



# Scala: Conceptos básicos

# Scala: Conceptos básicos

Objetivos del tema:

- Conocer cómo implementa Scala el paradigma de la Programación Orientada a Objetos
- Conocer el tipado de Scala y la inferencia de tipos
- Conocer la diferencia entre una variable definida como `var` y `val`
- Conocer los operadores disponibles en Scala, así como también conocer cómo se define una función, una sentencia o una expresión

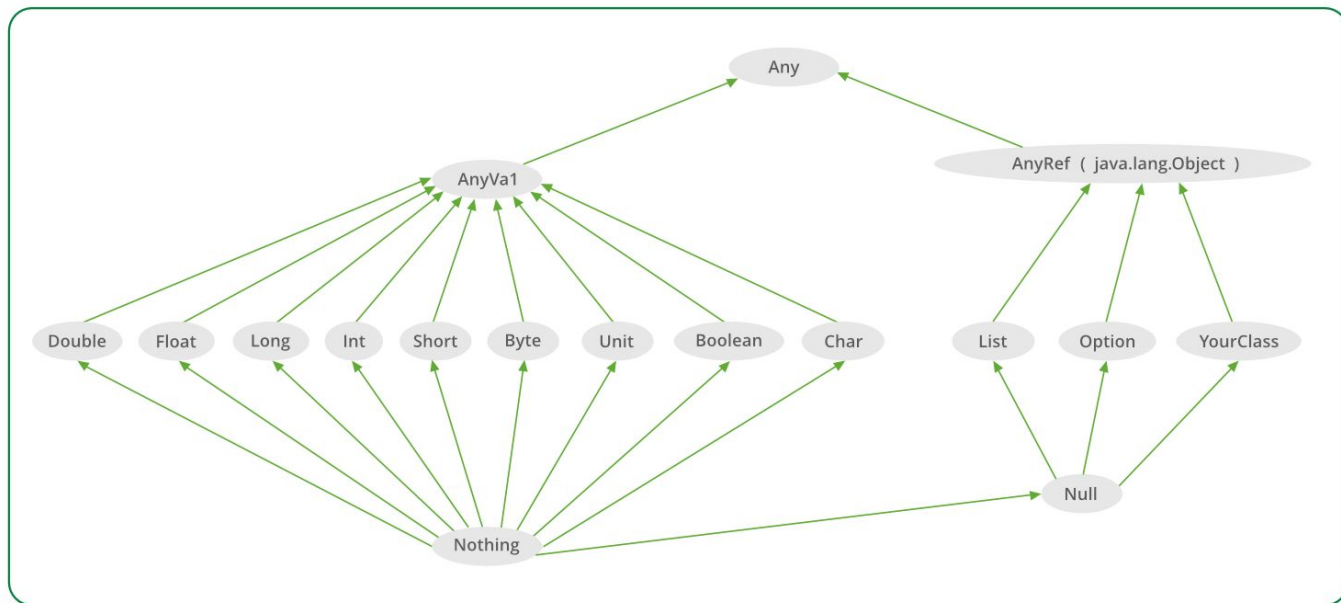


# 1 Todo es un Objeto

# Todo es un Objeto

Scala es un lenguaje puramente orientado a objetos en el sentido de que todo es un objeto, incluyendo números o funciones.

Dado que Scala está basado en clases, todos los valores son instancias de una clase. El diagrama siguiente ilustra esta jerarquía de clases:



# Todo es un Objeto

Difiere de Java en este aspecto, ya que Java distingue tipos primitivos (como boolean e int) de tipos referenciales, y no nos permite manipular las funciones como valores cosa que Scala de jacta de poder hacerlo y podemos ver que los valores primitivos de Java son Clases en Scala, como se observa en el diagrama anterior

Los números son Objetos:

- al ser objetos, también tienen métodos. La expresión aritmética como la siguiente:  
 $1 + 3 * 4 / 2$
- y es que no es más que llamadas a métodos, por lo que es equivalente a la siguiente expresión:  
 $1.+(3.*(4)./(2))$

```
scala> 1 + 3 * 4 / 2  
val res0: Int = 7
```

```
scala> 1.+(3.*(4)./(2))  
val res1: Int = 7
```

```
scala>
```

# Todo es un Objeto

Las funciones son objetos también:

- esta característica es una de las principales diferencias con Java, las funciones en Scala también son objetos.
- al ser objetos, es posible pasar funciones como argumentos, almacenarlas en variables, y devolverlas desde otras funciones.
- esta habilidad de manipular funciones como valores es una de los pilares fundamentales de un paradigma de programación llamado: programación funcional.
- Un ejemplo de por qué puede ser útil usar funciones como valores puede ser una función temporizador, cuyo propósito es realizar alguna acción cada un segundo.
  - ¿Cómo pasamos al temporizador la acción a realizar? Con una función. Este simple concepto de pasar funciones debería verse como el registro de "retrollamadas" (callback) que son invocadas cuando un evento ocurre.



# Todo es un Objeto

Veamos un ejemplo, la función del temporizador se llama `unaVezPorSegundo` y recibe una función callback como argumento.

El tipo de esta función es definido de la siguiente manera: `() => Unit` y es la definición del tipo la función especificando que no toma argumentos ni retorna valores (el tipo `Unit` es el equivalente a `void` en Java).

La función principal de este programa simplemente invoca esta función temporizador con una callback que imprime una sentencia en la terminal.

Este programa imprimirá interminablemente la sentencia "El tiempo vuela como una flecha" cada segundo.



# Todo es un Objeto

si observamos el código del ejemplo, notamos que en la sentencia:

- `unaVezPorSegundo(tiempoVuela):`  
tiempoVuela se referencia sin uso de paréntesis y esto se debe a que se trata a una función como un "valor"
- pero en la sentencia dentro del bucle `while callback():` realmente esto se traduciría a `tiempoVuela()` y aquí se le ponen los paréntesis porque se está invocando a la función.

```
object Temporizador {  
  def unaVezPorSegundo(callback: () =>  
    Unit) {  
    while (true) {  
      callback();  
      Thread sleep 1000  
    }  
  }  
  def tiempoVuela() {  
    println("El tiempo vuela como una  
flecha...")  
  }  
  def main(args: Array[String]) {  
    unaVezPorSegundo(tiempoVuela)  
  }  
}
```



## 2 Inferencia de Tipos



# Inferencia de Tipos

La inferencia de tipos es el proceso en el cual el compilador se encarga en determinar el tipo de datos del valor de una variable o de una función basándose en la sentencia que se va a evaluar.

Es una de las características más destacadas de Scala que además le ayuda a ser más conciso y elegante a la hora de escribir código en Scala.

Las variables de Scala pueden ser declaradas sin definirles el tipo de dato si contienen un valor de inicialización.



# Inferencia de Tipos

Veamos que si ejecutamos algunas sentencias en el REPL de Scala, el compilador es capaz de inferir el tipo de datos que estamos usando:

```
val num = 5
val decimal = 3.1415
var x = 1 + 2 * 3.5
List(1, 2, 3, 4)
List(1, true, "Ntic")
```

```
scala> val num = 5
val num: Int = 5

scala> val decimal = 3.1415
val decimal: Double = 3.1415

scala> var x = 1 + 2 * 3.5
var x: Double = 8.0

scala> List(1, true, "Ntic")
val res9: List[Any] = List(1, true, Ntic)

scala> List(1, 2, 3, 4)
val res10: List[Int] = List(1, 2, 3, 4)

scala> def sumaUno(x: Int) = x + 1
def sumaUno(x: Int): Int

scala> 
```



# 3 var y val

var y val son palabras reservadas en Scala que indican al compilador que se va a declarar una variable.

Las variables en Scala hacen referencia a espacios de memoria.

La memoria se reserva para almacenar el valor de una variable.

La cantidad de memoria almacenada para una variable lo determina el tipo de dato que se le asigne.

Las variables pueden ser definidas como valores o variables



# var y val

Al definir una variable con la palabra reservada **val**, ya sea si se le asigna un valor por defecto o haciendo referencia al resultado de una expresión, esta no puede volver a ser reasignada, **son inmutables**.

Al intentar reasignarle un valor a una variable val, el compilador fallará devolviendo un error:

```
scala> val immutable = 5
val immutable: Int = 5

scala> immutable = 7
      ^
      error: reassignment to val

scala> █
```



# var y val

Al definir una variable con la palabra reservada **var**, a diferencia de lo que sucede cuando se le declara a la variable como **val**, se le puede reasignar un valor nuevo, es **mutable**. Pero no se puede asignar un valor de tipo distinto al que fue inicializado.

Ambos pueden ser inicializados sin declararles un tipo ya que el compilador de Scala lo inferirá.

```
scala> var mutable = "Hola"
var mutable: String = Hola

scala> mutable = "mundo"
// mutated mutable

scala> println(mutable)
mundo

scala> mutable = 5
                ^
      error: type mismatch;
       found   : Int(5)
       required: String

scala> █
```



# 4 Los operadores



# Los operadores

Un operador es un símbolo que le dice al compilador que realice operaciones matemáticas o lógicas específicas. Scala es rica en operadores incorporados y proporciona los siguientes tipos de operadores:

- aritméticos
- relacionales
- lógicos
- binarios (operaciones de bitwise)
- de asignación





# Los operadores

- Operadores aritméticos:

+,

-,

\*,

/,

%

```
scala> 1+1
val res12: Int = 2

scala> 4-8
val res13: Int = -4

scala> 4*2
val res14: Int = 8

scala> 9/3
val res15: Int = 3

scala> 10%2
val res16: Int = 0

scala> 
```



# Los operadores

- Operadores relacionales:

==,

!=,

>,

<,

>=,

<=

```
scala> 1==1
val res17: Boolean = true

scala> 3!=4
val res18: Boolean = true

scala> 3>2
val res19: Boolean = true

scala> 3<1
val res20: Boolean = false

scala> 10 >= 20
val res21: Boolean = false

scala> 20 <= 20
val res22: Boolean = true

scala> 
```



# Los operadores

- Operadores lógicos:

&& (and)

|| (or)

! (not)

```
scala> true && false
val res0: Boolean = false

scala> false || true
val res1: Boolean = true

scala> !true
val res2: Boolean = false

scala> █
```



# Los operadores

- Operadores binarios (operaciones bitwise), son las que se realizan a nivel de bits de enteros:

& (and)

| (or)

^ (or exclusiva)

~ (complementario)

<< (desplazamiento a la izquierda)

>> (desplazamiento a la derecha)

En el ejemplo se tienen dos enteros  $x = 3$  e  $y = 9$  y conociendo su representación binaria: 0011, 1001 respectivamente:

$x \& y = 0001 \rightarrow 1$

$x | y = 1011 \rightarrow 11$

$x \wedge y = 1010 \rightarrow 10$

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

scala> val y = 9
val y: Int = 9

scala> x & y
val res0: Int = 1

scala> x | y
val res1: Int = 11

scala> x ^ y
val res2: Int = 10

scala> 
```



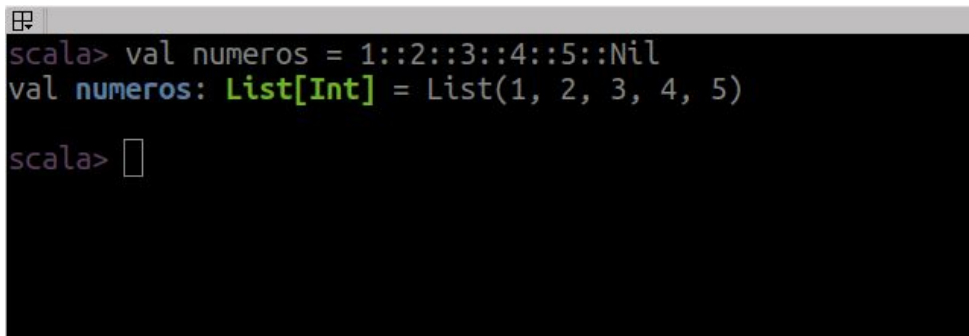
# Los operadores

Aparte de los operadores aritméticos vistos, Scala cuenta con una librería Math que cuenta con funciones como `abs`, `sqrt`, `min`, etc.

Scala también cuenta con un operador exclusivo, si lo comparamos con Java, que es 'cons' (`::`), este operador nos permite poner valores juntos y crear una lista. Por ejemplo:

```
val numeros = 1::2::3::4::5::Nil
```

- Se usa 'Nil' para indicar al compilador que se ha llegado al final de la lista.



```
scala> val numeros = 1::2::3::4::5::Nil
val numeros: List[Int] = List(1, 2, 3, 4, 5)

scala> 
```



# 5 Las Condiciones



# Las Condiciones

Scala contiene la expresión if-else. Es parecido a if-else de Java, pero usado para las expresiones también, no sólo para las sentencias.

Su sintaxis es parecida a la de Java:

```
if (<condición>) <expresión> else <expresion>
```

donde condición es una expresión booleana que se obtiene con operadores relacionales o booleanos en caso de evaluar varias expresiones booleanas.



# Las Condiciones

En el ejemplo se puede observar que:

- se compara el valor de n con 5,
- en caso de ser verdadero, se imprimirá "n es 5",
- en caso de ser falso, se imprimirá "n no es 5",
- los cuerpos de if y else no están cubiertas por llaves y esto se debe a que sólo contienen una sentencia, si hubieran más, sería obligatorio que estuvieran recubiertas por llaves {}

```
object Condicionales {  
  def main(args: Array[String]) {  
    val n = 5  
    if (n == 5)  
      println("n es 5")  
    else  
      println("n no es 5")  
  }  
}
```





# Las Condiciones

Otro ejemplo puede ser la siguiente función del cálculo del valor absoluto:

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

como se observa, la estructura if-else también se puede definir expresión que calcula el valor de retorno de la función.

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

scala> abs(-10)
val res3: Int = 10

scala> 
```



# 6 Las funciones



# Las funciones

En Scala las funciones y los métodos tiene la misma sintaxis.

Los métodos son parte de las clases.

Una función, como se ha visto anteriormente, es un objeto que se puede asignar a una variable.

Sintaxis para definir una función en Scala:

```
def nombreFuncion(parametros): <return type> = {}
```

Al definir una función si no se usa '=' Scala entenderá que es una función que no devuelve ningún valor



# Las funciones

Por ejemplo, si definimos una función `abs` que calcule el valor absoluto del parámetro de entrada la definición de la función sería:

```
def abs(x:Int) = if(x < 0) -x else x
```

- donde observamos la presencia de '=', esto le indica a Scala que la función devuelve un valor.
- se ha omitido la definición del tipo de datos devuelto por la función ya que Scala será capaz de inferirlo.
- se ve la ausencia de llaves '{' '}' que recubran el cuerpo de la función y esto se debe a que el cuerpo de la función `abs` es sólo una sentencia `if-else` que devuelve el valor



# Las funciones

Veamos otro ejemplo, la función factorial en su versión iterativa:

```
def fac(n:Int) = {  
  var r = 1  
  for(i <- 1 to n)  
    r = r * i  
  r  
}
```

- aquí se observa que también se omite la definición del tipo de retorno de la función
- y se nota también la presencia de llaves recubriendo el cuerpo de la función ya que esta vez consta de más de una sentencia



# Las funciones

Ahora otro ejemplo de la función factorial, pero de forma recursiva:

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

- en esta versión, sí se define el tipo de retorno de la función, debido a que el compilador no siempre es capaz de inferir el tipo de datos en una función recursiva



# Las funciones

En este ejemplo, en la invocación de métodos y funciones se usa el nombre de las variables explícitamente en la llamada asignándoles un valor, de la siguiente manera:

```
def imprimirNombre(nombre:String, apellido:String) = {  
    println(nombre + " " + apellido)  
}  
  
// Imprime "Charles Flores"  
imprimirNombre("Charles","Flores")  
// Imprime "John Smith"  
imprimirNombre(first = "John",last = "Smith")  
// Imprime "John Smith"  
imprimirNombre(last = "Smith",first = "John")
```

- fíjese que se llaman a las funciones pasándoles los parámetros nombrados y que al hacerlo, el orden de los parámetros no importa mientras todos los parámetros sean nombrados.



# Las funciones

El nombrar los parámetros a la hora de llamar a las funciones y métodos encaja con el poder establecer valores por defecto a los parámetros de una función o método:

```
def imprimirNombre(nombre:String = "John", apellido:String = "Smith") = {  
    println(nombre + " " + apellido)  
}  
  
// Imprime "John Jones"  
printName(apellido = "Jones")
```

- en la llamada a la función sólo se le está pasando el parámetro 'apellido' con valor "Jones", pero la llamada a la función imprimirá: "John Jones", debido a que el parámetro 'nombre' tiene el valor por defecto de "John"





# Las funciones

A una función también se le podría pasar un número arbitrario de parámetros usando el carácter especial '\*':

```
def sum(args:Int*) = {  
  var result = 0  
  for(arg <- args) result += arg  
  result  
}
```

- como se puede ver, la función sum espera un parámetro de nombre args y de tipo Int\*, lo cual indica que puede recibir 0 o más parámetros.
- en el cuerpo de la función sum, vemos que args se recorre con un bucle for, y almacenando la suma de los elementos de la lista args en la variable result que luego será devuelta como resultado.



