

Seminario 2: Seminario de Scala

Bibliografía

Este seminario está basado en los siguientes materiales:

- Programming in Scala. Martin Odersky, Lex Spoon, Bil Venners. Ed. Artima.
- Programming Scala. Dean Wampler, Alex Payne. Ed. O'Reilly.
- [Scala by Example](#). Martin Odersky. November 2010.
- [A Scala Tutorial for Java programmers](#). Michel Schinz, Philipp Haller. Mayo 2011.
- [The Scala Language Specification](#). Martin Odersky. November 2010.
- Growing a Language. Higher-Order and Symbolic Computation, 12:221–223. Steele, J G L. (1999)

El lenguaje de programación Scala

Scala es un lenguaje de programación de código abierto creado por Martin Odersky en 2003.

El nombre “Scala” simboliza que está diseñado para crecer con la demanda de sus usuarios (Scalable language). Scala es un lenguaje multiparadigma, que unifica la programación orientación a objetos y la programación funcional, y permite expresar patrones comunes de forma concisa y segura. Es compatible con Java Virtual Machine y la cantidad de código necesario para programar es inferior a la que se requiere con el lenguaje Java.

Estas ventajas han hecho que cada vez sean más las empresas y servicios que apuestan por Scala, como por ejemplo Twitter o el periódico The Guardian, que lo ha implementado en su sitio web, o Linked-in o Foursquare . También es uno de los lenguajes de programación que se utilizan para desarrollar aplicaciones para los dispositivos Android.

Cita de Guy Steele Jr en “Growing a Language” :

“Si le das a una persona un pescado, comerá un día. Si le enseñas a pescar, comerá toda su vida. Si le das las herramientas, puede hacerse de una caña de pescar —¡Y de muchas otras herramientas! Puede construir una maquina que produzca más cañas. De esta manera puede ayudar a otras personas a pescar.”

“El diseño de un lenguaje de programación no puede seguir siendo simplemente una cosa más. Debe tratarse de un patrón —un patrón de crecimiento— un patrón para desarrollar el patrón para definir los patrones que los programadores puedan usar para su trabajo real y su objetivo principal.”

Para Martin Odersky, la principal característica de Scala que permite la escalabilidad de los sistemas que se construyen con él, es la unificación de la programación orientada a objetos

con la programación funcional. En Scala el valor de una función es un objeto y los tipos de funciones son clases que pueden ser heredados.

Scala es un lenguaje que puede ser interpretado o compilado.

Descargar Scala

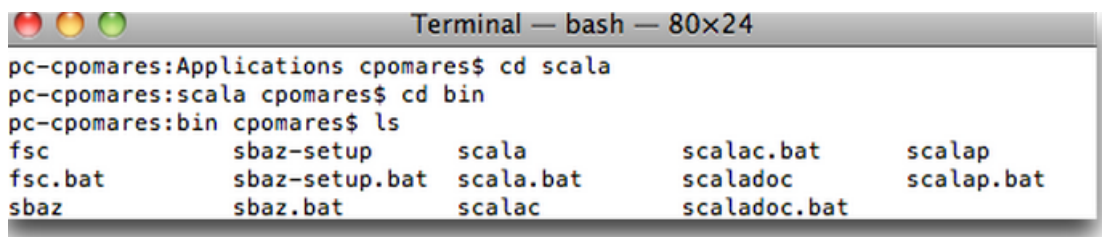
Puedes descargar la última distribución de Scala desde [aquí](#).

Para trabajar con Scala, puedes utilizar cualquier editor de texto y un terminal. Puedes instalarte las [tool support](#) para resaltar la sintaxis de la mayoría de editores de texto conocidos.

Si prefieres trabajar con un IDE, existen [plugins](#) para Eclipse, Netbeans o IntelliJ IDEA

Herramientas de Scala

Las herramientas de línea de comandos de Scala se encuentran dentro de la carpeta bin de la instalación.



```
Terminal — bash — 80x24
pc-cpomares:Applications cpomares$ cd scala
pc-cpomares:scala cpomares$ cd bin
pc-cpomares:bin cpomares$ ls
fsc          sbaz-setup    scala          scalac.bat     scalap
fsc.bat      sbaz-setup.bat scala.bat      scaladoc       scalap.bat
sbaz         sbaz.bat      scalac         scaladoc.bat
```

image

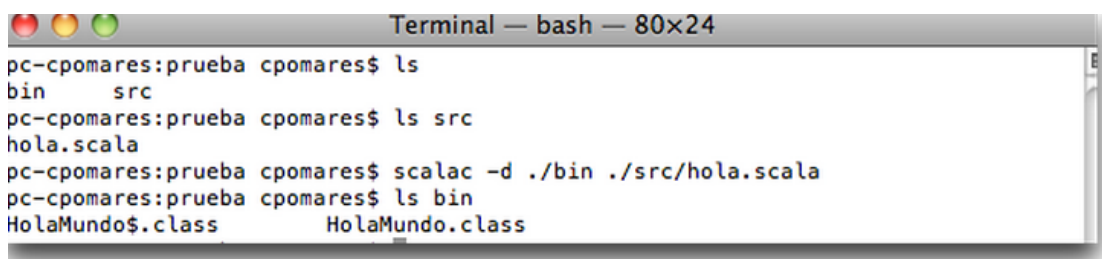
El compilador de scala: `scalac`

El compilador `scalac` transforma un programa de Scala en archivos `.class` para la máquina virtual de Java. El nombre del archivo no tiene que corresponder con el nombre de la clase.

Sintaxis: `scalac [opciones ...] [archivos fuente ...]`

Opciones:

- La opción `-classpath` (`-cp`) indica al compilador donde encontrar los archivos fuente
- La opción `-d` indica al compilador donde colocar los archivos `.class`
- La opción `-target` indica al compilador que versión de máquina virtual usar



```
Terminal — bash — 80x24
pc-cpomares:prueba cpomares$ ls
bin      src
pc-cpomares:prueba cpomares$ ls src
hola.scala
pc-cpomares:prueba cpomares$ scalac -d ./bin ./src/hola.scala
pc-cpomares:prueba cpomares$ ls bin
HolaMundo$.class      HolaMundo.class
```

image

Ejecutor de código: scala`scala` puede operar en tres modos:

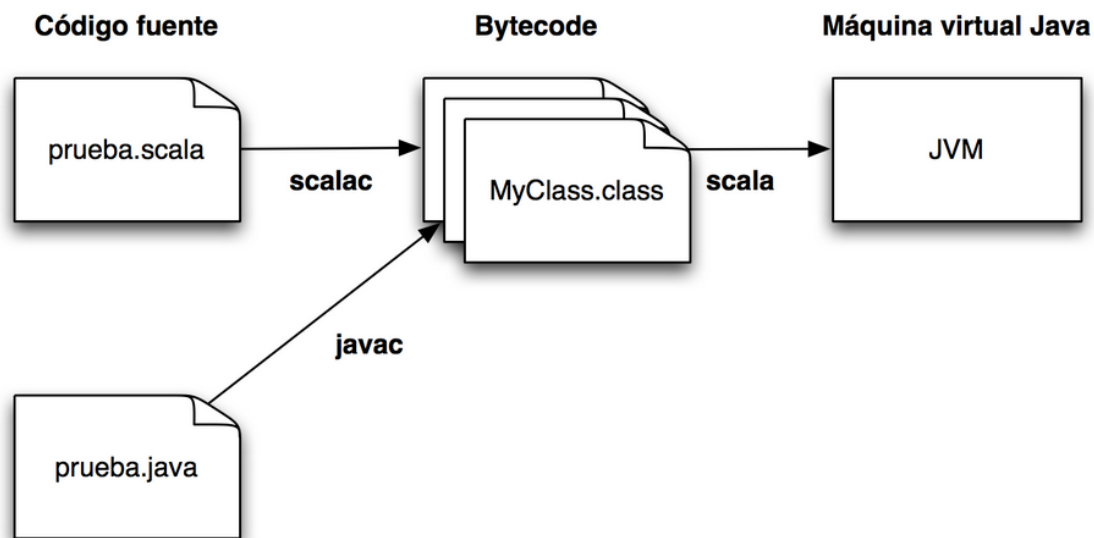
- Puede ejecutar clases compiladas
- Puede ejecutar archivos fuente (scripts)
- Puede operar en forma interactiva, es decir como intérprete

Sintaxis: `scala [opciones ...][script u objeto] [argumentos]`

Si no se especifica un script o un objeto, la herramienta funciona como un evaluador de expresiones interactivo. Si se especifica un script, `scala` lo compila y ejecuta. Cuando se especifica el nombre de una clase, `scala` la ejecuta.

Algunos comandos importantes:

- `:help` muestra mensaje de ayuda
- `:quit` termina la ejecución del intérprete
- `:load` carga comandos de un archivo



image

Scala como lenguaje compilado

Como primer ejemplo, vamos a crear el programa Hola mundo estándar.

```
object HolaMundo {
  def main(args: Array[String]) {
    println("Hola, mundo!")
  }
}
```

La estructura de este programa os debería resultar familiar porque ya conocéis Java: consiste de un método llamado `main` que toma los argumentos de la línea de comando (un array de objetos `String`) como parámetro; el cuerpo de este método consiste en una sola llamada al método predefinido `println` con el saludo como argumento. El método `main` no devuelve un valor, por lo tanto no es necesario que se declare un tipo retorno.

Lo que es menos familiar es la declaración de `object` que contiene al método `main`. Esa declaración introduce lo que es comunmente conocido como objeto *singleton*, que es una clase con una sola instancia. Por lo tanto, dicha construcción declara tanto una clase llamada `HolaMundo` como una instancia de esa clase también llamada `HolaMundo`. Esta instancia es creada bajo demanda, es decir, la primera vez que es utilizada.

Notaréis que el método `main` no está declarado como `static`. Esto es así porque los miembros estáticos (métodos o campos) no existen en Scala. En vez de definir miembros estáticos, el programador de Scala declara estos miembros en un objeto *singleton*.

Compilando el ejemplo

Para compilar el ejemplo utilizaremos `scalac`, el compilador de Scala. `scalac` funciona como la mayoría de los compiladores: toma un archivo fuente como argumento, algunas opciones y produce uno o varios archivos objeto. Los archivos objeto que produce son archivos `class` para la máquina virtual de Java.

Si guardamos el programa anterior en un archivo llamado `HolaMundo.scala`, podemos compilarlo ejecutando el siguiente comando:

```
$scalac HolaMundo.scala
```

Esto generará algunos archivos `class` en el directorio actual. Uno de ellos se llamará `HolaMundo.class` y contiene una clase que puede ser directamente ejecutada utilizando el comando `scala`, como mostramos en la siguiente sección.

Ejecutando el ejemplo

Una vez compilado, un programa Scala puede ser ejecutado utilizando el comando `scala`. Su uso es muy similar al comando `java` utilizado para ejecutar programas Java, y acepta las mismas opciones. El ejemplo de arriba puede ser ejecutado utilizando el siguiente comando, que produce la salida esperada:

```
$scala HolaMundo → Hola, mundo!
```

Scala como lenguaje interpretado desde un *script*

Un script no es más que una secuencia de instrucciones almacenadas en un fichero que se ejecutan secuencialmente. Vamos a crear un script llamado `hola.scala`:

```
println("Hola mundo, desde un script!")
```

Lo ejecutamos:

```
$scala hola.scala
```

Y obtenemos:

```
Hola mundo, desde un script!
```

Los argumentos desde la línea de comandos se le pueden pasar a los scripts mediante el array de argumentos `args`. En Scala, el acceso a los elementos de un array es especificando el índice entre paréntesis (no entre corchetes como en Java). Vamos a probarlo en el siguiente script llamado `holaarg.scala`:

```
println("Hola, "+ args(0) +"!")
```

Lo ejecutamos:

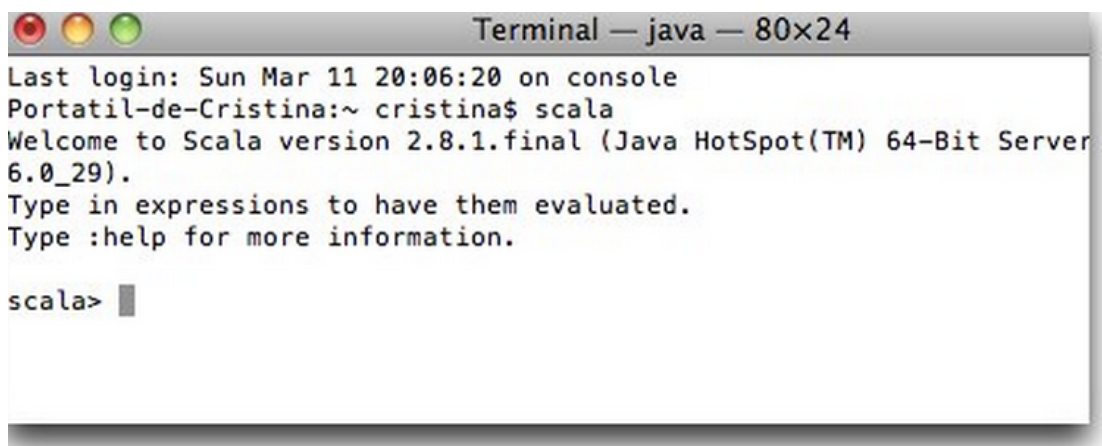
```
$scala holaarg.scala pepe
```

En este comando, “pepe” se pasa como argumento en la línea de comandos, el cual se accede desde el script mediante `args(0)`. Obtenemos:

```
Hola, pepe!
```

Scala como lenguaje interpretado desde un intérprete

Para lanzar el intérprete de Scala, tienes que abrir una ventana de terminal y teclear `scala`. Aparecerá el prompt del intérprete de Scala a la espera de recibir expresiones. Funciona, al igual que el intérprete de Scheme, mediante el bucle *Read-Eval-Print*.



image

Se puede utilizar el intérprete de Scala directamente al estilo de Scheme:

```
scala> 1+2
```

El intérprete devolverá:

```
res0: Int = 3
```

Significa: res0 es un nombre generado automáticamente o definido por el usuario que almacena el resultado

```
Int es el tipo de dato del resultado  
3 es el resultado
```

El identificador resX se puede utilizar en líneas posteriores, por ejemplo:

```
scala> res0 * 3  
res1: Int = 9
```

Podemos imprimir el mensaje “Hola mundo!”:

```
scala> println("Hola mundo!")  
Hola mundo!
```

Para salir del intérprete, podemos hacerlo mediante :quit o :q.

```
scala> :quit  
$
```

Expresiones y funciones simples

Ya hemos visto que el intérprete de Scala se puede ver como una calculadora, donde el usuario introduce expresiones y el intérprete devuelve la evaluación de las mismas. Ejemplo:

```
scala> 5 + 2 * 3  
res2: Int = 11  
  
scala> "hola" + " mundo!"  
res3: String = hola mundo!
```

`def` para dar nombre a expresiones

También es posible nombrar una sub-expresión y utilizar su nombre (identificador) en lugar de la expresión:

```
scala> def scale = 5
scale: Int

scala> 7 * scale
res4: Int = 35

scala> def pi = 3.141592653589793
pi: Double

scala> def radius = 10
radius: Int

scala> 2 * pi * radius
res5: Double = 62.83185307179586
```

`def` es una primitiva declarativa: le da un nombre a una expresión pero no la evalúa.

```
def t = 8 / 0 ⇒ No da error
t ⇒ al evaluar t, error división por cero
```

Es lo contrario que la forma especial `define` de Scheme, que se utiliza para asociar nombres con expresiones, y en primer lugar se evalúa la expresión. Recuerda que nos aporta azúcar sintáctico permitiéndonos crear funciones, evitando escribir la expresión lambda asociada. Es decir, es lo mismo que crear una función en Scheme sin argumentos.

Alternativamente Scala permite una definición de variables utilizando `val` o `var` (lo veremos posteriormente)

Operadores

Operadores aritméticos

Scala soporta los siguientes operadores aritméticos:

Suma	Resta	División	Multiplicación	Resto
+	-	/	*	%

Operadores relacionales

Scala soporta los siguientes operadores relacionales:

Menor	Menor/igual	Mayor	Mayor/igual	Distinto	Igual
<	<=	>	>=	!=	==

Operadores lógicos

Scala soporta los siguientes operadores lógicos:

AND	OR	NOT
&&		!

Tipos de datos en Scala

Los tamaños y rangos de los tipos básicos corresponden a los de Java. Aunque hablemos de tipos de datos, en Scala todos los tipos de datos son clases.

Tipo de dato	Rango	Ejemplo
Byte	8 bit con signo	38
Short	16 bit con signo	23
Long	64 bit con signo	3434332
Int	32 bit con signo	70
Char	16 bit sin signo	'A'
Float	32 bit flotante con signo	1.234
Double	64 bit flotante con signo	1.234
Boolean	true o false	true
String	secuencia de caracteres	"hola"

Excepto `String`, que es del paquete `java.lang`, el resto se encuentran en el paquete `Scala`. Todos se importan automáticamente.

`def` para definir funciones con argumentos

Podemos utilizar `def` para definir funciones con parámetros (al estilo del `define` de Scheme)

Sintaxis:

```
def <nombre_funcion>(<parametro1:tipo1>,...):<tipo_resultado>={
  <cuerpo de la función>
}
```



```
}
```

Ejemplo:

```
def max(x: Int, y: Int): Int = {  
  if(x > y) x  
  else y  
}  
max: (Int,Int)Int
```

El tipo del resultado (el tipo del valor que devuelve la función) es opcional excepto en las funciones recursivas que siempre hay que indicarlo. En la función `max` podríamos no haberlo puesto y Scala lo habría inferido. Sin embargo, en una función recursiva hay que indicarlo explícitamente. Las llaves que delimitan el cuerpo de la función también son opcionales si el cuerpo sólo tiene una sentencia. La función `max` se podría haber escrito como:

```
scala> def max2(x: Int, y: Int) = if (x > y) x else y  
max2: (Int,Int)Int
```

Una vez hemos definido una función, la podemos invocar por su nombre:

```
scala> max(3, 5)  
res6: Int = 5
```

A continuación hay un ejemplo de una función que no tiene parámetros y no devuelve nada:

```
scala> def saludar()=println("Hola mundo!")  
saludar:() Unit
```

El tipo devuelto es `Unit`, significa lo mismo que `void` en Java, es decir, que la función no devuelve nada. Los métodos que devuelven `Unit` se ejecutan por sus efectos laterales. En el caso de `saludar`, el efecto lateral que produce es la salida por pantalla del saludo.

Evaluación de una función

Supongamos las siguientes funciones:

```
scala> def square(x: Double) = x * x  
square: (Double)Double  
  
scala> square(2)  
res0: Double = 4.0
```

```
scala> square(5 + 3)
res1: Double = 64.0

scala> square(square(4))
res2: Double = 256.0

scala> def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares: (Double,Double)Double

scala> sumOfSquares(3, 2 + 2)
res3: Double = 25.0
```

La evaluación de una función con parámetros se realiza de la misma forma que la evaluación de expresiones. En primer lugar, se evalúan los argumentos de la función (de izquierda a derecha). Después se aplica el cuerpo de la función sustituyendo cada parámetro formal por sus correspondientes argumentos actuales. Ejemplo:

```
sumOfSquares(3, 2+2)
⇒ sumOfSquares(3, 4)
⇒ square(3) + square(4)
⇒ 3 * 3 + square(4)
⇒ 9 + square(4)
⇒ 9+4*4
⇒ 9 + 16
⇒ 25
```

Variables

Las variables están encapsuladas en objetos, es decir, no pueden existir por sí mismas. Las variables son referencias a instancias de clases.

Para definir una variable se requiere:

- Definir Mutabilidad
- Definir el Identificador
- Definir el Tipo
- Definir un Valor Inicial

Sintaxis para definir una nueva variable:

```
<var | val> <identificador> : <Tipo> = < _ | Valor Inicial>
```

Por ejemplo:

```
val x : Int = 1
```

Donde se usa la palabra reservada `val` para indicar que la referencia no puede reasignarse, mientras que `var` para indicar que si puede reasignarse. La variable `val` es similar a una variable final en Java: una vez inicializada, no se podrá reasignar. Una variable `var`, por el contrario, se puede reasignar múltiples veces.

Todas las variables o referencias en Scala tienen un tipo, esto es debido a que el lenguaje es estrictamente tipado. Sin embargo los tipos pueden en muchos casos omitirse, porque el compilador de Scala tiene inferencia de tipos. Por ejemplo, las siguientes definiciones son equivalentes:

```
var x : Int = 1
var x = 1
```

Otros ejemplos:

```
val msg = "Hola mundo!"
msg: java.lang.String = Hola mundo!
```

En este ejemplo vemos que Scala tiene inferencia de tipo, es decir, al haber inicializado la variable `msg` con una cadena, Scala asocia el tipo de `msg` a un `String`. Si intentamos modificar el valor de `msg`, no podremos, porque lo hemos definido como `val`:

```
scala> msg = "Hasta luego!"
<console>:5: error: reassignment to val
msg = "Hasta luego!"
```

Si imprimimos el valor de `msg`:

```
scala> println(msg)
Hola mundo!
```

Si quisiéramos reasignar el valor de una variable utilizaríamos `var`:

```
scala> var saludo = "Hola mundo!"
saludo: java.lang.String = Hola mundo!
scala> saludo = "Hasta luego!"
saludo: java.lang.String = Hasta luego!
```

Listas

Las listas son estructuras que contienen elementos del mismo tipo. Ejemplos:

```
val fruit = List("apples", "oranges", "pears")
```

```
val nums = List(1, 2, 3, 4)
val diag3 = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty = List()
```

Las listas en Scala, al igual que en Scheme, tienen 2 características:

- Son inmutables
- Tienen una estructura recursiva

Para crear e inicializar una lista en Scala:

```
val unoDosTres = List(1, 2, 3)
```

Operaciones sobre listas

Operador `cons`

El operador `cons` en Scala es el operador `::` (también se pronuncia `cons`, como en Scheme).

Este operador añade un nuevo elemento al principio de la lista, y devuelve la lista resultante (creando una nueva). Ejemplo:

```
val dosTres = List(2, 3)
val unDosTres = 1 :: dosTres
```

La lista vacía es `Nil`, una manera de inicializar nuevas listas es crearlas elemento a elemento con el operador `cons`, con `Nil` como último elemento, al igual que en Scheme, donde una lista se construye como `(cons 1 (cons 2 (cons 3 '())))`. En Scala la misma lista se construye como `1 :: (2 :: (3 :: Nil))`.

Más ejemplos:

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3 = (1 :: (0 :: (0 :: Nil))) ::
  (0 :: (1 :: (0 :: Nil))) ::
  (0 :: (0 :: (1 :: Nil))) :: Nil
val empty = Nil
```

En Scala se pueden eliminar los paréntesis:

```
val nums = 1::2::3::4::Nil
```

Operaciones básicas sobre listas

- `head` : devuelve el primer elemento de la lista (como `car` de Scheme)
- `tail` : devuelve una lista que contiene todos los elementos de la lista menos el primero (como `cdr` de Scheme)
- `isEmpty` : devuelve true si la lista está vacía (como `null?` de Scheme)

Estas operaciones están definidas como métodos del objeto lista.

Concatenación de listas

El operador `:::` concatena dos listas (equivalente a `append` de Scheme). El resultado de `(x :: y)` es una lista que contiene los elementos de `x` seguidos por los elementos de `y`.

Cadenas

Como hemos visto, una cadena se define como una secuencia de caracteres encerrada entre comillas: "hola"

Concatenación con operador +

Para concatenar cadenas se utiliza el operador `+`, como el `string-append` de Scheme.

```
"hola" + "adios" ⇒ "holaadios"
```

Funciones head, tail y charAt

Se utilizan el método `head` y `tail`, como en las listas.

```
"hola".head ⇒ h  
"hola".tail ⇒ "ola"  
"hola".charAt(0) ⇒ 'h'
```

Tuplas

Las tuplas son secuencias ordenadas de objetos, similares a las listas en el sentido de que son inmutables. Las tuplas en Scala, se utilizan para agrupar de manera ordenada objetos, que en general son de tipos diferentes. El uso ideal de una tupla es cuando se quiere retornar múltiples objetos en la invocación de un método. Es un tipo inmutable, al igual que las listas.

La sintaxis para definir una tupla de `n` tipos es:

```
(tipo1, tipo2, ..., tipoN)
```

Métodos de acceso al elemento de la tupla `._1`, `._2`, ...

Ejemplo con una tupla de tres elementos:

```
val trio = (99, "Hola", true)
println(trio._1)
println(trio._2)
println(trio._3)
```

Muy útil para funciones que tienen que devolver más de un elemento. Ejemplo:

```
def sumaCadenas(s1: String, s2: String): (String, Int) =
    (s1+s2, s1.length+s2.length)
```

Expresiones condicionales

Expresiones `if-else`

El `if-else` de Scala permite seleccionar entre dos alternativas. Hay que tener cuidado con los fines de línea si no utilizamos llaves.

Correcto:

```
def abs(x: Double) = if (x >= 0) x else -x
```

Incorrecto:

```
def abs(x: Double) =
    if (x >= 0) x
    else -x
```

Correcto:

```
def abs(x: Double) =
    if (x >= 0) x else
    -x
```

Es posible anidar sentencias `if`. Ejemplo:

```
def entre(x: Double, x1: Double, x2: Double) =
    if (x < x1) false else
    if (x > x2) false else
    true
```

Expresiones `match`

Las expresiones `match` de Scala se utilizan para seleccionar entre una lista de alternativas, del estilo del cond de Scheme. Ejemplo:

```
var myVar = 3;

myVar match {
  case 1 => "Uno";
  case 2 => "Dos";
  case 3 => "Tres";
  case 4 => "Cuatro";
}
```

Recursión en Scala

Vamos a ver algunos ejemplos sencillos de recursión en Scala. Tenemos que tener en cuenta que hay que indicar explícitamente el tipo devuelto por la función, es el único caso en que Scala no puede inferirlo.

Factorial:

```
def factorial(n: Int) : Int = {
  if (n == 0) 1
  else n * factorial(n-1)
}

factorial: (n: Int) Int
factorial(4) => 24
```

Sumar los elementos de una lista:

```
def sumaLista(lista: List[Int]) : Int = {
  if (lista.isEmpty) 0
  else lista.head + sumaLista(lista.tail)
}

sumaLista: (l: List[Int]) Int
sumaLista(List(2,3,8,1,6)) => 5
```

Elevar al cuadrado los elementos de una lista:

```
def cuadradoLista(l: List[Int]) : List[Int] = {
  if (l.isEmpty) Nil
  else (l.head * l.head) :: cuadradoLista(l.tail)
}
```

```
cuadradoLista: (l: List[Int])List[Int]

scala> cuadradoLista(List(1,2,3,4,5,6,7,8))
res3: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64)
```

Buscar en una lista de tuplas:

Función que busca una clave en una lista de tuplas (del estilo de una lista de asociación en Scheme) y devuelve su valor asociado:

```
def getValor(l: List[(Char, Int)], c: Char): Int = {
  if (l.isEmpty) -1
  else if (l.head._1 == c) l.head._2
  else getValor(l.tail, c)
}
getValor: (l: List[(Char, Int)], c: Char)Int

scala> val lista = List(('a',1),('b',4),('d',8))
lista: List[(Char, Int)] = List((a,1), (b,4), (d,8))

scala> getValor(lista, 'b')
res4: Int = 4

scala> getValor(lista, 'z')
res5: Int = -1
```

Función que ordena una lista de números en orden ascendente:

```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))
```

La función `insert` defínela tú (como ejercicio).

Interacción con Java

Scala es un lenguaje tremendamente poderoso que ha sabido heredar las mejores cosas de cada uno de los lenguajes más exitosos que se han conocido. Java no es la excepción, y comparte muchas cosas con éste. La diferencia que vemos es que para cada uno de los conceptos de Java, Scala los aumenta, refina y mejora. Poder aprender todas las características de Scala nos equipa con más y mejores herramientas a la hora de escribir nuestros programas.

Scala hace muy fácil la interacción con código Java. Todas las clases del paquete `java.lang` son importadas por defecto, mientras que otras necesitan ser importadas explícitamente. También es posible heredar de clases Java e implementar interfaces Java

directamente en Scala.

Veamos un ejemplo que demuestra esto. Queremos obtener y formatear la fecha actual de acuerdo a convenciones utilizadas en un país específico, por ejemplo Francia.

Las librerías de clases de Java definen clases de utilidades interesantes, como `Date` y `DateFormat`. Ya que Scala interacciona fácilmente con Java, no es necesario implementar estas clases equivalentes en las librerías de Scala, podemos simplemente importar las clases de los correspondientes paquetes de Java:

```
import java.util.{Date, Locale}
import java.text.DateFormat._
object FrenchDate {
  def main(args: Array[String]) {
    val ahora = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format ahora)
  }
}
```

Las declaraciones de importación de Scala parecen muy similares a las de Java, sin embargo, las primeras son bastante más potentes. Se pueden importar múltiples clases desde el mismo paquete al encerrarlas en llaves como se muestra en la primer línea. Otra diferencia es la forma de importar todos los nombres de un paquete o clase. Scala, al igual que Java, permite importar todos los nombres de un paquete o clase, pero utilizando el carácter guión bajo (`_`) en lugar del asterisco (`*`). *Eso es porque el asterisco es un identificador válido en Scala (por ejemplo podemos nombrar a un método).*

La declaración `import` en la segunda línea por lo tanto importa todos los miembros de la clase `DateFormat`. Esto hace que el método estático `getDateInstance` y el campo estático `LONG` sean directamente visibles.

Dentro del método `main` primero creamos una instancia de la clase `Date` la cual por defecto contiene la fecha actual. A continuación definimos un formateador de fechas utilizando el método estático `getDateInstance` que importamos previamente. Finalmente, imprimimos la fecha actual formateada de acuerdo a la instancia de `DateFormat` que fue “localizada”. Esta última línea muestra una propiedad interesante de la sintaxis de Scala. Los métodos que toman un solo argumento pueden ser usados con una sintaxis de infijo. Es decir, la expresión:

```
df format ahora
```

es solamente otra manera más corta de escribir la expresión:

```
df.format(ahora)
```

Lenguajes y Paradigmas de Programación, curso 2014–15

© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante

Cristina Pomares, Domingo Gallardo