



# Scala Funcional: Funciones de primer orden y Pattern Matching

# Scala Funcional: Funciones de primer orden y Pattern Matching

Objetivos del tema:

- Conocer la sintaxis funcional en Scala
- conocer lo que son las funciones de orden superior
- Pattern Matching



# 1 Scala y el paradigma Funcional



# Scala y el paradigma Funcional

Veamos qué características tiene Scala con la que se pueda hacer una programación funcional:

- Scala mezcla la orientación de los objetos con las características funcionales.
- En un lenguaje de programación funcional, las **funciones son ciudadanos de primera clase** que pueden ser pasadas y manipuladas como cualquier otro tipo de datos.
- Esto es muy útil siempre que se quiera pasar alguna acción que sea ejecutada, en un lenguaje funcional, simplemente envuelves esa acción en una función que pasas como un parámetro.
- Se pueden crear funciones anónimas y pasarlas a otras funciones.



# Scala y el paradigma Funcional

- Una función también puede ser devuelta por otra función.
- El tener funciones como ciudadanos de primera clase es una manera flexible de componer programas.
- En las colecciones existen métodos que toman funciones como parámetros que se aplican a cada elemento de la colección, por ejemplo, la función map, foreach, o filter.



# Scala y el paradigma Funcional

Veamos un ejemplo con las colecciones que ya conocemos: Seq

En el ejemplo vemos que a la secuencia definida 'seq':

- `seq.filter(x => x > 5)`: filtra los elementos que cumplan con la función: `x => x > 5`
- `seq.map(x => x * 2)`: aplica la función: `x => x * 2` a cada elemento de la colección y devuelve los resultados en una nueva colección
- `seq.foreach(println)`: aplica el método `println` e imprime por pantalla cada valor de la colección

```
scala> val seq = Seq(6,4,3,9,2,11,1)
val seq: Seq[Int] = List(6, 4, 3, 9, 2, 11, 1)

scala> seq.filter(x => x > 5)
val res44: Seq[Int] = List(6, 9, 11)

scala> seq.map(x => x*2)
val res45: Seq[Int] = List(12, 8, 6, 18, 4, 22, 2)

scala> seq.foreach(println)
6
4
3
9
2
11
1

scala>
```



# Scala y el paradigma Funcional

Como hemos podido ver en los ejemplos en scala es posible pasar las funciones como parámetros a otras funciones.

A este tipo de funciones al igual que a aquellas funciones que devuelven una función se les conoce como **funciones de orden superior**.

Pongamos el caso de que tenemos una colección de números y queremos obtener una colección sólo con los números pares de la primera colección, la versión iterativa sería:

```
def generaPares(seq: Seq[Int]): Seq[Int] = {  
  var pares: Seq[Int] = Seq.empty  
  for (n <- seq) {  
    if (n % 2 == 0)  
      pares = pares :+ n  
  }  
  pares  
}
```

```
scala> val a = Seq(1,2,3,4,5,6)  
val a: Seq[Int] = List(1, 2, 3, 4, 5, 6)  
  
scala> def generaPares(seq: Seq[Int]): Seq[Int] = {  
  |   var pares: Seq[Int] = Seq.empty  
  |   for (n <- seq) {  
  |     if (n % 2 == 0)  
  |       pares = pares :+ n  
  |   }  
  |   pares  
  | }  
def generaPares(seq: Seq[Int]): Seq[Int]  
  
scala> generaPares(a)  
val res53: Seq[Int] = List(2, 4, 6)  
  
scala> 
```



# Scala y el paradigma Funcional

Ahora veamos cómo quedaría la misma funcionalidad si lo hacemos utilizando funciones de primer orden:

Vemos que hemos definido una función `esPar` que devuelve `true` en caso de que el número es par y `false` en caso contrario.

Definimos `numeros` de tipo `Seq[Int]`.

Se llama a la función `filter` de `numeros`. '`filter`' es una función disponible en las colecciones de Scala que toma como argumento una función que devuelva un `boolean`.

La función que se le pase a `filter` se aplicará a cada elemento de la colección y creará una nueva colección con aquellos elementos que hayan devuelto `true` tras aplicársele la función que `filter` recibió como argumento; en nuestro ejemplo: '`esPar`'

```
def esPar(x: Int): Boolean = x % 2 == 0
val numeros = Seq(1,2,3,4,5,6,7,8)
numeros.filter(esPar)
```

```
scala> def esPar(x: Int): Boolean = x % 2 == 0
def esPar(x: Int): Boolean

scala> val numeros = Seq(1,2,3,4,5,6,7,8)
val numeros: Seq[Int] = List(1, 2, 3, 4, 5, 6, 7, 8)

scala> numeros.filter(esPar)
val res47: Seq[Int] = List(2, 4, 6, 8)

scala> 
```





## 2 Funciones de Orden Superior



# Funciones de Orden Superior

Tanto en matemáticas como en informática, las funciones de orden superior son aquellas que cumplen, al menos, una de estas condiciones:

- Esperan como argumento/s una o más funciones.
- Devuelven una función como resultado.

La programación funcional pretende tratar la programación como la evaluación de funciones matemáticas, paradigma muy diferente a la programación imperativa, basada en estados y en instrucciones que lo cambian. Las características funcionales de Scala nos exige una forma distinta de enfocar los problemas.

Ejemplos en matemáticas son la derivada y la integral:

$$\frac{d}{dx}(f(x)) \quad \int f(x)dx$$

— El operador diferencial  
es una función de  
orden superior

— La antiderivada de una  
función  $f$  es una  
función  $F$  tal que  $F' = f$



# Funciones de Orden Superior

Un ejemplo claro de función de orden superior sería la función `map` que está disponible en las colecciones de Scala y que toma como argumento una función que aplica a cada uno de los elementos de la colección devolviendo otra colección con los resultados de haber aplicado la función a cada elemento.

Por ejemplo, si tuviéramos una lista de números y quisiéramos generar una lista nueva duplicando el valor de los elementos de la primera lista:

- `doblar` es una función que toma un argumento de tipo `Int`, `x`, y devuelve `x*2`. En la sintaxis, la tupla que está a la izquierda `('x: Int')` de `=>` es la lista de parámetros y a la derecha estaría el valor de la expresión que es lo que se devolverá.
- en la tercera línea se pasa `doblar` a la función `map` y para ser aplicado a cada elemento de la colección.

```
val numeros = Seq(2, 7, 4)
val doblar = (x: Int) => x * 2
val doblados = numeros.map(doblar)
```

```
scala> val numeros = Seq(2, 7, 4)
val numeros: Seq[Int] = List(2, 7, 4)

scala> val doblar = (x: Int) => x * 2
val doblar: Int => Int = $Lambda$1003/1939282277@76a14c8d

scala> val doblados = numeros.map(doblar)
val doblados: Seq[Int] = List(4, 14, 8)

scala> █
```



# Funciones de Orden Superior

En Scala no se tiene porqué dar un nombre a toda función que vayas a necesitar definir, a este tipo de funciones se les conoce como funciones anónimas. Si retomamos el ejemplo anterior podemos ver que se almacena una referencia a una función:

```
val doblar = (x: Int) => x * 2 // se está almacenando la función en la variable doblar
def doblar(x: Int) = x * 2 // se está definiendo una función y manteniendo una referencia a ella
```

ambas maneras anteriores implican tener una función con 'nombre', pero:

```
// esto también es una función definida, pero "sin nombre" y sería completamente
// válido pasarlo como argumento a una función que espere un argumento de tipo función.
(x: Int) => x * 2
```



# Funciones de Orden Superior

Duplicuemos el valor de una colección de números, pero usando funciones anónimas:

- Nótese que no es necesario definir el tipo de 'x', esto se debe a que el compilador es capaz de inferir su tipo basándose en el tipo que espera la función map de una colección que ha inferido previamente el tipo de sus elementos.

```
val numeros = Seq(2, 7, 4)
val doblados = numeros.map(x => x*2)
```

```
scala> val numeros = Seq(2, 7, 4)
val numeros: Seq[Int] = List(2, 7, 4)

scala> val doblados = numeros.map(x => x*2)
val doblados: Seq[Int] = List(4, 14, 8)

scala> 
```

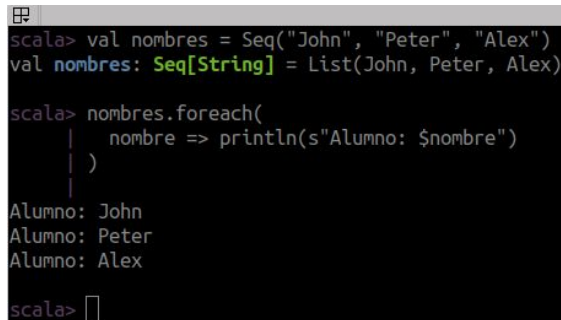


# Funciones de Orden Superior

Otro ejemplo de orden superior es la función `foreach`, también disponible para las colecciones en Scala. En este caso `foreach` espera una función que devuelva el tipo de valor `Unit`, en otras palabras, espera un método. El método que se le pase se aplicará a todos los elementos de la colección:

- por ejemplo, si tenemos una lista de nombres y queremos darle un formato: "Alumno: <nombre>"

```
val nombres = Seq("John", "Peter", "Alex")
nombres.foreach(
  nombre => println(s"Alumno: $nombre")
)
```



```
scala> val nombres = Seq("John", "Peter", "Alex")
val nombres: Seq[String] = List(John, Peter, Alex)

scala> nombres.foreach(
  |   nombre => println(s"Alumno: $nombre")
  | )
Alumno: John
Alumno: Peter
Alumno: Alex

scala> 
```



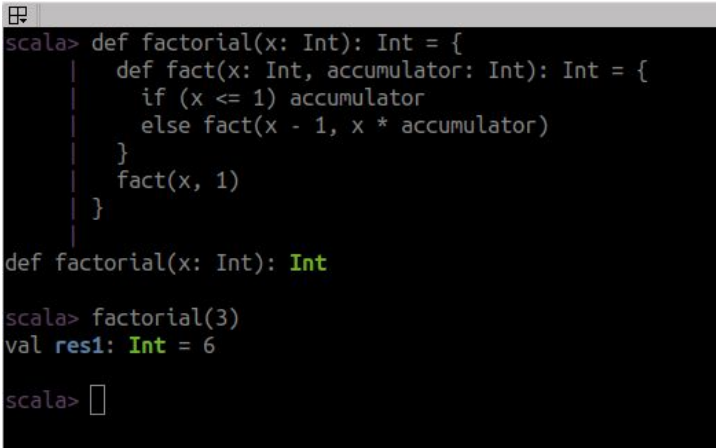
# Funciones de Orden Superior

Otra característica de Scala que nos permite componer funciones, sería el anidamiento de funciones.

Las funciones anidadas son aquellas que se definen dentro de una función.

- en el siguiente ejemplo, el cálculo del factorial de un número, se puede observar que se define la función `factorial` y en el cuerpo de la función se define `fact` la cual es llamada desde dentro de `factorial`.

```
def factorial(x: Int): Int = {  
  def fact(x: Int, accumulator: Int): Int = {  
    if (x <= 1) accumulator  
    else fact(x - 1, x * accumulator)  
  }  
  fact(x, 1)  
}
```



```
scala> def factorial(x: Int): Int = {  
  |   def fact(x: Int, accumulator: Int): Int = {  
  |     if (x <= 1) accumulator  
  |     else fact(x - 1, x * accumulator)  
  |   }  
  |   fact(x, 1)  
  | }  
def factorial(x: Int): Int  
  
scala> factorial(3)  
val res1: Int = 6  
  
scala> 
```



# 3 Pattern Matching





# Pattern Matching

Es un mecanismo para comparar un valor con un patrón.

Este mecanismo de comparación de patrones que tiene un número de aplicaciones:

- sentencias switch
- inferencia de tipo
- desestructuración

En su sintaxis, 'match' sería el equivalente a Switch de Java, pero mejorado.

Mecanismo de control en caso de que el patrón no haya coincidencia con el patrón.

Un patrón puede incluir una condición arbitraria, llamada guardia (guard)

Puedes hacer coincidir patrones de Arrays, Tuples y Case Classes, y unir partes del patrón a las variables.

Permite hacer coincidir el tipo de una expresión en lugar de usar `isInstanceOf/asInstanceOf`.



# Pattern Matching

Para definir una patrón, hay que usar la palabra reservada `match` seguido del cuerpo que incluirán los casos a evaluar

```
x match {  
  case 0 => "cero"  
  case 1 => "uno"  
  case 2 => "dos"  
  case _ => "otro" // para todos los demás casos en los que x no sea 0, 1, 2  
}
```

```
scala> val x = 0  
val x: Int = 0  
  
scala> x match {  
  | case 0 => "cero"  
  | case 1 => "uno"  
  | case 2 => "dos"  
  | case _ => "otro" // para todos los demás casos en los que x no sea 0, 1, 2  
  | }  
val res2: String = cero  
scala> 
```



# Pattern Matching

las expresiones de 'match' tiene un valor y se pueden asignar como una función:

```
def matchTest(x: Int): String = x match {  
  case 0 => "cero"  
  case 1 => "uno"  
  case 2 => "dos"  
  case _ => "otro"  
}
```

```
scala> def matchTest(x: Int): String = x match {  
  |   case 0 => "cero"  
  |   case 1 => "uno"  
  |   case 2 => "dos"  
  |   case _ => "otro"  
  | }  
def matchTest(x: Int): String  
scala> matchTest(2)  
val res3: String = dos  
scala> |
```



# Pattern Matching - Guards

Los guards de un patrón son simplemente expresiones booleanas que se usan para hacer los casos más específicos.

Hay que añadir `if <expresión booleana>` después del patrón.

```
def matchTest(x: Int): String = x match {  
  case 0 => "cero"  
  case 1 => "uno"  
  case 2 => "dos"  
  case _ if x < 10 => "otro, pero menor que 10"  
  case _ => "otro"  
}
```

```
scala> def matchTest(x: Int): String = x match {  
  |   case 0 => "cero"  
  |   case 1 => "uno"  
  |   case 2 => "dos"  
  |   case _ if x < 10 => "otro, pero menor que 10"  
  |   case _ => "otro"  
  | }  
def matchTest(x: Int): String  
  
scala> matchTest(5)  
val res4: String = otro, pero menor que 10  
  
scala> matchTest(10)  
val res5: String = otro  
  
scala> |
```



# Pattern Matching - Inferencia de tipos

En java se puede comprobar el tipo de un objeto con un tipo específico usando `isInstanceOf`

O en python usando `isinstance`

Scala ofrece una mejor manera de hacerlo es usando pattern matching evitando tener que usar `isInstanceOf`, ya que es desaconsejado:



# Pattern Matching - Match de tipos

Ejemplo con una variable de tipo Array

En el primer patrón busca un Array que contenga sólo el 0

En el segundo, que contenga dos elementos cualesquiera y almacena estos elementos en las variables x e y

En el tercer patrón sólo coincidirá en caso de que el primer elemento del array sea 0

```
val arr = Array(5,4)
arr match {
  case Array(0) => "0"
  case Array(x, y) => s"$x $y"
  case Array(0, _) => "0 ..."
  case _ => "something else"
}
```

```
scala> val arr = Array(5,4)
val arr: Array[Int] = Array(5, 4)

scala> arr match {
  |   case Array(0) => "0"
  |   case Array(x, y) => s"$x $y"
  |   case Array(0, _) => "0 ..."
  |   case _ => "something else"
  | }
val res5: String = 5 4

scala> []
```



# Pattern Matching - Match de tipos

Ejemplo con una variable de tipo List

el comportamiento es análogo al del ejemplo con Array, pero en este caso usando el operador `::` para comparar los patrones.

```
val lst = List(0)
lst match {
  case 0 :: Nil => "0"
  case x :: y :: Nil => s"$x $y"
  case 0 :: tail => "0 ..."
  case _ => "something else"
}
```

```
scala> val lst = List(0)
val lst: List[Int] = List(0)

scala> lst match {
  | case 0 :: Nil => "0"
  | case x :: y :: Nil => s"$x $y"
  | case 0 :: tail => "0 ..."
  | case _ => "something else"
  | }
val res7: String = 0

scala> List(3,4) match {
  | case 0 :: Nil => "0"
  | case x :: y :: Nil => s"$x $y"
  | case 0 :: tail => "0 ..."
  | case _ => "something else"
  | }
val res8: String = 3 4

scala> List(3,4,5) match {
  | case 0 :: Nil => "0"
  | case x :: y :: Nil => s"$x $y"
  | case 0 :: tail => "0 ..."
  | case _ => "something else"
  | }
val res9: String = something else

scala> |
```



# Pattern Matching - Match de tipos

Con tuplas también se puede realizar un pattern matching:

En este ejemplo espera recibir una tupla de dos elementos y alguno de ellos sea 0, en caso contrario devolverá: "neither is 0"

```
val pair = (0, 10)
pair match {
  case (0, _) => "0 ..."
  case (y, 0) => s"$y 0"
  case _ => "neither is 0"
}
```

```
scala> val pair = (0, 10)
val pair: (Int, Int) = (0,10)

scala> pair match {
  | case (0, _) => "0 ..."
  | case (y, 0) => s"$y 0"
  | case _ => "neither is 0"
  | }
val res11: String = 0 ...

scala> (2, 0) match {
  | case (0, _) => "0 ..."
  | case (y, 0) => s"$y 0"
  | case _ => "neither is 0"
  | }
val res12: String = 2 0

scala> (3, 4) match {
  | case (0, _) => "0 ..."
  | case (y, 0) => s"$y 0"
  | case _ => "neither is 0"
  | }
val res13: String = neither is 0

scala> |
```





# Pattern Matching - Asignación de variables

Como se ha visto anteriormente, los patrones pueden contener variables. Por lo que, los patrones se pueden usar en las declaraciones de variables:

Con esto se está definiendo simultáneamente  $x=1$  e  $y=2$

Este tipo de declaraciones suele ser útil cuando se tenga alguna función que devuelve una tupla.

```
val (x, y) = (1, 2)
```

```
scala> val (x, y) = (1, 2)
val x: Int = 1
val y: Int = 2

scala> x
val res14: Int = 1

scala> y
val res15: Int = 2

scala> 
```



