

UNIVERSIDAD
NACIONAL
DE COLOMBIA

PROYECTO **CULTURAL, CIENTÍFICO Y COLECTIVO** DE NACIÓN

Scala

Prof. Oscar Mauricio Salazar Ospina
omsalazaro@unal.edu.co

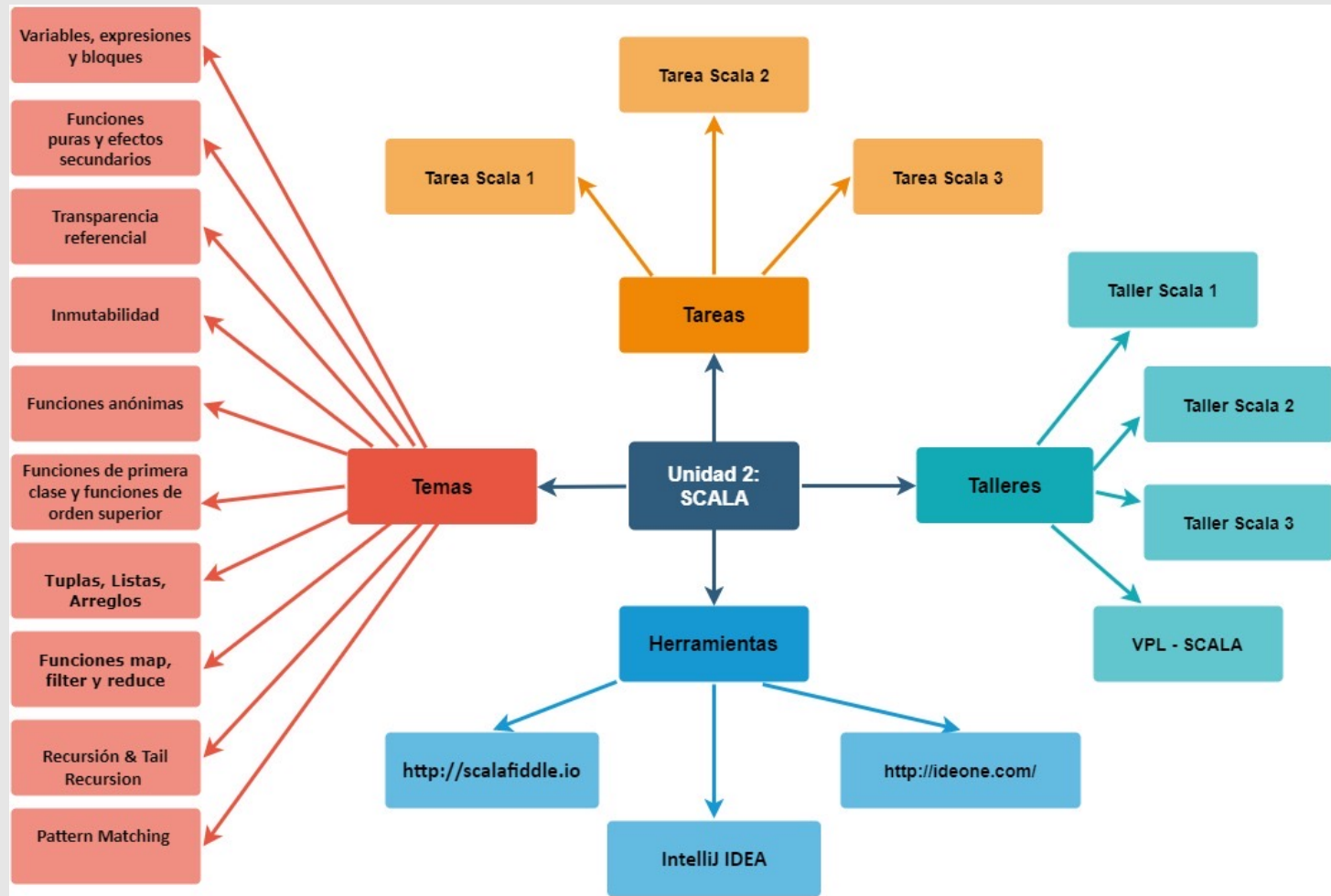
Facultad de Minas
Departamento de Ciencias de la Computación y la Decisión
Facultad de Minas

Universidad Nacional de Colombia

PROYECTO **CULTURAL, CIENTÍFICO Y COLECTIVO** DE NACIÓN

Unidad 3: Scala

Mapa conceptual



Scala

Generalidades

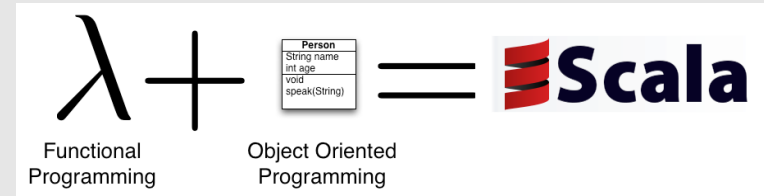
Scala es un lenguaje de programación de **código abierto** el cual se desarrolló en el año 2001, por un equipo de desarrolladores dirigidos por Martin Odersky de la universidad EPFL (École Polytechnique Fédérale de Lausanne) de Suiza.



Scala

Generalidades

- Basado en **JAVA**
- Multiparadigma
 - **Orientado a Objetos (POO):** Utiliza nociones de objeto, paquete, atributos, métodos, polimorfismo, visibilidad, etc.
 - **Funcional:** Funciones como ciudadanos de primera clase. Soporta funciones anónimas (lambda), patrones, etc.
- **First Class Citizens:** Objeto puede ser guardado en variables, puede ser pasado como argumento, puede ser usado como valor de retorno y si tiene identidad propia.



Scala

Generalidades

- Conciso, puntual.
- Declaración implícita y/o explícita.
- Elegante.
- Tipado estático.
- Compila contra la JVM y es compatible con Java.
- Lenguajes mejor pagos en los últimos años.
- Fácil de probar (test unitarios).



Scala

Generalidades

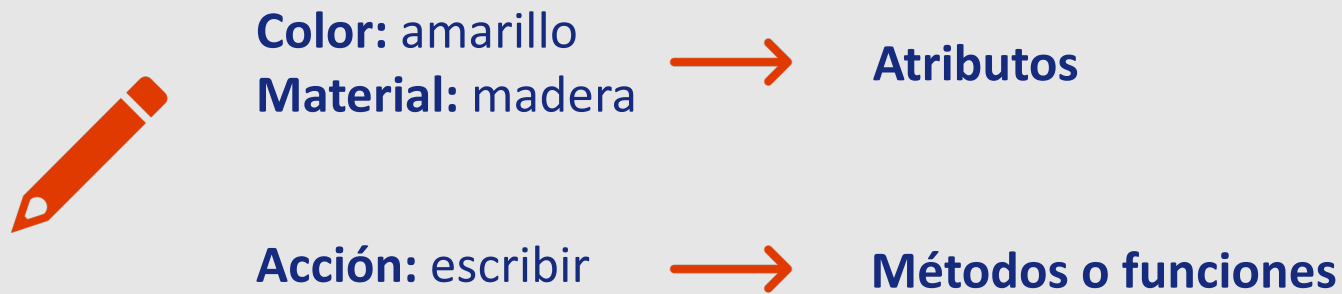
- Razonamiento Simple para el desarrollador (menos líneas de código => menos bugs => menos mantenimiento).
- Proporciona un nivel de abstracción mucho más alto.
- Fácil modularidad en implementación.
- Facilidad en la implementación de paralelismo para multiprocesamiento y computación en la nube. Lo cual Odersky denomina el principio de **"Working Hard to Keep it Simple"**.



Scala

Programación Orientada a Objetos (POO)

- Objeto: Entidad que tiene características que lo hacen diferente a otros.



Scala

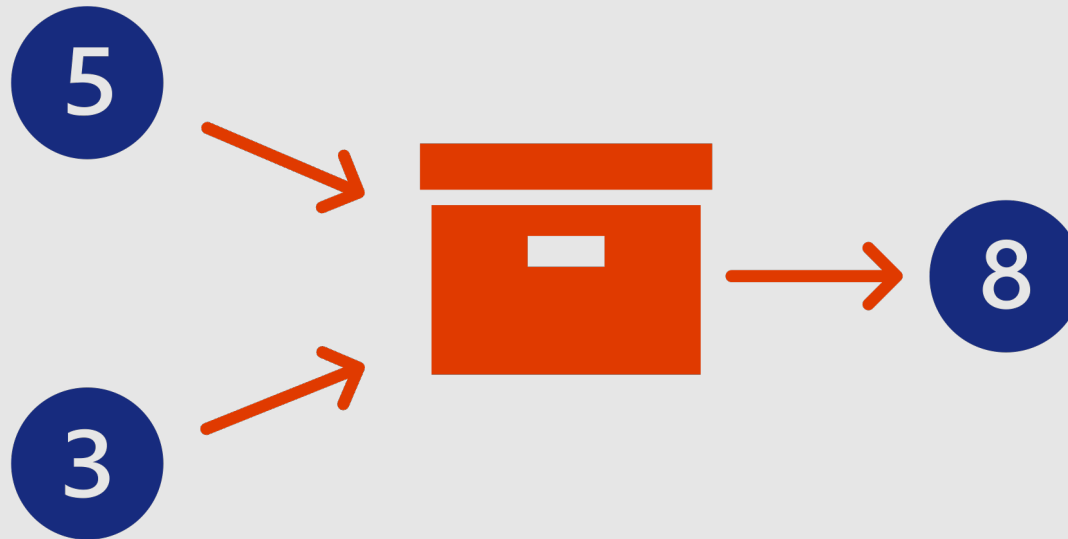
Programación Orientada a Objetos (POO)

- Clases: Moldes



Scala

Programación funcional



Scala

Elementos de la programación funcional

1. **Funciones puras y efectos secundarios.**
2. **Composición de funciones.**
3. **Transparencia referencial.**
4. **Inmutabilidad**
5. **Funciones anónimas.**
6. Funciones de primera clase y funciones de orden superior.
7. Recursión y Tail recursion.
8. Pattern Matching



Scala

1. Funciones puras y efectos secundarios

```
import scala.math._  
  
val respuesta = sqrt(4)  
  
println("La raiz de 4 es " + respuesta)
```

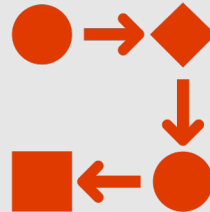


Son funciones que dadas el **mismo input**, siempre retornan el **mismo output** y no tienen efectos secundarios.



Scala

2. Composición de funciones

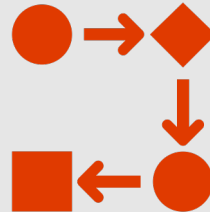


Combinar dos o más funciones teniendo como finalidad ejecutarlas en secuencia para obtener un resultado en concreto.



Scala

2. Composición de funciones



Combinar dos o más funciones teniendo como finalidad ejecutarlas en secuencia para obtener un resultado en concreto.





- Es un método para validar que tan **pura** es una función.
- Se dice que una función es **referencialmente transparente** si la podemos reemplazar por su valor correspondiente sin cambiar el comportamiento del programa.
- Esto se puede hacer con funciones **puras** porque éstas retornarán siempre el mismo valor, cuando la entrada sea la misma.
- **No efectos secundarios.**



Scala

3. Transparencia referencial

¿Se puede hacer esto con la función `sqrt(4)`?

- La función **`sqrt(4)`** se puede cambiar en el programa por su correspondiente resultado, el cual es 2.
- Porque **`sqrt()`** siempre retornará 2 siempre y cuando su entrada sea 4.
- Esto **no cambiará el comportamiento** del programa.
- La función `sqrt()` es una **función pura** porque se puede cambiar su referencia por el resultado siempre y cuando el valor de entrada sea siempre el mismo.



Scala

3. Transparencia referencial

¿Es función `rt()` es una función pura?

```
import scala.math._
//Variable global
var g = 10
//funcion en Scala
def rt(i:Int): Int = {
  g = i + g
  return g
}

val v1 = rt(5)
println(v1) // Se imprime 15

val v2 =rt(5)
println(v2) // Se imprime 20
```

15
20

SCALA ⚙️

RESULT

Scala

3. Transparencia referencial

¿Es función `rt()` es una función pura?

La función `rt()` se llamó dos veces usando el mismo valor de entrada, 5.

- El primer llamado produjo 15.
- El segundo llamado produjo 20.

¿Puedo reemplazar las referencias con 15 ó con 20?

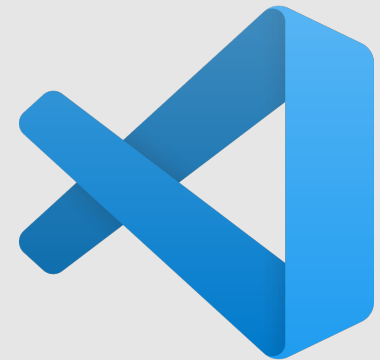
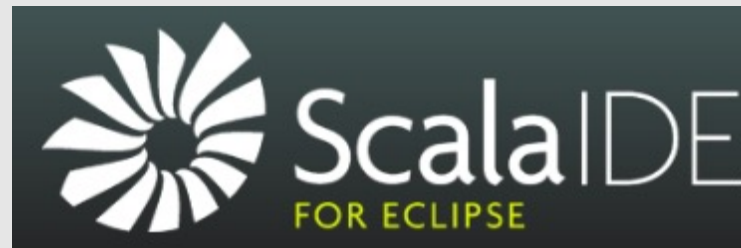
- **La respuesta es no !**

La función `rt()` no cumple con la transparencia referencial:

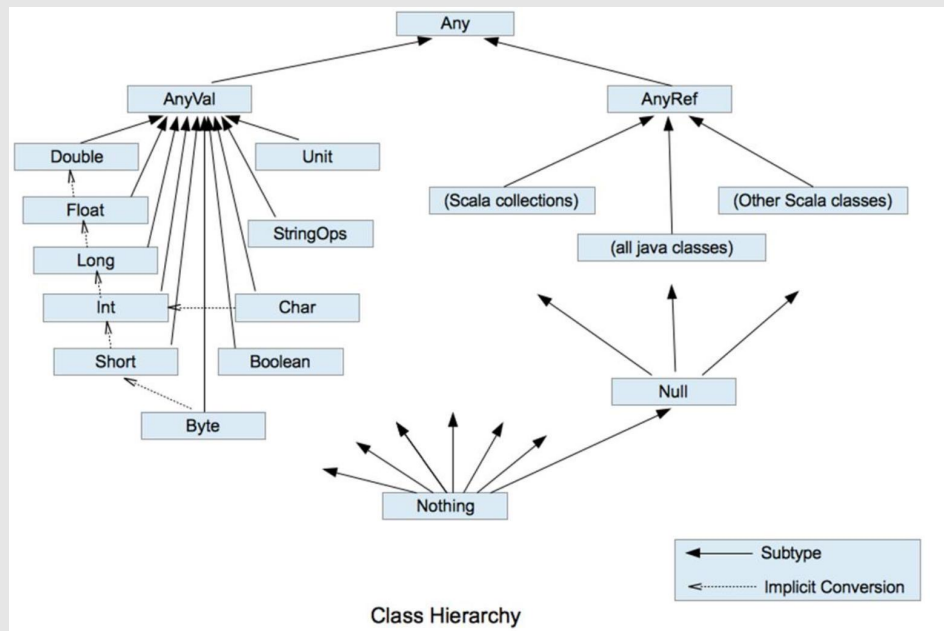
- No es una función pura.
- `rt()` depende de la variable global `g`.
- La salida no esta determinada solo por el valor de entrada.
- `rt()` modifica una variable externa a ella misma.
- Esto se conoce como **'efecto secundario'**.

Scala

Herramientas de desarrollo (IDE)



Jerarquía de tipos en SCALA




Tipo de dato	Descripción
Byte	8 bits con signo. [-128, 127]
Short	16 bits con signo. [-32768, 32767]
Int	32 bits con signo.
Long	64 bits con signo.
Float	32 bits IEEE 754 coma flotante de precisión simple.
Double	64 bits IEEE 754 coma flotante de doble precisión.
Char	Caracter Unicode de 16 bits sin signo.
String	Secuencia de caracteres.
Boolean	Verdadero o falso.
Unit	Corresponde a valor vacío. Equivale al void de Java.
Null	Valor nulo (compatibilidad con null de JVM).
Nothing	Tipo de datos fondo de la jerarquía de tipos de Scala.
Any	Cualquier superclase.
AnyRef	Cualquier clase referenciada de la clase base.

Scala

Jerarquía de tipos en SCALA

```
1 def p(a:Any): Any = {println(a+"") }
2
3 def factorial(n:Int): Int = {
4   if(n<=0)
5     return 1
6   else
7     return n*factorial(n-1)
8 }
9 p("Factorial de 5 = "+factorial(5))
10 p("Factorial de 40 = "+factorial(40))
11
12 def factorial1(n:Int): BigInt = {
13   if(n<=0)
14     return 1
15   else
16     return n*factorial1(n-1)
17 }
18 p("Factorial1 de 5 = "+factorial1(5))
19 p("Factorial1 de 40 = "+factorial1(40))
20
```



```
Factorial de 5 = 120
Factorial de 40 = 0
Factorial1 de 5 = 120
Factorial1 de 40 = 815915283247897734345611269596115894272000000000
```

Scala

Tipado en SCALA

Tipado estático

- ✓ La verificación del tipo de las variables la realiza el compilador antes de la ejecución. Es decir, el compilador **valida que todo el código este bien escrito** antes de ejecutar el código. Ejm. Haskell, Scala, Java, C.

```
1 val x = 5 * 2.5 // Tipado Implícito
2 println(x)
3
4 val y:Integer = 7 * 2.5 // Tipado Explícito
5 println(y)
```



```
ScalaFiddle.scala:4: error: type mismatch;
found   : scala.this.Int(7)
required: ?{ def $times( x$1: ? >: scala.this.Double(2.5)): ?}
```

Tipado implícito

- ✓ En **SCALA no se está obligado** (implícito) a proveer anotaciones de tipos, por ejemplo para las variables, parámetros y valores de retorno de un método / procedimiento / función. Esta característica implica que **hay inferencia de tipos al ser implícito**.

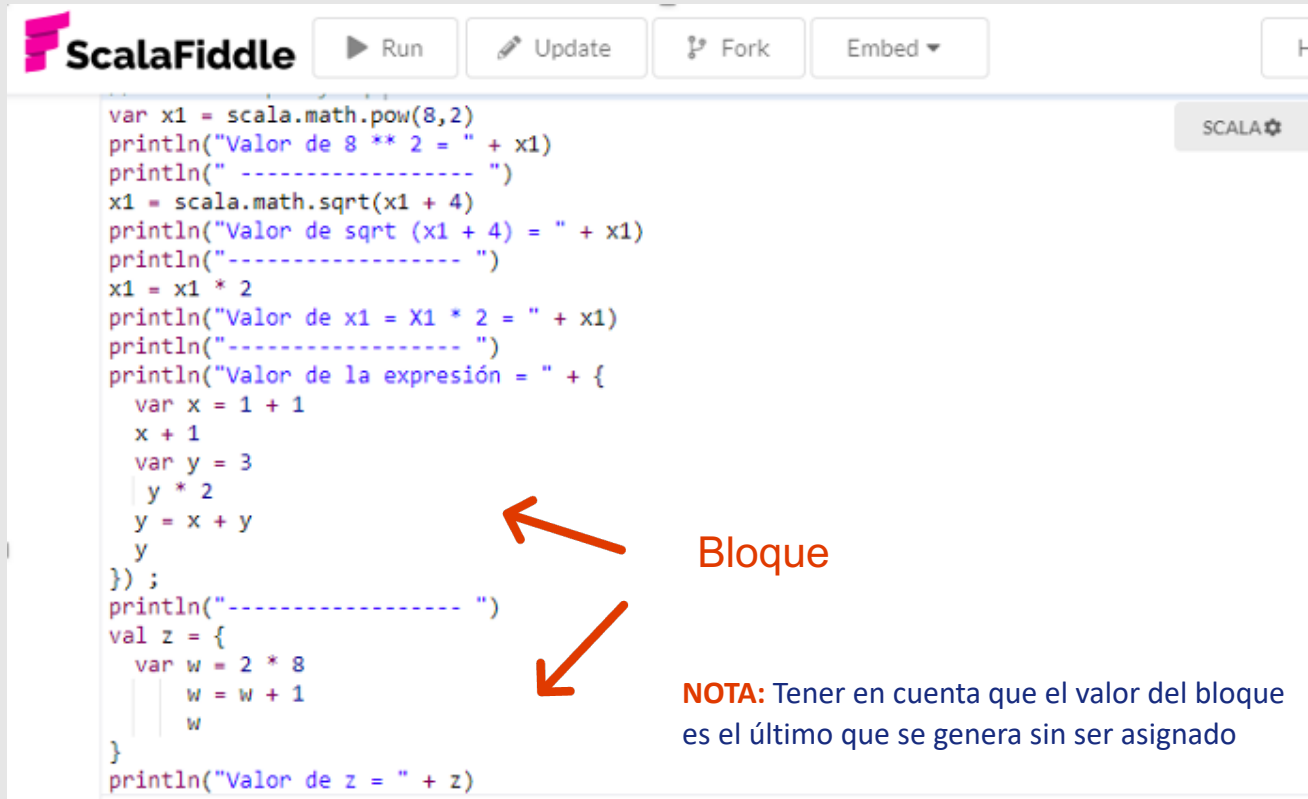
```
1 val x = 5 * 2.5 // Tipado Implícito
2 println(x)
3
4 val y:Double = 7 * 2.5 // Tipado Explícito
5 println(y)
```



SCALA ⚙️	12.5
	17.5

Scala

Como definir una función



The image shows the ScalaFiddle web interface. At the top, there are buttons for 'Run', 'Update', 'Fork', and 'Embed'. Below these is a text area containing Scala code. The code defines a variable `x1`, prints its value, updates it with a square root, prints it again, multiplies it by 2, and prints it. Then, it prints a block of code enclosed in curly braces. Finally, it prints the value of `z`, which is a `val` defined with a block of code. Two red arrows point from the text 'Bloque' and 'NOTA' to the curly-braced blocks in the code.

```
var x1 = scala.math.pow(8,2)
println("Valor de 8 ** 2 = " + x1)
println("-----")
x1 = scala.math.sqrt(x1 + 4)
println("Valor de sqrt (x1 + 4) = " + x1)
println("-----")
x1 = x1 * 2
println("Valor de x1 = X1 * 2 = " + x1)
println("-----")
println("Valor de la expresión = " + {
  var x = 1 + 1
  x + 1
  var y = 3
  y * 2
  y = x + y
  y
}) ;
println("-----")
val z = {
  var w = 2 * 8
  w = w + 1
  w
}
println("Valor de z = " + z)
```

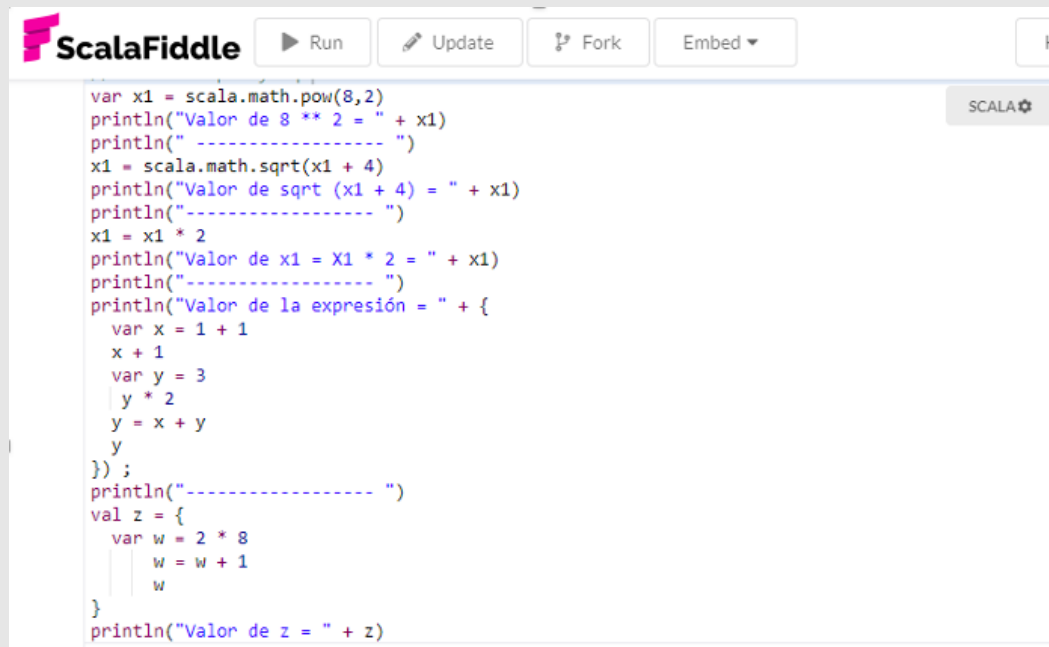
Bloque

NOTA: Tener en cuenta que el valor del bloque es el último que se genera sin ser asignado

Resultado???

Scala

Como definir una función



The image shows the ScalaFiddle web interface. At the top, there are buttons for 'Run', 'Update', 'Fork', and 'Embed'. Below these is a text area containing Scala code. To the right of the code area is a 'SCALA' button with a gear icon. The code defines a variable x1, prints its value, calculates its square root, prints it, multiplies it by 2, prints the result, defines a block expression, prints its value, and finally defines a block z and prints its value.

```
var x1 = scala.math.pow(8,2)
println("Valor de 8 ** 2 = " + x1)
println("-----")
x1 = scala.math.sqrt(x1 + 4)
println("Valor de sqrt (x1 + 4) = " + x1)
println("-----")
x1 = x1 * 2
println("Valor de x1 = X1 * 2 = " + x1)
println("-----")
println("Valor de la expresión = " + {
  var x = 1 + 1
  x + 1
  var y = 3
  y * 2
  y = x + y
  y
}) ;
println("-----")
val z = {
  var w = 2 * 8
  w = w + 1
  w
}
println("Valor de z = " + z)
```



```
Valor de 8 ** 2 = 64
-----
Valor de sqrt (x1 + 4) = 8.246211251235321
-----
Valor de x1 = X1 * 2 = 16.492422502470642
-----
Valor de la expresión = 5
-----
Valor de z = 17
```


Scala

Como definir una función

Visibilidad Definición Nombre Parámetros Tipo retorno

```
private def sayHi(name: String): Unit = {  
  println("Hola mundo " + name)  
}
```

Cuerpo

Llamados

```
sayHi(name = "Oscar")  
sayHi(name = "Pedro")
```

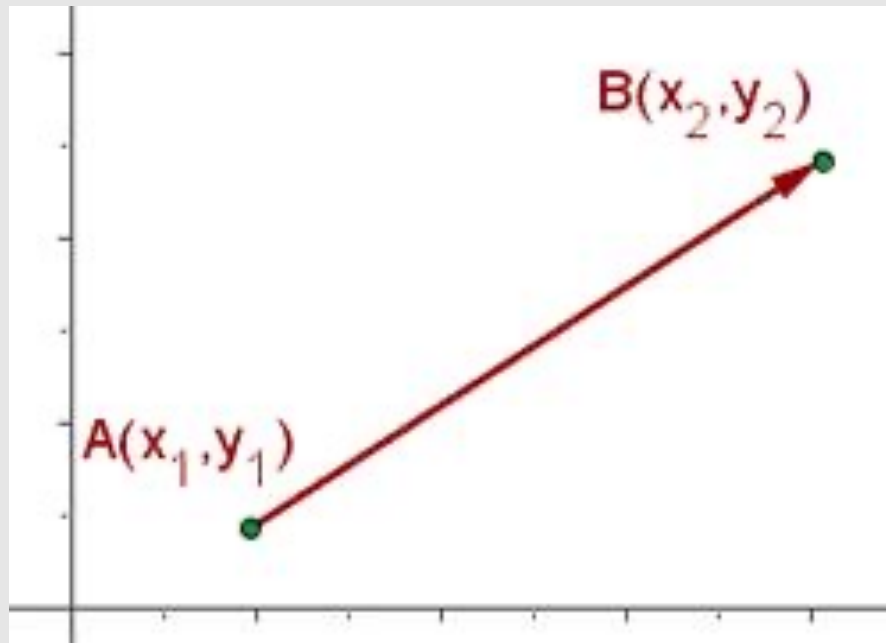
Resultado

```
Hola mundo Oscar  
Hola mundo Pedro
```

Scala

Como definir una función - ejercicio

Distancia entre dos puntos



Distancia entre dos puntos

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Distancia entre dos puntos

Vamos a hallar la distancia entre puntos A(-4; -3), B(2; 5):

$$d = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2} =$$

$$= \sqrt{(2 - (-4))^2 + (5 - (-3))^2} =$$

$$= \sqrt{6^2 + 8^2} = \sqrt{36 + 64} =$$

$$= \sqrt{100} = 10$$

Scala

Como definir una función - ejercicio

Distancia entre dos puntos

```
def distancia(x1: Int, y1: Int, x2: Int, y2: Int): Double = {  
  Math.sqrt(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2))  
}  
  
println(distancia(x1 = -4, y1 = -3, x2 = 2, y2 = 5))
```

Scala

4. Inmutabilidad



- La programación funcional se fundamenta sobre **conceptos matemáticos** que requieren de la **inmutabilidad** para seguir siendo válidos.
- Una **variable es inmutable** cuando su valor no se puede modificar (ejm. val y). Del mismo modo un objeto es inmutable cuando su estado no puede ser actualizado tras la creación del objeto.
- De esta forma nos asegurarnos que nuestra variable u objeto **no se modifica en lugares inesperados**. Evitando que se afecte la ejecución de nuestro programa.
- La **inmutabilidad** simplifica mucho el **tratamiento de la concurrencia**, en las aplicaciones multihilo. Si algo no se puede modificar, nos da igual que se acceda desde distintos hilos a la vez, así como el orden en que se haga.
- La **inmutabilidad** hace nuestro código mucho **más predecible**, porque somos más conscientes de dónde se producen los cambios de estado.

Scala

4. Inmutabilidad

```
val variable = "Prueba"  
variable = "Pepe"
```

```
val variable = Array[1,2,3,4,5]  
variable = Array[4,5,6]
```

```
var variable = Array[1,2,3,4,5]  
variable = Array[4,5,6]
```

Scala

Programación imperativa vs funcional

La **programación imperativa** se basa en:

- Modificación de variables mutables usando asignaciones. Ejm. $\text{var } g = 10$.
(después puedo tener $g = g + 12$)
- Uso masivo de estructuras de control: bucles, if-then-else, etc...

La **programación funcional** se basa en:

- Definir teorías para operadores expresadas como funciones.
- **Evitar en la medida de lo posible, el uso de mutaciones usando asignaciones** (en sentido estricto las prohíbe). Ejm. $\text{val respuesta} = \text{sqrt}(4)$ (respuesta no se puede reasignar)
- Se basa en la **abstracción y composición de funciones**.

La programación funcional se basa en teorías matemáticas en las cuales **no está permitida la mutación**.

Scala

5. Funciones anónimas

```
(i: Int) => {i * 2} :Int
```

En Scala, una función anónima tiene antes del cuerpo este **símbolo** ‘=>’ que se conoce como el **literal de una función**



Una función anónima tiene:

- Una lista de parámetros.
- Un tipo de retorno (opcional).
- Un cuerpo.
- No tiene nombre.

Scala

5. Funciones anónimas

- Como la función no tiene nombre, **es necesario asignarla a una variable.**
- **Luego se puede llamar usando la variable.**
- Pero este, **no es el propósito real de las funciones anónimas.**

```
val f2 = (i: Int) => {i * 2}:Int  
println(f2(12)) // retorna => 24
```

- Las funciones anónimas son funciones que se usan en una sola línea y se llaman una sola vez, **y nombrarlas no es de utilidad.**
- Las funciones **anónimas solo se usan en una parte** y no se desea usarlas en otro lugar diferente.
- En este caso es bastante **conveniente crear funciones anónimas.**

Scala

5. Funciones anónimas

```
var arreglo = Array[Int]()  
for (w <- 1 until 10) {  
  arreglo = arreglo :+ w * w  
}  
println(arreglo.toList)
```

```
val arreglo = 1 to 10  
val nuevoArreglo = arreglo.map(numero => {  
  numero * numero  
})  
println(nuevoArreglo)
```



Eliminar elementos mayores a 10 en un array

```
var lista = Array[Int](xs = 1,2,3,4)
for ( i <- 0 to (lista.length - 1)) {
  println(lista(i))
}
```

Gracias

Universidad Nacional de Colombia

PROYECTO **CULTURAL, CIENTÍFICO Y COLECTIVO** DE NACIÓN