TUTORIAL DE SCALA

Lenguajes de Programación
Universidad Nacional de Colombia
2017-II

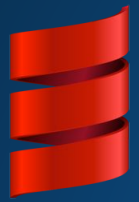
Cristian C. Lozano J. - Diana C. Navarrete R.

Introducción

Apareció en 2003 diseñado por Martin Odersky
Scala es un acrónimo de Scalable Languaje.
Lenguaje multiparadigma, compilado

Corre sobre JVM y android.

Compatible con librerías, frameworks, herramientas e IDEs que se pueden usar con java.



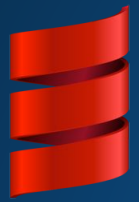
Scala Orientado a Objetos

Los valores son objetos, y las operaciones son llamados a métodos. Soporta patrones de diseño.

Compañías que usan scala

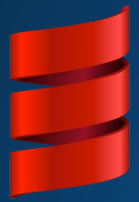
coursera

LinkedIn



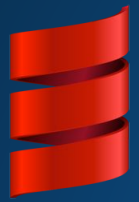
Instalando Scala - Consola

- Verificamos que esté instalado el JDK en el computador, mirando las versiones de java y javac.
- Descargamos Scala Binaries y corremos el correspondiente instalador: ‘./scala’ en carpeta bin de Binaries para Linux, o correr la extensión ‘.msi’ para Windows.
- Agregamos al PATH la dirección del archivo: ‘./scala’ para Linux y ‘C:\Program Files (x86)\scala\bin>scala.bat’ para Windows, con lo cual se puede llamar el programa desde cualquier parte.
- Para más información dirigirse a: http://www.journaldev.com/7456/installing-scala-on-linuxunix-and-windows-os



Instalando Scala - Eclipse

- Se recomienda tener la última versión del IDE Eclipse para la instalación.
- Al igual que al agregar la herramienta Antlr, se da el botón ‘Help’ -> ‘Install New Software’ -> ‘Add’
  y se agrega desde el siguiente link:
  http://download.scala-ide.org/sdk/lithium/e46/scala211/stable/site
  Dando como nombre ‘scala’.
- Iniciar proyecto con ‘File’ -> ‘New’ -> ‘Project’ -> ‘Scala Wizard’ -> ‘Scala Project’.
- Para más información:
  http://scala-ide.org/download/current.html



Scala Funcional

Preferencia de inmutabilidad
“java sin punto y coma”

- Funciones son valores de primera clase.
- Soporta funciones anónimas.
- Orden superior
- Funciones anidadas.
- Currificación
- line-oriented language
- Código inmutable.

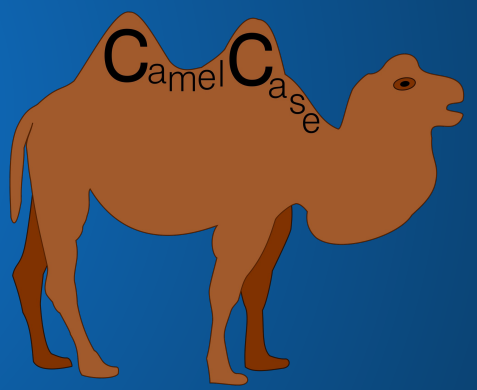
Sintaxis Básica

Case-sensitive

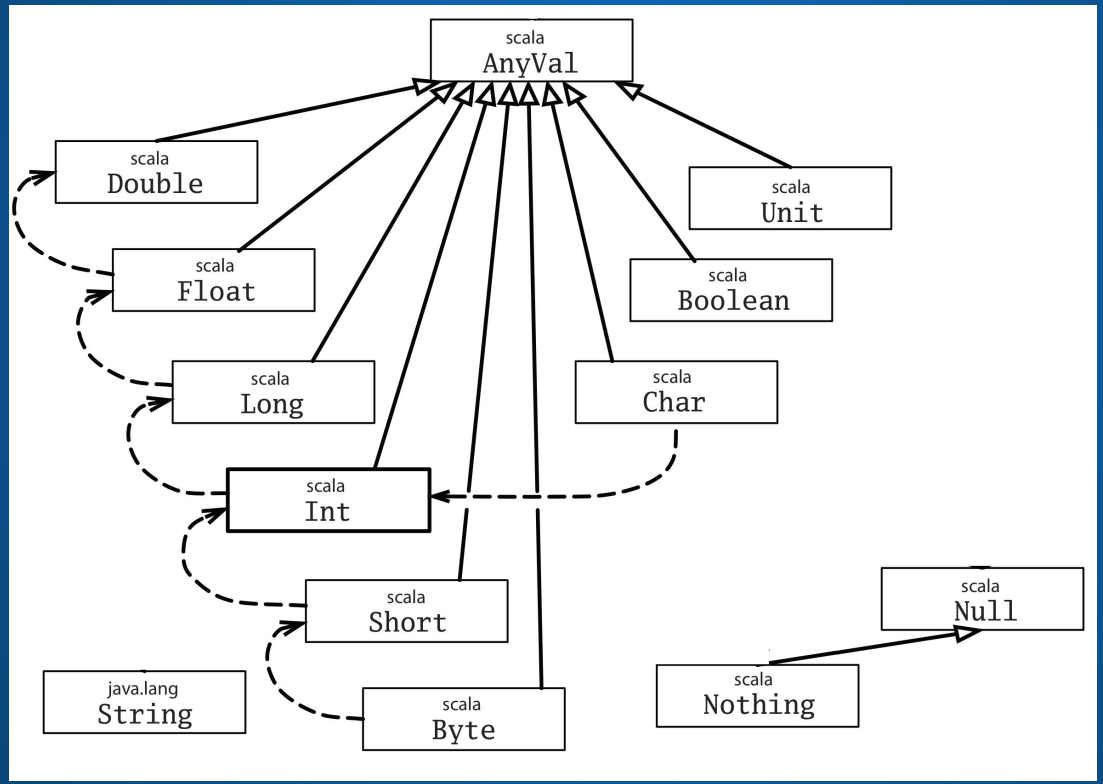
Class names -> UpperCamelCase
Method names -> lowerCamelCase
Program file name -> NameObject.scala

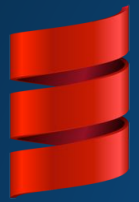
def main(args: Array[String])
java.lang.*

Comentarios -> /* ... ... */ Multilínea
// Única línea



Tipos de datos



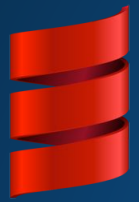


Variables y Constantes

Utilizadas para guardar información, las variables se usan para comportamientos en objetos, y las constantes en las clases (parte funcional de Scala).

```
var myVar : String = "Test"
```
variable

```
val myVal : String = "Test"
```
constante o var. inmutable



Operadores aritméticos

+ - * / % =

Operadores relacionales

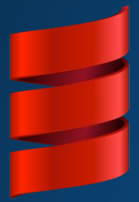
== != > < >= <=

Operadores Lógicos

&& || !

Operadores de bit

& | ^ ~ << >> >>>



Rangos:

```
2 to 6              -> inclusivo
2 until 6           -> exclusivo
2 until 6 by 2      -> con salto
```

Operaciones de String: Strings son inmutables.

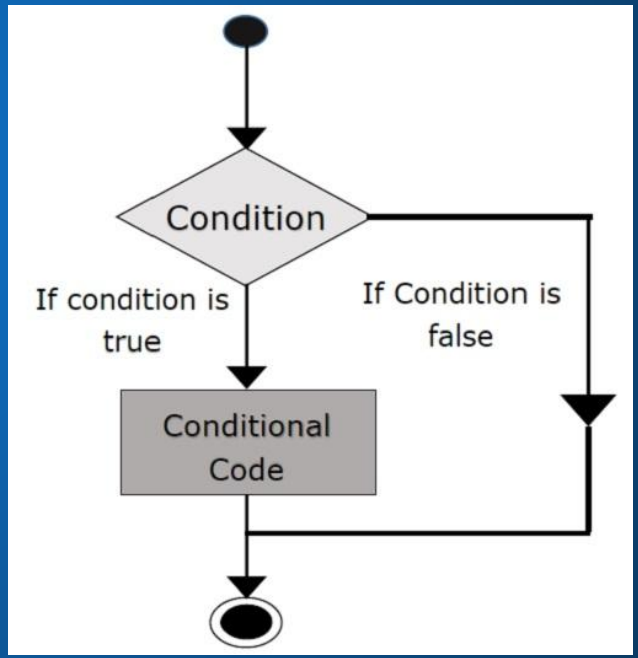
```
"scala".reverse
"scala".capitalize
"scala".length()
"scala".indesOf("c")
"sca".concat("la")      "sca" + "la"
"scala" * 3
"123".toInt
```

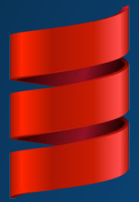
Condicionales:

```
if (condición) verdadero else falso

if (condición) { . . . }
else if {. . . }
else  {. . . }

variable match {
    case 1  =>  . . .
    case _  =>  . . .
}
```



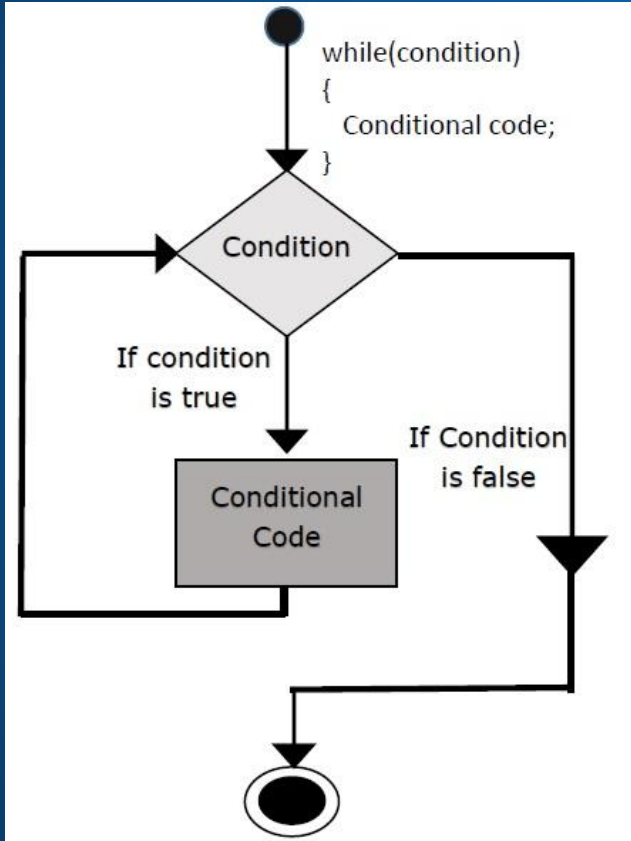


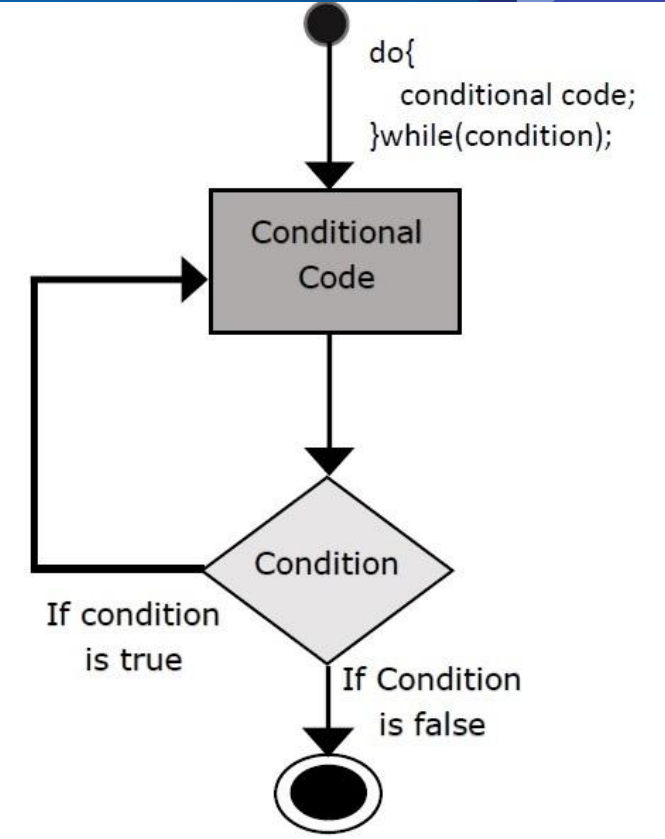
Loops:

```
for ( a <- 1 to 10) { . . .}
for ( a <- 1 until 10 { . . .}
for ( a <- 1 to 3; b <- to 3) { . . .}
for ( a <- list) { . . .}
for ( a <- list
        if a != 3; if a < 8) {. . .}

while( . . .) { . . .}

do { . . .} while ( . . .)
```


while(condition)
{
Conditional code;
}
Condition
If condition
is true
If Condition
is false
Conditional
Code


do{
conditional code;
}while(condition);
Conditional
Code
Condition
If condition
is true
If Condition
is false

Objetos:

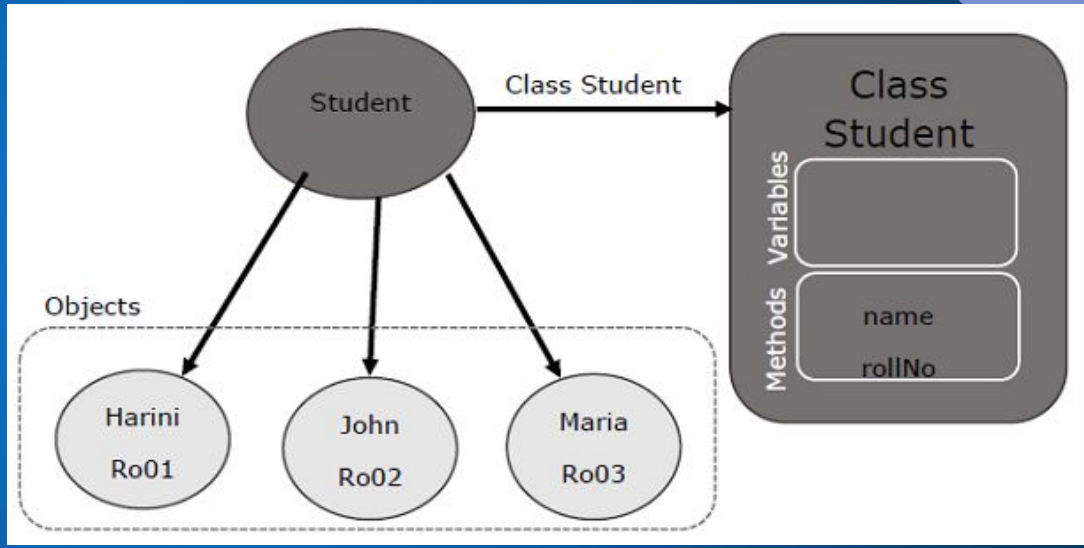
Instancia de una clase

Clase:

Abstracción del objeto.

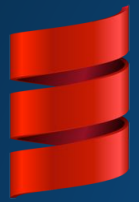
Métodos:

Define comportamientos.



Campos:

Conjunto único de variables de instancia de un objeto.



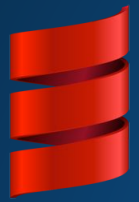
Salida:

`println()`

-formatos:

`%f` -> float
`%d` -> int
`%s` -> string

-interpolador `'s'`.
-interpolador `'f'`.
-interpolador `'raw'`.



Colecciones

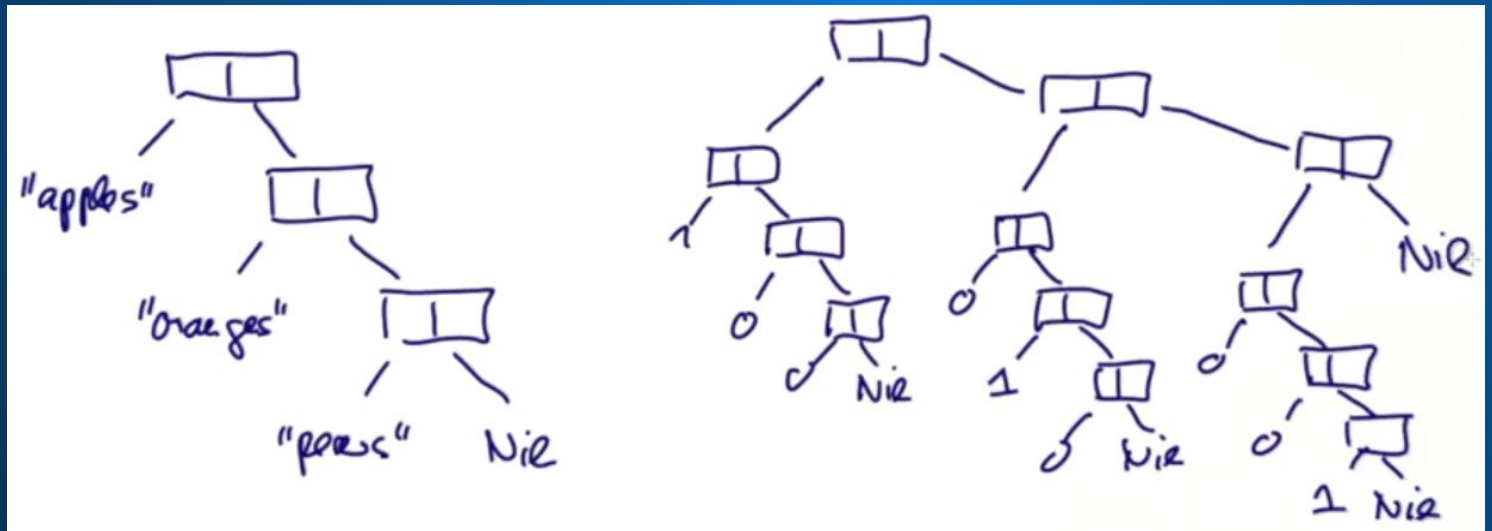
Contenedores de cosas, pueden ser mutables o inmutables.

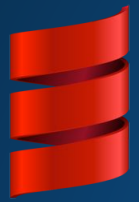
- Lists
- Sets
- Maps
- Tuples
- Options
- Iterators

Listas

Hace parte de las colecciones inmutables de Scala.

```
val fruit = List(“apples”, “oranges”, “pears”)
val diag3 = List(List(1,0,0),List(0,1,0),List(0,0,1))
```



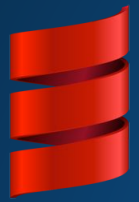


Listas

Las listas tienen el siguiente operador de construcción `::` que se usa de la siguiente manera:

`x :: xs` da una lista con primer elemento `x` seguido de los elementos de `xs`

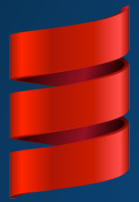
```
fruit = “apples” :: (“oranges” :: (“pears” :: Nil))
nums  = 1 :: 2 :: 3 :: 4 :: Nil
empty = Nil
```



Listas

Insertion Sort: organizar una lista de menor a mayor.

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case y :: ys => insert(y, isort(ys))
}
```

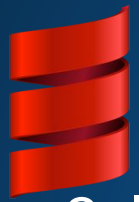


Arreglos

En Scala es mejor pensar los arreglos como un grupo de variables del mismo tipo:

```
val a = Array( 1, 2 ,3)
var a:Array[String] = new Array[String] (3)
var a= new Array[String](3)(2)
var a=range(1,5)
array(0) = 5
```

Normalmente se utilizan estructuras loop para procesar arreglos.



Sets

Colección de elementos no duplicados.
Para usar es necesario importar la clase:

```
scala.collection.mutable.Set
```

Sus operadores básicos son: `head` (retorna el primer elemento)
`tail` (retorna todos los elementos menos el primero), y
`isEmpty` (retorna verdadero si la lista está vacía, o falso en otro caso)

```
val myVal = Set(1,5,20,3,7,2)
myVal.head        myVal.tail      myVal.isEmpty
```

Resultados: 1 Set(5,20,3,7,2) false

Sets

Se pueden concatenar Sets

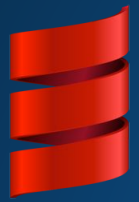
```
var conct = myVal1 ++ myVal2
var conct = myVal1.++(myVal2)
```

Encontrar maximo y minimo

```
var conct = myVal.min
var conct = myVal.max
```

Incluso valores comunes entre sets

```
var conct = myVal1.&(myVal2)
var conct = myVal1.intersect(myVal2)
```



Mapas

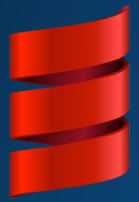
Estructura de dato que asocia llaves de tipo `Key` con valores.

```
val romanNumerals = Map("I" -> 1, "V" -> 5, "X" -> 10)

val capitalCountry = Map("US" -> "Washington",
                         "Colombia" -> "Bogotá")
```

Y se usa de la siguiente manera:

```
romanNumerals("V")              Retorna    5
capitalCountry get Colombia     Retorna    Some(Bogotá)
```



Mapas

Se pueden organizar o agrupar los datos de un mapa:

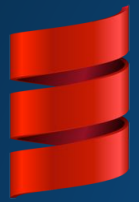
```
val fruit= List(“apple”, “pear”, “orange”)
```

Organizar con `sortWith` y `sorted` :

```
fruit.sortWith (._lenght < _.lenght)
               //List(“pear”, “apple”, “orange”)
fruit.sorted //List(“apple”, “orange”, “pear”)
```

Organizar con `groupBy` :

```
fruit.groupBy (_.head)
              //Map(p ->List(pear),
                    a -> List(apple),
                    o -> List(orange))
```



Parejas y Tuplas

Nos permite guardar y recibir múltiples datos en un valor.

```
val pair = (“answer”, 42) > pair:
val (label, value) = pair
```

donde el valor pair tiene como tipo (String, Int)

Con dichas parejas se pueden realizar tareas como división de listas, o para tener subgrupo de n tuplas:

```
case class Tuple2[T1, T2](_1: +T1 , _2: +T2 ){
  override def toString = “(“ + _1 + “,” + _2 + “)”
}
```

```
val label = pair._1
val value = pair._2
```

Parejas y Tuplas = MergeSort

```
def msort(xs: List[Int]): List[Int] = {
  val n = xs.lenght/2
  if (n==0) xs
  else{
    def merge(xs: List[Int], ys: List[Int]): List[Int] = (xs, ys) match {
      case (Nil, ys) = > ys; case (xs, Nil) => xs
      case(x : : xs1, y : : ys1) = >
          if (x<y) x : : merge(xs1, ys) else y : : merge(xs,ys1)
    }
    val (fst, snd) = xs splitAt n
    merge( msort(fst), msort(snd))
  }
}
```

Options[T]

Contenedor de cero o un elemento de un tipo dado.
Puede ser objeto `Some[T]` o `None`

La colección Options es la que define el retorno de los diferentes metodos usados en Scala, y tiene diferentes retornos dependiendo el método, por ejemplo:

En una constante tipo `Map` al usar `map.get(valor)`
El retorno será:

`Some(valor)`	Si el valor está en el map
`None`	Si el valor no está en el map

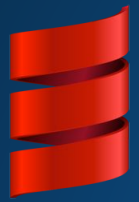
Iterators

No es propiamente una colección, sino una forma de acceder a los elementos de una colección uno por uno.

Operadores básicos: `next` y `hasNext`

```
while (myVal.hasNext){
    println(myVal.next())
 }
```

En estos se puede clasificar también `max` y `min`, `size` y `lenght` entre otros.



Traits

Su principal funcionalidad es que dichas encapsulaciones pueden ser reusadas mezclandolas en clases, y a diferencia de heredar, las clases pueden mezclarse con cualquier número de traits.

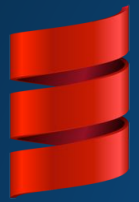
```
trait MyTrait{
      ...
  }

class MyClass(...) extends MyTrait{
      ...
}
```

Métodos o funciones.

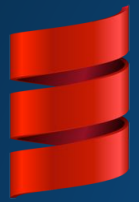
```
def nombre ([parámetros]) : [tipo de retorno ]
```

- call-by-name-
- valores de parámetros por defecto
- parcialmente aplicadas
- con argumentos nombrados
- de orden superior
- recursivas
- anónimas
- anidadas
- curring



Closure: función donde el valor retornado depende de variables definidas fuera de la función.

Traits: usado en las clases para encapsular métodos y definiciones de campo(field).

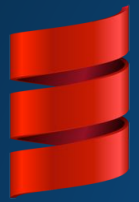


Call-by-name

Tiene como mecanismo pasar un bloque de código como llamada, y cada vez que la llamada accesa a un parámetro este código de bloque es ejecutado.

Esto quiere decir que se define el parámetro como otra función que será llamada cada vez que se necesita el parámetro.

```
funcion(otraFuncion())
```

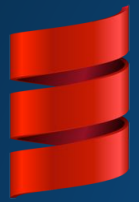


Con Argumentos Variables

Scala permite indicar a una función que el último parámetro de la función puede repetirse, permitiendo a los listas con tamaño variable a una función, ejecutando dicha entrada de forma deseada.

```
def printStrings( args:String* )
```

Aquí podemos pasar varios strings como parámetros.

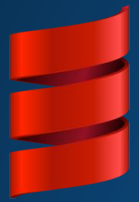


Parámetros Por Defecto

Scala permite especificar valores por defecto para parámetros de función, en cuyo caso el parámetro puede ser omitido opcionalmente al llamar la función, y se utilizarán los valores de los parámetros marcados por defecto.

```
def addInt( a:Int = 5, b:Int = 7 )
```

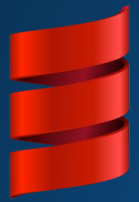
Aquí podemos llamar a la función sin parámetros.



Funciones Parcialmente Aplicadas

Cuando se invoca una función, se dice que aplique a la función a los argumentos si estos son pasados completamente, pero puede no ser necesario pasar siempre los argumentos, sobre todo si algunos de estos se repiten.

Así se crea una función que encapsula otra, y manda unos parámetros por defecto y otros directamente.

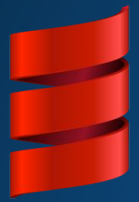


Funciones de Orden Superior

Nos permite pasar funciones como argumentos y/o retornarlos como resultado: Las funciones son valores de primera clase.

Dichas funciones que reciben o retornan otras funciones son las llamas de Orden Superior.

Las funciones de primera clase sólo manejan datos simple como enteros, longs, listas, etc.



Funciones Anónimas

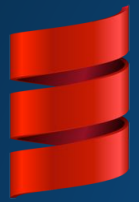
También llamadas funciones literales, al momento de ejecutarse son instanciadas en objetos llamados valores de función.

Simplemente son funciones que al ejecutar el programa son guardadas en objetos tipo var, así:

```
var inc = (x:Int) => x+1

println(inc(2))
```

Esto va a imprimir 3, siendo llamado como variable pero ejecutado como función anónima.



Currificación

Se trata de hacer la sintaxis de funciones más sencilla de escribir.

Re definición de parámetros: Minimizar parámetros entrantes a función, o reacomodación de sintaxis:

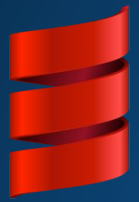
```
def sum(f: Int => Int)(a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
```

Nos permite definir cosas como `sum(cube)` sin necesidad de agregar los otros parámetros sino en un contexto necesario, haciendo el uso de la función más fácil usando funciones anónimas.

Parámetros implícitos

Nos permite redefinir funciones para ser usadas con cualquier tipo de dato:

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) = {
    ...
    merge(msort(fst)(lt), msort(snd)(lt))
}
def merge (xs: List[T], ys: List[T])=(xs,ys) match {
    ...
    case (x :: xs1, y :: ys1) = >
        if(lt(x,y)) ...
        else ...
}
```

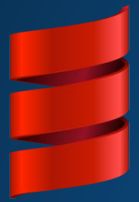


Parámetros implícitos

Y se llama de la siguiente manera:

```
val nums = List(2, -4, 5, 7, 1)
val fruts = List(“apple”, “orange”, “banana”)

msort(nums)((x, y) = x < y)
msort(fruts)((x, y) => x.compareTo(y) <0)
```

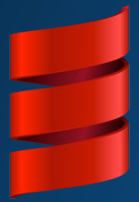


Parámetros implícitos

En Scala tenemos una librería que se encarga de dichos ordenamientos:

```
import math.Ordering

def msort[T](xs: List[T])
            (implicit ord: Ordering[T]): List[T] = {
    ...
       if(ord.lt(x,y)) ...
       else …
   merge(msort(fst), msort(snd))
}
msort(nums)
```



Recomendación

Se recomienda ir al curso de Scala en Coursera en el siguiente link:

https://www.coursera.org/learn/progfun1/home/week/1

ó

https://www.tutorialspoint.com/scala/index.htm

En ellos se desarrolla a más a profundidad el lenguaje y sus características como lenguaje de programación funcional.

Referencias

- https://www.scala-lang.org/
- https://es.wikipedia.org/wiki/Scala_(lenguaje_de_programaci%C3%B3n)
- https://www.tutorialspoint.com/scala/
- http://scala-ide.org/
- https://www.tutorialspoint.com/compile_scala_online.php
- http://www.w3big.com/es/scala/default.html
- http://www.w3big.com/es/scala/default.html
- http://www.dccia.ua.es/dccia/inf/asignaturas/LPP/2010-2011/teoria/tema6.html