
Dia 7
```

Estructuras iterativas

Mediante el uso de **yield** podemos generar una colección de mensajes (uno para cada valor del rango):

```
scala> for(x <- 1 to 7) yield {s"Dia $x"}
res2: scala.collection.immutable.IndexedSeq[String] =
      Vector(Dia 1, Dia 2, Dia 3, Dia 4, Dia 5, Dia 6, Dia 7)
```

Se aprecia aquí parecido con la aplicación de la función **map**.

Estructuras iterativas

Pueden usarse filtros para que el recorrido se haga únicamente sobre los valores deseados:

```
scala> val mult3 = for(i <- 1 until 20 if i%3 == 0) yield i
mult3: scala.collection.immutable.IndexedSeq[Int] =
                                        Vector(3, 6, 9, 12, 15, 18)
```

Estructuras iterativas

Pueden usarse simultáneamente varios filtros:

```
scala> val archivos=(new java.io.File(".")).listFiles

scala> for (archivo <- archivos if archivo.isFile
     |                  if archivo.getName.contains("t"))
     | println(archivo)
./beamercolorthememetropolis.sty
./beamerfontthememetropolis.sty
./beamerinnerthememetropolis.sty
.....................................
```

Estructuras iterativas

Pueden usarse varios iteradores a la vez:

```
scala> for{x <- 1 to 2
     |     y <- 1 to 3}
     | println(s"($x,$y)")
(1,1)
(1,2)
(1,3)
(2,1)
(2,2)
(2,3)
```

Estructuras iterativas

Esta expresión puede escribirse con paréntesis si se separan con ; cada iterador o filtro:

```
sscala> for(x <- 1 to 2;
     |      y <- 1 to 3)
     | println(s"($x,$y)")
(1,1)
(1,2)
(1,3)
(2,1)
(2,2)
(2,3)
```

Estructuras iterativas

A veces resulta conveniente definir variables temporales en el cuerpo de la expresión **for**:

```
scala> val resultado=for(i <- 1 to 8;
     |                   pow = 1 << i)
     | yield pow
resultado: scala.collection.immutable.IndexedSeq[Int] =
                  Vector(2, 4, 8, 16, 32, 64, 128, 256)
```

Estructuras iterativas

Otras estructuras iterativas son **while** y **do while**, aunque no son muy usadas en **Scala** al no tratarse de expresiones y no producir valores como resultado.

```
scala> var x=10
x: Int = 10

scala> while(x > 0) {
     |    println(x)
     |    x-=1
     | }
10
9
.....................
1
```

Estructuras iterativas

```
scala> var x=10
x: Int = 10

scala> do {
     |    println(s"Valor de x: $x")
     |    x-=1
     | } while(x > 0)
Valor de x: 10
Valor de x: 9
Valor de x: 8
Valor de x: 7
Valor de x: 6
Valor de x: 5
Valor de x: 4
Valor de x: 3
Valor de x: 2
Valor de x: 1
```