Tema 9: DSLs en Scala

Contenidos

- 1. Domain Specific Languages
  - 1.1. Introducción a los DSLs
  - 1.2. Objetivos y usos de los DSLs
  - 1.3. Características de los lenguajes para definir DSLs
- 2. Definición de DSLs en Scala
  - 2.1. Puntos y paréntesis opcionales
  - 2.2. Bloques de código
  - 2.3. Currying
  - 2.4. Aplicación a DSLs: `writeToFile`
  - 2.5. Parámetros *by-name*
  - 2.6. Aplicación a DSLs: `repetirHasta`
  - 2.7. Implicit
  - 2.8. Aplicación a DSLs: bolsa
  - 2.9. *Pattern matching*
  - 2.10. Clases *case*
  - 2.11. Clases *case* y pattern matching
  - 2.12. Extra: número variable de argumentos

1. Domain Specific Languages

1.1. Introducción a los DSLs

1.1.1. ¿Qué es un DSL?

- DSL: *Domain Specific Language*, Lenguaje Específico de un Dominio
- Un DSL es un “mini-lenguaje” de programación diseñado con un lenguaje de programación más general, que está orientado a poder especificar soluciones concretas en ese dominio restringido.
- Los usuarios finales del lenguaje DSL serán capaces de escribir pequeños programas en ese nuevo lenguaje sin conocer las complejidades del lenguaje subyacente en el que el DSL está construido.
- No todos los lenguajes de programación son apropiados para construir DSLs. El lenguaje subyacente tiene que tener ciertas características para que el lenguaje DSL sea flexible, potente y tenga un aspecto (sintaxis) de un verdadero lenguaje de programación.

1.1.2. Ejemplos de DSLs en Scala

DSL diseñado por Debasish Ghosh para definir acciones en mercados de bolsa (enlace)

```
val orders = List[Order](

  // use premium pricing strategy for order
  new Order to buy(100 sharesOf "IBM")
           maxUnitPrice 300
           using premiumPricing,

  // use the default pricing strategy
  new Order to buy(200 sharesOf "GOOGLE")
           maxUnitPrice 300
           using defaultPricing,

  // use a custom pricing strategy
  new Order to sell(200 bondsOf "Sun")
           maxUnitPrice 300
           using {
             (qty, unit) => qty * unit - 500
           }
)
```

Otro ejemplo, bastante curioso, diseñado por Michael Fogus. Un DSL en Scala que define un lenguaje BASIC (enlace).

Un ejemplo:

```
 object SquareRoot extends Baysick {
      def main(args:Array[String]) = {
        10 PRINT "Enter a number"
        20 INPUT 'n
        30 PRINT "Square root of " % "'n is " % SQRT('n)
        40 END

        RUN
      }
    }
```

Otro ejemplo, algo más complicado, el juego Lunar Lander:

```
object Lunar extends Baysick {
  def main(args:Array[String]) = {
    10 PRINT "Welcome to Baysick Lunar Lander v0.9"
    20 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)

    60 PRINT "You are drifting towards the moon."
    70 PRINT "You must decide how much fuel to burn."
    80 PRINT "To accelerate enter a positive number"
    90 PRINT "To decelerate a negative"

    100 PRINT "Distance " % 'dist % "km, " % "Velocity " % 'v % "km/s, " % "
Fuel " % 'fuel
    110 INPUT 'burn
    120 IF ABS('burn) <= 'fuel THEN 150
    130 PRINT "You don't have that much fuel"
    140 GOTO 100
    150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
    160 LET ('fuel := 'fuel - ABS('burn))
    170 LET ('dist := 'dist - 'v)
    180 IF 'dist > 0 THEN 100
    190 PRINT "You have hit the surface"
    200 IF 'v < 3 THEN 240
    210 PRINT "Hit surface too fast (" % 'v % ")km/s"
    220 PRINT "You Crashed!"
    230 GOTO 250
    240 PRINT "Well done"

    250 END

    RUN
  }
}
```

El código fuente de este DSL en Scala se puede consultar en este enlace.

1.2. Objetivos y usos de los DSLs

- Habitualmente, el producto que diseñamos en informática es un *programa*:
    - Aplicación de escritorio para calcular la declaración de la renta
    - Aplicación web para reservar vuelos
- Los programas tienen un diseño y un uso fijo y específico, cualquier cambio en algunas de sus funcionalidades obliga a recompilarlo y reinstalarlo, o bien en el dispositivo, si es una aplicación de escritorio, o bien en el servidor web si es una aplicación web.
- En informática podemos también construir productos más flexibles que programas:

    - APIs (software que otros desarrolladores pueden instalar y utilizar en sus programas). Por ejemplo, un API para generar documentos PDF o un API para procesar imágenes como OpenCV
    - Servicios web (software desplegado en servidores a los que podemos realizar peticiones desde las aplicaciones que diseñamos). Por ejemplo, el API de servicios de Google Maps o el API de servicios de Evernote.

- Un DSL permite proporcionar mayor flexibilidad todavía que un API o un servicio web:
    - Para solucionar un problema diseñamos un lenguaje y dejamos que el usuario final sea el que especifique la solución.

1.3. Características de los lenguajes para definir DSLs

1.3.1. La definición de DSLs depende de las características del lenguaje

No todos los lenguajes permiten definir DSLs. Ejemplos de lenguajes de programación con características muy adecuadas para definir DSLs:

- Scheme: macros
- Ruby: uso de bloques sin paréntesis
- Scala: lo vemos en el siguiente apartado

1.3.2 Características de Scala que lo hacen apropiado para definir DSLs

Scala tiene muchas características que lo hacen muy apropiado para definir lenguajes de dominio:

- Puntos y paréntesis opcionales
- Operadores como métodos
- Apply
- Parámetros *by-name*
- Currying
- Implicit
- Funciones de orden superior
- Clases Case

Algunas ya las hemos visto. Vamos a ver con más detalle y a repasar alguna de ellas.

2. Definición de DSLs en Scala

2.1. Puntos y paréntesis opcionales

Ya sabemos que cuando un método tiene un único argumento, el compilador permite invocar al método sin escribir los paréntesis ni el punto.

Por ejemplo podemos invocar al método `+` del objeto `1` pasando como argumento el `2` con la expresión `1 + 2`. El compilador la convierte en `1.+(2)`

Se puede hacer más de 2 veces y el compilador llama a los métodos de izquierda a derecha, empezando por el objeto que hay más a la izquierda, llamando al método y pasándole como argumento el siguiente elemento. Esta llamada devolverá un objeto en el que se invoca el siguiente método con el siguiente parámetro. Y así sucesivamente.

Por ejemplo, `1 + 2 + 3` se convierte en `(1.+(2)).+(3)`.

Mezclándolo con funciones de orden superior podemos formar expresiones de este estilo:

```
miListaDeNumeros filter esPar map cuadrado
```

El compilador de Scala convierte esta expresión en:

```
(miListaDeNumeros.filter(esPar)).map(cuadrado)
```

Estamos aplicando el método `filter` a `milistaDeNumeros` con el argumento `esPar`. Al resultado (otra lista) le aplicamos el método `map` con el argumento `cuadrado`. O sea, estamos devolviendo los cuadrados de los números pares de una lista.

2.2. Bloques de código

Scala permite definir bloques de código en los que se realizan una serie de instrucciones y se devuelve el valor de la última instrucción ejecutada. Se definen encerrando las instrucciones entre llaves y separando las instrucciones por puntos y comas o por fines de línea:

```
{println("hola");4}
```

Desde las instrucciones del bloque se pueden acceder a las variables del ámbito en el que se crea el bloque, y en los bloques se pueden crear variables cuyo valor queda encerrado en el bloque:

```
val x = 10
val y = 5
{
   val z = x+10
   val y = 20
   z+y
}
y ⇒ 5
z ⇒ error: not found
```

Podemos escribir un bloque en cualquier lugar en donde vaya un valor, siempre que el valor devuelto por el bloque sea compatible con el tipo del valor que esperamos. El bloque se evalúa y se utiliza el valor devuelto:

```
val x: Int = {println("Hola");2+3}
```

Podemos no escribir el tipo de la variable y Scala lo infiere a partir del valor devuelto por el bloque:

```
val x = 10
val z = {val y = 3; x > y}
⇒ z: Boolean = true
```

Por ejemplo, podemos utilizar un bloque en un argumento de una invocación a una función o un método:

```
def doble(x:Int) = {
   println("Estoy en doble")
   x+x
}

doble({println("Hola");5})
```

Vemos que se evalúa el bloque, imprimiéndose `"hola"` y después se realiza la invocación a `doble` con el valor `5`.

Es muy interesante para construir DSLs que en este caso Scala permite quitar los paréntesis, y dejar sólo el bloque:

```
doble {println("Hola");5}
```

Currying

Recordemos que el currying permite definir una función con varios parámetros como una secuencia de definiciones de funciones de un parámetro.

Por ejemplo una multiplicación

```
def mult(x: Int, y:Int) = x*y
```

la podríamos definir como una función de un argumento *x*, que devuelve una función (clausura) que toma otro parámetro *y* y devuelve el resultado de multiplicar *x* por *y*:

```
def multC(x: Int) = (y:Int) => x*y
```

Para realizar la multiplicación de dos números hay que invocar a `multB` con un parámetro y el resultado (una clausura) invocarlo con el segundo:

```
multC(8)(9)
```

Scala permite definir este tipo de funciones utilizando una notación especial, dos paréntesis en los argumentos:

```
def multC(x:Int)(y:Int) = x*y
```

Al ser funciones de un argumento, podemos utilizar la notación anterior y sustituir los paréntesis por bloques:

```
multC {3} {4}
```

Veremos en el siguiente ejemplo que esto es muy útil para definir DSLs.

2.4. Aplicación a DSLs: `writeToFile`

Vamos a ver un ejemplo algo más avanzado en el que aplicamos las técnicas anteriores. Empezaremos con un ejemplo en el que no las utilizamos y veremos que podemos modificar la solución inicial para el código mucho más expresivo y conciso.

Veamos primero el problema y su solución inicial.

En cualquier proceso de escritura en un fichero en Scala, hay que hacer siempre el siguiente proceso:

1. Obtener el fichero a partir del nombre.
2. Obtener el `PrintWriter` a partir del fichero.
3. Realizar las llamadas a los métodos `write` del `PrintWriter`.
4. Cerrar el `PrintWriter`.

Vamos a intentar encapsular estos pasos en una función llamada `writeToFile`.

Los pasos 1, 2 y 4 siempre son iguales, lo único que cambiará será el nombre de fichero, que lo podemos pasar como parámetro. Pero tenemos el problema del paso 3, la escritura en el fichero. ¿Cómo podemos pasar como parámetro las sentencias `write` sobre el fichero? Queremos diseñar una función que siempre haga lo mismo al principio y al final y que nos permita invocarla **pasándole lo que queremos que haga en el paso 3**.

La solución natural en Scala es pasar como parámetro **una función** que haga el paso 2. Será una función que tendrá como parámetro el objeto `PrintWriter` sobre el que se va a escribir (y que se obtendrá dentro de `writeToFile`) y que realizará la escritura sobre él. La podremos crear con una expresión lambda. Por ejemplo:

```
writeToFile("date.txt", (writer: PrintWriter) => {writer.println(new java.ut
il.Date)}
```

Definimos la función `writeToFile` de la siguiente forma:

```
def writeToFile(fileName: String, op: PrintWriter => Unit) {
   val file = new File(fileName)
   val writer = new PrintWriter(file)
   try {
      op(writer)
   } finally {
      writer.close()
   }
}
```

La función recibe dos argumentos: un nombre de fichero y una función. El fichero se utiliza para obtener un `PrintWriter` y la función es una función que toma un `PrintWriter` y escribe en él. La función no devuelve nada, por eso el tipo de retorno es `Unit`.

Esta es la solución inicial. Vamos a ver cómo hacer todo más conciso, tanto la invocación como la definición. Para ello vamos a utilizar algunas de las características de Scala que hemos visto, como la inferencia de tipos, los bloques o el *currying*.

Empezamos por definir la función `writeToFile` como un *currying*:

```
def writeToFile(fileName: String)(op: PrintWriter => Unit) {
   val file = new File(fileName)
   val writer = new PrintWriter(file)
   try {
      op(writer)
   } finally {
      writer.close()
   }
}
```

Con esto y las bloques podemos hacer una invocación mucho más natural:

```
writeToFile("date.txt") {
    (writer: PrintWriter) => {writer.println(new java.util.Date)}
}
```

Incluso podemos quitar el tipo del parámetro de la clausura porque Scala lo infiere. Y también las llaves del cuerpo de la clausura. Este sería el resultado final, con una sintaxis muy natural:

```
writeToFile("date.txt") {
    writer => writer.println(new java.util.Date)
}
```

2.5. Parámetros by-name

Hay veces que es interesante hacer que un bloque de código que se usa como argumento de una función se evalúe *dentro* de la función en lugar de antes de invocarla.

Ya conocemos una forma de hacerlo, escribiendo el bloque de código en una función sin argumentos que es la que se pasa como parámetro.

Vamos a ver un ejemplo con una nueva versión de la función `doble`:

```
def doble(x: () => Int) = {
    println("Estoy en doble")
    x()+x()
}
```

La función acepta un parámetro que es otra función sin argumento que devuelve un entero. Dentro de la función se evalúa la función que se pasa como parámetro:

```
doble(() => {println("Hola");3})
⇒ Estoy en doble
⇒ Hola
⇒ Hola
⇒ res3: Int = 6
```

Antes de invocar a `doble` se crea la función y se realiza la invocación a `doble` con esa función, que después se invoca en el cuerpo de `doble`.

Vamos a ver que Scala tiene una forma más limpia de definir esto, utilizando una característica denominada parámetros *by-name*. Para definir un *parámetro by-name* hay que definir el tipo del parámetro empezando por `=>` en lugar de `() =>`. Por ejemplo, en el caso anterior, podemos definir la función `dobleByName` cambiando la definición del parámetro `x: () => Int` por `x: => Int`. En el cuerpo de la función cambiaremos la invocación a `x` y en lugar de `x()+x()` escribiremos `x+x`:

```
def dobleByName(x: => Int) = {
    println("Estoy en doble")
    x+x
}
```

Para invocar a `dobleByName` ya no podemos utilizar la sintaxis de antes para crear una función en la invocación, sino que tenemos que escribir un bloque de código en el parámetro.

```
dobleByName {println("Hola");3}
⇒ Estoy en doble
⇒ Hola
⇒ Hola
⇒ res3: Int = 6
```

El funcionamiento es idéntico a antes: se crea una función sin argumentos que contiene el bloque de código y después se invoca a `dobleByName` con esta función. Pero el código es mucho más limpio.

En el bloque que se pasa como argumento se pueden referenciar variables definidas en el ámbito de la invocación.

```
var x = 10
dobleByName {x=x+1;x}
⇒ 23
x ⇒ 12
```

2.6. Aplicación a DSLs: `repetirHasta`

Supongamos que queremos hacer un DSL que nos permita escribir un instrucción *repetir hasta* de la siguiente forma:

```
val x = 0
repetirHasta {
   x = x + 1
   println(x)
}  { x > 10 }
```

¿Cómo lo haríamos?

Podríamos usar la característica anterior de los *parámetros by-name* para pasar el código de los bloques a la función `repetir`. Tendríamos que definir dos bloques, el primero que no devuelva nada y el segundo que devuelva un booleano.

La implementación de `repetirHasta` sería la siguiente:

```
def repetirHasta (codigo: => Unit) (terminacion: => Boolean) = {
   codigo
   while (!terminacion) {
      codigo
   }
}
```

2.7. Implicit

El modificador `implicit` se utiliza para definir conversiones automáticas de objetos de una clase a otra.

Veamos un ejemplo sencillo. Supongamos que definimos la clase `Clase1` que tiene el método `f`:

```
class Clase1(x:Int) {
   def f(y:Int) = x * y
}

implicit def intToClase1(x: Int) = new Clase1(x)
```

Hemos definido la clase y una función que realiza una conversión implícita de un objeto `Int` a un objeto de `Clase1` obtenido a partir del `Int`. El nombre de la función no importa, puede ser cualquiera.

Supongamos ahora que invocamos al método `f` en un objeto `Int`. En principio nos debería dar un error, pero no es así:

```
2 f 3
⇒ 6
```

¿Qué ha pasado? Scala se ha dado cuenta de que el método `f` no está definido en la clase `Int`, pero sí en la clase `Clase1`, por lo que ha buscado si existe alguna función `implicit` que convierta `Int` en `Clase1`. La ha encontrado (es la función `intToClase1`), la ha invocado con el `2` y ha ejecutado `f` sobre el objeto resultante.

Podemos definir distintas conversiones `implicit`, Scala busca la que obtiene el objeto con el método que necesitamos invocar:

```
class Clase2(x:Int) {
   def g(y:Int) = x + y
}

implicit def intToClase2(x: Int) = new Clase2(x)

2 g 3
⇒ 5
```

Por último, Scala también utiliza el tipo del parámetro para desambiguar cuando hay dos clases con métodos con el mismo nombre:

```
class Clase3(x:Int) {
   def f(y:String) = x + y.length
}

implicit def intToClase3(x: Int) = new Clase3(x)
```

Hemos definido otra clase con el mismo método `f` que la `Clase1`, pero este recibe un `String` en lugar de un `Int`. Dependiendo del parámetro de `f` Scala realiza una conversión de un `Int` a una `Clase1` o `Clase3`:

```
5 f 10
⇒ 50
5 f "Hola"
⇒ 9
```

Aplicación a DSLs: bolsa

Veamos un ejemplo de DSL en el que se utiliza `implicit`. Se trata de un DSL que permite crear objetos de tipo `Inversion` a partir de expresiones como `(10 bonosDe "TFN")` o `(40 accionesDe "AAPL")`.

El código completo es el siguiente:

```
import scala.language.implicitConversions

abstract class Inversion(name: String) { def stype: String }
case class Accion(name: String) extends Inversion(name) {
   val stype = "accion"
}
case class Bono(name: String) extends Inversion(name) {
   val stype = "bono"
}

class Cantidad(qty: Int) {
   def accionesDe(name: String) = {
      (qty, Accion(name))
   }

   def bonosDe(name: String) = {
      (qty, Bono(name))
   }
}

class MiniDSL {
   implicit def cantidadInt(i: Int) = new Cantidad(i)
}

object Compras extends MiniDSL {

   def main(args: Array[String]): Unit = {
      val compras = List(
         (10 bonosDe "TFN"),
         (40 accionesDe "AAPL"),
         (100 accionesDe "GOOGL")
      )
      compras map (x => println(x.toString))
   }
}
```

Vemos como se utiliza el modificador `implicit` para definir una conversión automática:

```
implicit def cantidadInt(i: Int) = new Cantidad(i)
```

Esta definición hace que el compilador convierta de forma automática un objeto de tipo `Int` en un objeto de tipo `Cantidad` cuando sea necesario. Por ejemplo, veamos la siguiente expresión:

```
(10 bonosDe "TFN")
```

El compilador se da cuenta de que `bonosDe` es un método de la clase `Cantidad` y de que necesita convertir el `Int` a un objeto de tipo `Cantidad`. El `implicit` es quien se encarga de realizar la conversión.

2.9. *Pattern matching*

- El *pattern matching* es una característica del lenguaje que permite procesar una expresión y realizar acciones en función de la estructura de esa expresión
- Muchos lenguajes de programación tienen sentencias de *pattern matching*: Haskell, Perl, …
- En Scala se utiliza la sentencia `match`, ya hemos visto algunos ejemplos. Vamos a ver algunos más.

2.9.1. Ejemplos de `match`

Ejemplo básico, con un funcionamiento similar al `switch-case` de C y Java:

```
def toYesOrNo(choice: Int): String = choice match {
    case 1 => "yes"
    case 0 => "no"
    case _ => "error"
}
```

Varios casos en un `case`:

```
def toYesOrNo(choice: Int): String = choice match {
    case 1 | 2 | 3 => "yes"
    case 0 => "no"
    case _ => "error"
}
```

En el `case` se pueden instanciar variables. En el siguiento ejemplo se hace el `match` sobre un objeto de tipo `String`:

```
def parseArgument(arg: String) = arg match {
    case "-h" | "--help" => displayHelp
    case "-v" | "--version" => displayVerion
    case whatever => unknownArgument(whatever)
}
```

Se puede hacer un `match` sobre cualquier clase. En el siguiente ejemplo se instancia la variable que empareja con el tipo de `x`:

```
def f(x: Any): String = x match {
    case i:Int => "integer: " + i
    case _:Double => "a double"
    case s:String => "I want to say " + s
}
```

Definición de factorial usando un `match`:

```
def fact(n: Int): Int = n match {
    case 0 => 1
    case n => n * fact(n - 1)
}
```

Definición de `length`. Se usa el patrón "_ :: tail" para comprobar que
la lista tiene uno o más elementos. La variable `tail` se instancia al resto de la lista:

```
def length[A](list : List[A]) : Int = list match {
    case _ :: tail => 1 + length(tail)
    case Nil => 0
}
```

Clases *case*

- Las construcción `case class` es idéntica a `class`
- Por ejemplo, el siguiente código define la clase abstracta `Expr` y sus clases hijas `Number`, `UnOp` y `BinOp`

  ```
  abstract class Expr
  case class Number(num: Double) extends Expr
  case class UnOp(operator: String, arg: Expr) extends Expr
  case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
  ```

- La diferencia con `class` es que el compilador de Scala trata estas clases de una forma especial, permitiendo ciertas expresiones que no se permiten en clases normales.
- En primer lugar, es posible crear objetos sin `new`, usando sólo el nombre de la clase:

  ```
  scala> val n = Number(10.0)
  n: Number = Number(10.0)
  ```

- También es posible hacerlo de forma anidada, creando una estructura compuesta en una única expresión:

  ```
  scala> val exp = BinOp("+",Number(10.0),Number(3.0))
  exp: BinOp = BinOp(+,Number(10.0),Number(3.0))
  ```

- Los parámetros en el constructor de la clase reciben automáticamente el prefijo `val` por lo que se crean como campos:

```
scala> n.num
res9: Double = 10.0
scala> exp.left
res10: Expr = Number(10.0)
```

- Por último, el compilador añade implementaciones por defecto de los métodos `toString`, `hashCode` y `equals`, que recorren recursivamente la estructura:

```
scala> val exp2 = BinOp("+",Number(10.0),Number(3.0))
exp2: BinOp = BinOp(+,Number(10.0),Number(3.0))

scala> exp.equals(exp2)
res13: Boolean = true

scala> exp.toString
res14: String = BinOp(+,Number(10.0),Number(3.0))
```

2.11. Clases *case* y *pattern matching*

- La combinación de la sentencia `match` con las clases case permite expresiones de muy alto nivel.

- Por ejemplo:

```
def describe(e: Expr): String =
    e match {
        case Number(_) => "Numero"
        case UnOp("-",_) => "UnOp negativo"
        case BinOp(_,_,_) => "BinOp"
    }
```

- Evaluación de expresiones:

```
def evalua(e: Expr): Double = e match {
    case Number(x) => x
    case UnOp("-",x) => evalua(x)*(-1.0)
    case BinOp("+",x,y) => evalua(x)+evalua(y)
    case BinOp("-",x,y) => evalua(x)-evalua(y)
    case BinOp("*",x,y) => evalua(x)*evalua(y)
    case BinOp("/",x,y) => evalua(x)/evalua(y)
}

scala> val e = BinOp("*",BinOp("+",Number(10.0),UnOp("-",Number(3.0))),Nu
mber(3.4))
scala> evalua(e)
res16: Double = 23.8
```

2.11.1. Modificador `sealed`

- El modificador `sealed` a una clase obliga a que la herencia de esa clase únicamente se pueda realizar en el mismo fichero fuente donde está definida la clase marcada como sealed:

```
sealed abstract class Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

- Una ventaja es que el pattern matching emite avisos si nos dejamos algún caso sin contemplar (cuando el match no es exhaustivo)

```
def evalua(e: Expr): Double = e match {
    case Number(x) => x
    case BinOp("+",x,y) => evalua(x)+evalua(y)
    case BinOp("-",x,y) => evalua(x)-evalua(y)
}
<console>:12: warning: match is not exhaustive!
missing combination          UnOp
```

2.12. Extra: número variable de argumentos

La sintaxis consiste en añadir un `*` junto al tipo de los argumentos en la definición del método o de la función. Cuando se realiza la invocación los argumentos se empaquetan en un Array.

```
def sum(args: Int*) : Int = {
   if (args.length == 0) 0
      else args.head + sum(args.tail : _*)
}
```

Con `_*` indicamos que hacemos la llamada con una secuencia de elementos, no con un único elemento.

Veamos cómo funciona:

```
sum(1,2,3,4,5,6)
⇒ 21
```

La lista de argumentos la podemos tratar también con un enfoque imperativo:

```
printStrings(args:String*) = {
   var i : Int = 0;
   for( arg <- args ) {
      println("Arg value[" + i + "] = " + arg );
      i = i + 1;
   }
}
```

Bibliografía

- Oderski, *Programming in Scala*: Capítulo 9 (Control Abstraction) y Capítulo 15 (Case Classes and Pattern Matching)
