Scala

Datos

```
{val|var} nombre[: tipo] = expresión
```

- val = constante
- var = variable

Tipos básicos:

- Enteros: Byte, Short, Int, Long (123, 0x7B)
- Reales: Float, Double (1.23, 1.2e3)
- Otros: Char, String, Boolean ('h', "hola", true, false)

Estructuras de datos

```
var tupla = (1, 2, 3)
var (x, y, z) = (1, 2, 3)

var xs = List(1, 2, 3)
var xs = 1 :: List(2, 3)
var xs = 1 to 5
var xs = 1 until 6
var xs = 1 to 10 by 2
var xp2 = xs(2)

var void = ()
```

Encaje de patrones

```
(xs zip ys) map { case (x, y) => x * y }

val num37 = 37
val Num42 = 42
Some(3) match {
   case Some(`num37`) => println("37")
   case Some(Num37) => println("42")
   case _ => println("??")
}
```

Estructuras de control

```
// Condicional completa:
if (condición) expresión else expresión
```

```
// Condicional parcial:
if (condición) expresión
if (condición) expresión else ()

// Bucles:
while (condición) { expresión }
do { expresión } while (condición)
for (nombre <- lista; ...) { expresión }

// Break:
import scala.util.control.Breaks._
breakable { for (x <- xs) { if (Math.random < 0.1) break } }

// Ejemplos for:
for ((x, y) <- xs zip ys) { yield x * y }
(xs zip ys) map { case (x, y) => x * y }

for (x <- xs; y <- yx) { yield x * y }
xs flatMap { x => ys map { y => x * y } }

for (i <- 1 to 5) { println(i) }
for (i <- 1 until 5) { println(i) }
```

Funciones

```
// Alias para un tipo:
type nombre = tipo
type nombre = tipo_genérico[tipo]

// Funciones con nombre:
def nombre(parámetros)[: tipo] = { sentencias }
def nombre[T](parámetros)[: tipo] = { sentencias }
```

- Parámetros: (nombre:{tipo|Any}, ...) // Modo estándar
  (nombre: => {tipo|Any}, ...) // Modo perezoso
  (args:{tipo|Any}*) // Args. variables
- Sentencias una línea: { ...; ... }
- Sentencias varias líneas:

```
{
    ...
    ...
}
```

```
// Funciones anónimas/lambdas:
(parámetros) => expresión
(parámetros) => { sentencias }

// Operadores como funciones:
(1 to 5).map(_*2)
```

```
(1 to 5).map(2*_)
(1 to 5).reduceLeft(_+_)

// Uso de lambdas:
(1 to 5).map(x => x * x)
(1 to 5).map { val x = _*2; println(x); x }

// Concatenación de operaciones:
(1 to 5) filter { _%2 == 0 } map { _*2 }

// Composición de funciones:
def composición(g:R=>R, h:R=>R) = (x:R) => g(h(x))
val f = composición({ _*2 }, {_-1})

// Currying:
val f1 = (a:R, b:R) => (x:R) => (x-a) / b
def f2(a:R, b:R) = (x:R) => (x-a) / b

// Currying (azúcar sintáctico):
def f3(a:R, b:R) (x:R) => (x-a) / b
val f4 = f3(1, 2)_

// Genéricos:
def gmap[T](g:T=>T) (s:List[T]) => s.map(g)
```

Paquetes

```
import scala.collection._                  // Todo el paquete
import scala.collection.Vector             // Solo una clase
import scala.collection.{Vector, Sequence} // Parte del paquete
import scala.collection.{Vector => Vect42} // Parte del paquete
import java.util.{Date => _, _}            // Todo menos Date

package nombre // Al principio del fichero.
package nombre { ... }
```

Orientación a objetos

```
nombre(parámetros)

[abstract] nombre[(parámetros)] [extends Clase/Trait [with Trait
[with ...]]] { sentencias }

trait nombre { sentencias } // Interfaz.

new { sentencias } // Clase anónima.

Parámetros: [private] [var|val] nombre: tipo, ...
```

- x: R, es igual que: **private val** x: R

- Si se indica **val** o **var**, se entiende que el parámetro es público, por lo que se podrá acceder al valor desde fuera de la clase.

Ejemplos:

```
class C(var x: R) { // Constructor definido.
   assert(x > 0, "¡sólo positivos!")
   var y = x             // Miembro público
   val fijo = 5          // Miembro constante
   private var oculto = 1 // Miembro privado
   def this = this(42)    // Constructor alternativo
}

// Invocando el constructor padre:
class D(var x: R)
class C(x: R) extends D(x) { ... }

// Sobre-escribiendo una definición:
class A extends B { override def f = ... }

// Singleton:
object O extends D { ... }

new java.io.File("f.txt") // Crear un objeto nuevo
classOf[String]           // Obtención del tipo
x.isInstanceOf[String]    // Comprobación de tipos
x.asInstanceOf[String]    // Casting de tipos
```

Hola mundo

```
// Versión estándar:
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}

// Versión simplificada:
object HelloWorld extends App {
  println("Hello, world!")
}
```

Ejecutar en el intérprete:

```
HelloWorld.main(null)
```

Compilar:

```
scalac HelloWorld.scala
scalac -d directorio HelloWorld.scala
```

Ejecutar:

```
scala HelloWorld
scala -cp clases HelloWorld
```

Funciones con List

```
++ ++: +: /: :+ :: ::: :\ addString aggregate andThen apply
applyOrElse canEqual collect collectFirst combinations companion
compose contains containsSlice copyToArray copyToBuffer corresponds
count diff distinct drop dropRight dropWhile endsWith equals exists
filter filterNot find flatMap flatten fold foldLeft foldRight forall
foreach genericBuilder groupBy grouped hasDefiniteSize hashCode head
headOption indexOf indexOfSlice indexWhere indices init inits
intersect isDefinedAt isEmpty isTraversableAgain iterator last
lastIndexOf lastIndexOfSlice lastIndexWhere lastOption length
lengthCompare lift map mapConserve max maxBy min minBy mkString
nonEmpty orElse padTo par partition patch permutations prefixLength
product productIterator productPrefix reduce reduceLeft
reduceLeftOption reduceOption reduceRight reduceRightOption repr
reverse reverseIterator reverseMap reverse_::: runWith sameElements
scan scanLeft scanRight segmentLength seq size slice sliding sortBy
sortWith sorted span splitAt startsWith stringPrefix sum tail tails
take takeRight takeWhile to toArray toBuffer toIndexedSeq toIterable
toIterator toList toMap toParArray toSeq toSet toStream toString
toTraversable toVector transpose union unzip unzip3 updated view
withFilter zip zipAll zipWithIndex
```