Módulo Profesional: Big Data Aplicado

Scala

FUNCTIONS

```scala
object WhatsAFunction extends App {

  val doble = new MyFunction [Int, Int] {
    ↑ apply
   override def apply(elemento: Int): Int = elemento * 2
  }

  println (doble(2))

}

trait MyFunction[A, B] {
  def apply(elemento: A):B
}
```

Todas las funciones en Scala son objetos

FUNCTIONS

```
//function types = Function1[A,B]

val cadenaAentero = new Function1[String, Int] {
  ↑ apply
 override def apply(string: String): Int = string.toInt
}

println (cadenaAentero("3")+ 4)
```

FUNCTIONS. Ejercicio 1

```
/*Ejercicio 1: Realiza una función con dos parámetros de tipo cadena
y el resultado es su concatenación */
```

```
def concatenar: (String, String) => String = new Function2[String, String, String] {
  ↑ apply
 override def apply(a: String, b: String): String = a + b

 println(concatenar ("Hola, ", "Scala"))
```

FUNCTIONS. Ejercicio 2

```
/*Ejercicio 2: Define una función que toma como argumentos un entero y otra función que
la cual toma otro entero y retorna otro entero
  - cuál es el tipo de esta función
  - cómo hacer esto
*/
```

```
//Function1[Int, Function1[Int,Int]]
val specialFunction: Function1[Int, Function1[Int,Int]] = new Function1[Int, Function1[Int,Int]] {
     ↑ apply
    override def apply(x: Int): Function1[Int, Int] = new Function1[Int,Int] {
         ↑ apply
        override def apply(y: Int): Int = x + y
    }
}
```

FUNCTIONS. Declaración de Tipo Value

```
//Function1[Int, Function1[Int,Int]]
val specialFunction: Function1[Int, Function1[Int,Int]] = new Function1[Int, Function1[Int,Int]] {
    override def apply(x: Int): Function1[Int, Int] = new Function1[Int,Int] {
        override def apply(y: Int): Int = x + y
    }
}
```

FUNCTION. Ejemplo: Function2 (A, B, R) ≡ (A, B) => R; Function3(A, B, C, R) ≡ (A, B, C) => R

```
val specialFunction: ((Int, Int) => Int) = new Function1[Int, Function1[Int,Int]] {
    override def apply(x: Int): Function1[Int, Int] = new Function1[Int,Int] {
        override def apply(y: Int): Int = x + y
    }
}
```

FUNCIONES ANÓNIMAS

```
// función anónima
 val fDoble = (x: Int) => x * 2
```

```
val fDoble: Int => Int = x => x * 2
```

MÚLTIPLES PARÁMETROS

```
//MÚLTIPLE PARÁMETROS
val suma = (a: Int, b: Int) => a + b
```

```
val suma: (Int, Int) => Int = (a: Int, b: Int) => a + b
```

Una **función anónima** es una función sin nombre que se define directamente como una expresión, también conocida como *lambda*.

La función anónima se declara:

- Los **parámetros** aparecen a la **izquierda** del **=>**
- El **cuerpo de la función** aparece a la **derecha** del **=>**

FUNCIONES ANÓNIMAS

SIN PARÁMETROS. Misma definición

```
//SIN PARÁMETROS
val haceAlgo = () => 3
val haceAlgo: () => Int =  () => 3
```

¿Imprime lo mismo por pantalla?

```
println(haceAlgo)
println(haceAlgo())
```

FUNCIONES ANÓNIMAS

La impresión que realiza es la siguiente:

```
println(haceAlgo)
println(haceAlgo())
```

```
println(haceAlgo) //devuelve la instancia
println(haceAlgo()) //devuelve el 3
```

Resultado de la ejecución:

```
WhatsAFunction$$$Lambda$7/0x0000000080109c7a0@71be98f5
3
```

FUNCIONES ANÓNIMAS

```
//más sintaxis
val incrementa: Int => Int = _ + 1 // equivalente x = x + 1
val sumar: (Int, Int) => Int = _ + _ //equivalente (a,b) = a + b
```

HOF (HIGHER ORDER FUNCTION) . CURRIES

En Scala, currying es una técnica funcional que permite transformar una función que toma múltiples argumentos en una secuencia de funciones que toman un solo argumento cada una.

```
val superFuncion: (Int, (String, (Int => Boolean)) => Int)
```

1º parámetro de entrada un entero (**Int,** (String,….)

2º parámetro de entrada Función (Int, **(String, (Int => Boolean)) => Int)**

HOF (HIGHER ORDER FUNCTION) . CURRIES

¿Qué imprime por consola?

Función que aplica una función n veces sobre el valor x

nVeces (f, n, x) --> f es la función, n número de veces, x es el valor

nVeces (f, 3, x) --> f(f(f(x))) = f(f, n-1, f(x))

```
def nVeces (f: Int => Int, n: Int, x: Int): Int =
  if (n <= 0) x
  else nVeces (f, n-1, f(x))

val unoMas = (x: Int) => x + 1
println(nVeces(unoMas, 10, 1))
```

HOF (HIGHER ORDER FUNCTION) . CURRIES

Analiza la función y explica

```
//nvm(f,n) = x => f(f(f....(x)))
//increment10 = nvm(nVeces,10) = x => nVeces(nVeces....(x))
//val y = increment10(1)
def nVecesMejor(f: Int => Int, n: Int): (Int => Int) =
  if (n<=0) (x: Int) => x
  else (x: Int) => nVecesMejor(f, n-1) (f(x))

val mas10 = nVecesMejor(unoMas,10)
println(mas10(1))
```

HOF (HIGHER ORDER FUNCTION) . CURRIES

```scala
//curried functions
val superAdder: Int => (Int => Int) = (x: Int) => (y: Int) => x + y
val add3 = superAdder(3) //y => 3 + y
println (add3(10))
```

SuperAdder:

- Toma como entrada un entero Int y devuelve una función Int => Int y la función de salida toma como entrada un entero y devuelve como salida un entero.
- Función anónima (lambda):
    - (x: Int) indica que toma como parámetro un entero
    - (y: Int) => x + y indica que devuelve otra función que toma un entero como entrada y devuelve la suma de x +y

HOF (HIGHER ORDER FUNCTION) . CURRIES

```
//curried functions
val superAdder: Int => (Int => Int) = (x: Int) => (y: Int) => x + y
val add3 = superAdder(3) //y => 3 + y
println (add3(10))
```

add3:
- Función *superAdder* pasa el valor 3 como x y devuelve una nueva función (y: Int)=> 3 + y
- Esta nueva función, que llámamos add3, suma 3 a cualquier número que se le pase como argumento.

Por lo tanto, add3 es una función que toma un entero y devuelve 3 + y

HOF (HIGHER ORDER FUNCTION) . CURRIES

```
//curried functions
val superAdder: Int => (Int => Int) = (x: Int) => (y: Int) => x + y
val add3 = superAdder(3) //y => 3 + y
println (add3(10))
```

println(add3(10)):

- **superAdder** es una función de orden superior porque devuelve otra función.
- **add3** es el resultado de llamar a superAdder(3), lo que devuelve una función que suma 3 a su argumento.
- **add3(10)**, que calcular 3 + 10 = 13 y lo imprime

¿Imprime lo mismo que println(add3(10))?

```
println(superAdder(3)(10))
```

HOF (HIGHER ORDER FUNCTION) . CURRIES. EJERCICIOS

```
//EJERCICIO 1
/* 1. Define una función llamada superMult que recibe:
  - Parámetro de entrada un número entero
  - Parámetro de salida devuelve una función que recibe un número entero y devuelve un número entero
  2. Define una función multPorCinco que use la función superMult para multiplicar por 5.
  3. Imprime por pantalla el resultado de llamar a multPorCinco pasándole como parámetro el valor 7.
*/
```

HOF (HIGHER ORDER FUNCTION) . CURRIES. EJERCICIOS

```
//EJERCICIO 2
/* 1. Define una función llamada superConcatenar que recibe:
  - Parámetro de entrada una cadena.
  - Parámetro de salida devuelve otra función que devuelve otra cadena como sufijo.
  La función debe concatenar ambas cadenas con un espacio entre ellas.

  2. Define una función que utilice la función superConcatener que siempre utilice el prefijo "Hola"

  3. Imprime por pantalla a la función superConcatenar.

*/
```

HOF (HIGHER ORDER FUNCTION) . CURRIES. EJERCICIOS

```
//EJERCICIO 3
/* Sistema de notificaciones que genera mensajes personalizados para los usuarios.
Implementa en Scala:
  1. Definición de un saludo inicial como Hola o Estimado cliente.
  2. Añadir el nombre del usuario
  3. Personalizar el mensaje con un contenido específico.

  Tareas a realizar:
    1. Implementa una función generaMensaje con los siguientes parámetros de entrada:
      - tipo cadena para indicar el saludo.
      - tipo cadena para indicar el nombre
      - tipo cadena para indicar el mensaje

      El resultado de la función debe devolver los tres parámetros en un solo mensaje con el siguiente formato:
        saludo, nombre: mensaje

    2. Usar la función generaMensaje para crear una nueva función saludoCliente que utilice el saludo "Estimado Cliente"

    3. Genera un mensaje completo utilizando saludoCliente para un cliente llamado Carlos con el contenido
    "Su pedido ha sido enviado con éxito"

    4. Genera e imprime otro mensaje directamente utilizando generaMensaje con los valores:
      "Hola", "María", "Gracias por registrarse en nuestra plataforma"
    */
```