Programación funcional con Scala

Tutorial práctico: primeros pasos, funciones y colecciones

Integrantes

Daniel Aguilar Castro

Javier Esteban Martinez

Jenny Catherine Herrera

Julian Santiago Becerra

Curso & Docente

Lenguajes de Programación ·

Felipe Restrepo Calle

Universidad Nacional de Colombia

Facultad de Ingeniería

UNIVERSIDAD NACIONAL DE COLOMBIA

Objetivo de la sesión

Al finalizar este tutorial, podrás:

Reconocer Scala

Identificar las características básicas del lenguaje y su sintaxis esencial.

Aplicar inmutabilidad

Usar `val` y comprender por qué la inmutabilidad es clave en programación funcional.

Funciones de orden superior

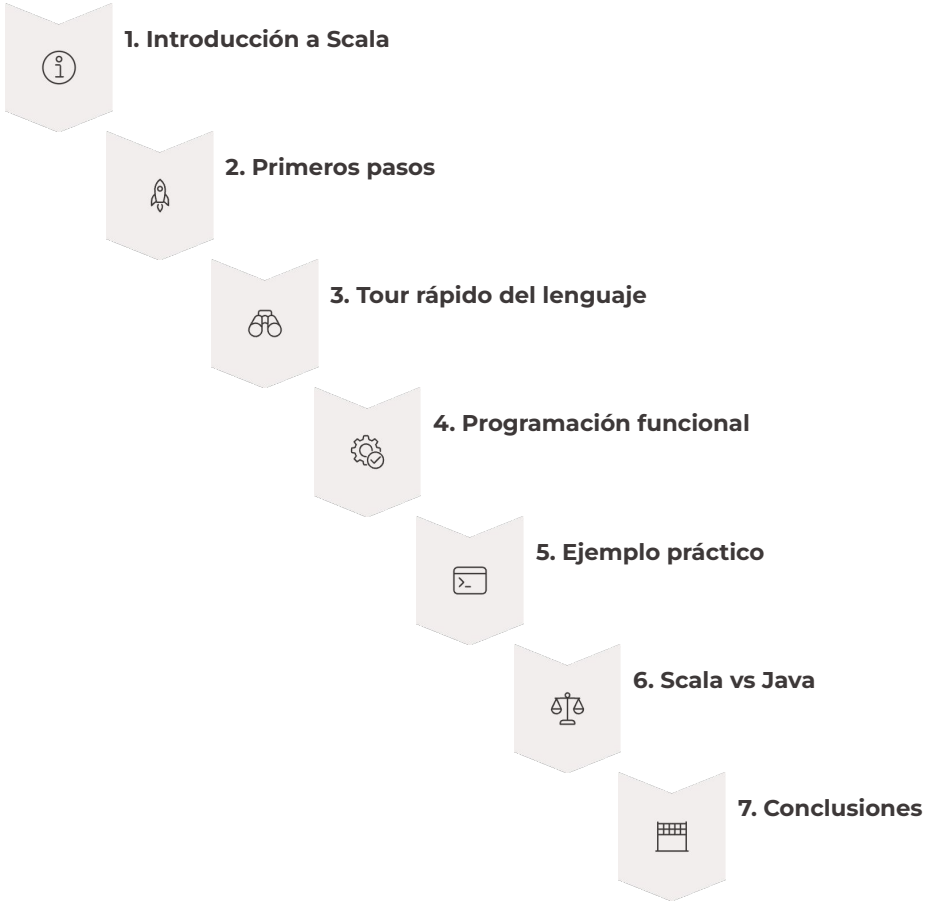
Entender y aplicar funciones puras y funciones como valores de primera clase.

Transformar colecciones

Usar `map`, `filter`, `reduce` y `flatMap` sobre colecciones.

Ruta del tutorial

Este tutorial se divide en siete etapas progresivas.



Preparación de ambiente

javiermartinezgi01/LP-2026-I-TUTORIAL-SCALA

GitHub CodeSpaces

Para agregar algo de dinamismo e interacción al tutorial se ha creado un notebook de Jupyter que tiene los pasos para configurar el ambiente de ejecución de Scala desde GitHub CodeSpaces.

Scala

jupyter

PROGRAMACIÓN FUNCIONAL CON SCALA

Introducción

- *Que es Scala*
- *Programación Funcional en Scala*

LENGUAJE MULTIPARADIGMA

¿Qué es Scala?

Scala es un lenguaje que combina **orientación a objetos** y **programación funcional** en una sola plataforma.

Diseñado por Martin Odersky

Creado en 2003 en la EPFL (Suiza).

Nombre: "Scalable Language"

Diseñado para crecer con las necesidades del programador.

Corre sobre la JVM

Interopera completamente con librerías y código Java existente.

¿Por qué Scala para programación funcional?

Código conciso

Menos boilerplate que Java para expresar la misma lógica.

Inmutabilidad

Favorece el uso de valores inmutables por defecto con `val`.

Funciones como valores

Las funciones son ciudadanos de primera clase: se pasan y retornan.

Colecciones potentes

API rica con `map`, `filter`, `reduce` y más.

Seguridad de tipos

Sistema de tipos estático que previene errores en tiempo de compilación.

PROGRAMACIÓN FUNCIONAL CON SCALA

Primeros Pasos Scala

- *Donde ejecutarlo*
- *Hello World*
- *Val vs Var*
- *Tipos de Datos e Inferencia*

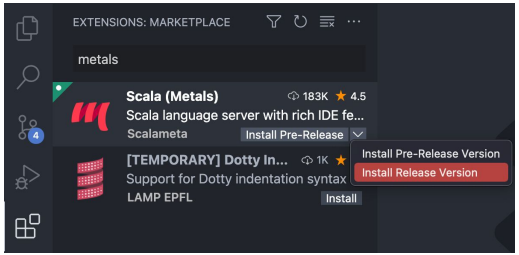
¿Dónde ejecutar Scala?

Instalación local

Descarga el SDK desde **scala-lang.org** y ejecuta desde la terminal.

IntelliJ IDEA

IDE robusto con soporte nativo para Scala mediante plugin oficial.



VS Code + Metals

Editor liviano con la extensión Metals para soporte completo de Scala.

Scastie

Entornos online: sin instalación, ideales para aprender y practicar.

Scastie

Hello World en Scala

Ejemplo de código (estructura básica)

```scala
object Main {
  def main(args: Array[String]): Unit = {
    println("Hola, Scala")
  }
}

```

```
Hola, Scala
```

`object Main`

Define un **objeto singleton** — el punto de entrada del programa.

`def main`

Método principal, equivalente al `main` de Java.

`Unit`

Tipo de retorno equivalente a `void` en Java.

`println`

Imprime un mensaje en la consola estándar.

`val` vs `var`

Ejemplo de código

```
1  val nombre = "Ana"
2  var edad = 20
3
4  edad = 21 // permitido
5  nombre = "Laura" // error: un val no se puede reasignar
6
```

valores inmutables y variables mutables

Tipos básicos e inferencia de tipos

Con anotación explícita de tipo

```
val entero: Int     = 10
val decimal: Double = 3.14
val texto: String   = "Hola"
val activo: Boolean = true
```

Con inferencia de tipos

```
val numero  = 10      // Int
val saludo  = "Hola"  // String
```

Estáticamente tipado

Scala verifica los tipos en tiempo de compilación, reduciendo errores en ejecución.

Inferencia de tipos

El compilador deduce el tipo automáticamente. El código es más limpio sin perder seguridad.

Tipos principales

`Int`, `Double`, `String`, `Boolean`, `Long`, `Char`.

Colecciones básicas en Scala

Ejemplos de declaración

```scala
val numeros = List(1, 2, 3, 4, 5)

val nombres = Vector("Ana", "Luis", "Marta")

val edades = Map(
  "Ana"  -> 20,
  "Luis" -> 22)
```

List

Lista **inmutable** y ordenada. Ideal para secuencias de datos.

Vector

Colección inmutable con **acceso eficiente por índice**.

Map

Colección de **pares clave-valor**. Inmutable por defecto en Scala.

🚀 Estas colecciones serán la base para aplicar **map, filter, reduce** y **flatMap** en la siguiente parte del tutorial.

PROGRAMACIÓN FUNCIONAL CON SCALA

Tour rápido del lenguaje

- *Expresiones*
- *Condicionales*
- *Funciones*
- *Case Clases*
- *Pattern Matching*

Expresiones

Expresiones: Muchas cosas retorna un valor

expresiones: producen un valor que puede asignarse directamente a una variable.

2+3
expresión

5
produce valor

Clave: Scala trabaja màs con expresiones que con instrucciones

Esta idea se extiende a estructuras como if, bloques de código y match:

```
val resultado = if (10 > 5) "mayor" else "menor"
val bloque = {
  val base = 10
  base * 2
}
println(s"if => $resultado, bloque => $bloque")
```

```
if => mayor, bloque => 20
resultado: String = "mayor"
bloque: Int = 20
```

Condicionales

condicionales if/else como expresión

No solo sirve para controlar el flujo del programa, también puede usarse como una expresión que devuelve un valor

```
val edad = 17
val etapa = if (edad >= 18) "adulto" else "menor"
println(s"Con $edad años, la etapa es $etapa")
```

```
Con 17 años, la etapa es menor
edad: Int = 17
etapa: String = "menor"
```

```
var etapa = ""
if (edad >= 18) etapa = "adulto"
else etapa = "menor"
```

Comparación mental:

- Imperativo → crear variable y la modifica
- Scala → producir el valor directamente

“Scala evita mutaciones innecesarias, produciendo directamente el valor que necesitamos.”

FUNCIONES

Funciones con `def` y funciones almacenadas en `val`

`def` declara un método o una función que es evaluada al invocarse.

Un `val` también puede almacenar una función como **valor de primera clase**.

```
def duplicar(x: Int): Int = x * 2
val triplicar: Int => Int = x => x * 3
println(s"duplicar(4) = ${duplicar(4)}")
println(s"triplicar(4) = ${triplicar(4)}")
```

```
duplicar(4) = 8
triplicar(4) = 12
```

Clave: Scala permite definir funciones *y* tratarlas como datos.

MODELADO DE DATOS

Case class para modelar datos

Es una de las herramientas más útiles de Scala para modelar datos inmutables, permite definir una estructura de datos de forma concisa

```
case class Persona(nombre: String, edad: Int)
val persona = Persona("Ana", 30)
println(persona)
println(s"Nombre: ${persona.nombre}, edad: ${persona.edad}")
```

```
Persona(Ana,30)
Nombre: Ana, edad: 30
defined class Persona
persona: Persona = Persona(nombre = "Ana", edad = 30)
```

Scala genera automáticamente:

Constructor
crea objetos facilmente

equals / hashCode
compara objetos por sus valores

toString
imprime el objeto de forma legible

Pattern matching
compatible

Reduce código repetitivo y favorece datos **inmutables**.

CONTROL DE FLUJO

Pattern matching con `match`

Se escribe con `match` y permite decidir qué hacer según el valor de una expresión.

> El `pattern matching` puede entenderse como una versión más poderosa y flexible de un `switch`, pero en Scala puede trabajar no solo con valores simples, sino también con tipos, estructuras de datos, tuplas, listas y `case class`.

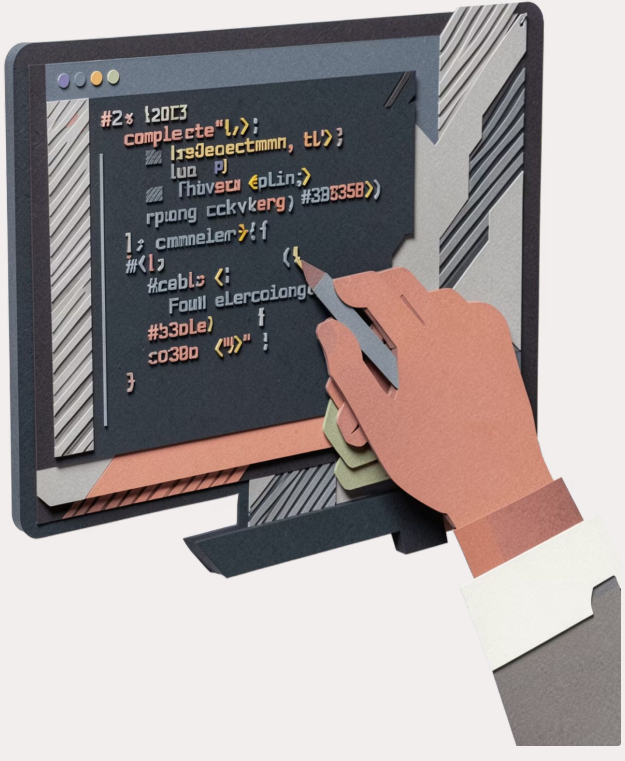
```
val numero = 2
val descripcion = numero match {
  case 1 => "uno"
  case 2 => "dos"
  case _ => "otro"
}
println(descripcion)
```

```
dos
numero: Int = 2
descripcion: String = "dos"
```

PROGRAMACIÓN FUNCIONAL CON SCALA

Particularidades del Lenguaje

- *Inferencia de tipos*
- *Lazy Val*
- *Option*
- *Recursión con @tailrec*



SISTEMA DE TIPOS

Inferencia de tipos

```
val mensaje =
  "Scala infiere este tipo
automáticamente"
val longitud = mensaje.length

println(s"$mensaje tiene
$longitud caracteres")
// ...tiene 39 caracteres
```

Scala infiere los tipos **automáticamente** sin sacrificar la seguridad: el compilador sigue verificando tipos en tiempo de compilación(verificación estática)

El código es más **limpio y legible**, sin necesidad de anotaciones redundantes.

EVALUACIÓN

Evaluación perezosa con `lazy val`

Un `lazy val` **no se calcula al declararse**: se evalúa en el primer acceso y luego conserva el valor para usos posteriores.

Útil para: cálculos costosos que solo deben ejecutarse si realmente se necesitan.

```
lazy val cargaPesada = {
 println("Calculando...")
 99
}

println("Antes del primer acceso")
println(cargaPesada)
println(cargaPesada)
// Antes del primer acceso
// Calculando...
// 99
// 99
```

SEGURIDAD

Option para manejar ausencia de datos

```
def buscarEdad(nombre: String):
Option[Int] =
  if (nombre == "Ana") Some(30) else None

val edadAna   = buscarEdad("Ana")
val edadPedro = buscarEdad("Pedro")

println(edadAna.getOrElse(0))   // 30
println(edadPedro.getOrElse(0)) // 0
```

Some(valor)

Indica que el dato **existe**

None

Indica **ausencia** del dato

⚠ **Sin null:** Option hace explícita la posibilidad de ausencia y evita errores en tiempo de ejecución.

RECURSIÓN

Recursión de cola con `@tailrec`

La anotación `@tailrec` verifica que la llamada recursiva sea la **última operación**, permitiendo al compilador optimizarla y evitar desbordamiento de pila.

Clave: Recursión más segura, eficiente y sin riesgo de `StackOverflow`.

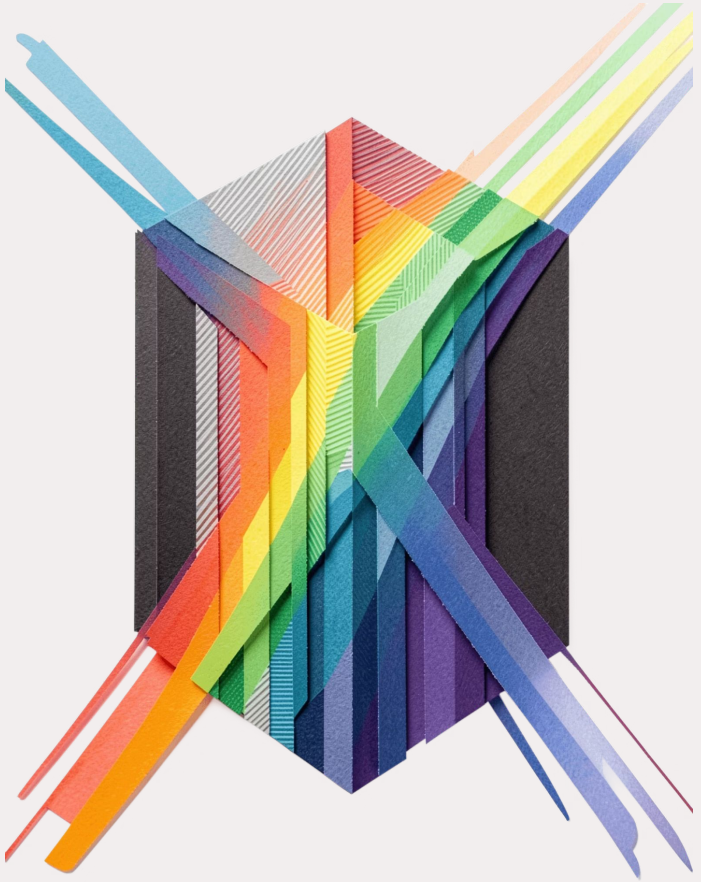
```
import scala.annotation.tailrec

@tailrec
def suma(n: Int, acumulado: Int = 0): Int =
  if (n == 0) acumulado
  else suma(n - 1, acumulado + n)

println(suma(5)) // 15
```

> Estas características preparan el camino para la programación funcional en Scala: **transformar datos, evitar mutaciones y componer funciones.**

Programación funcional en Scala

De modificar datos a transformar datos

Scala permite escribir programas más expresivos usando funciones, inmutabilidad y colecciones. En este bloque exploramos el cambio de paradigma que define la programación funcional.



El cambio de paradigma

Imperativa

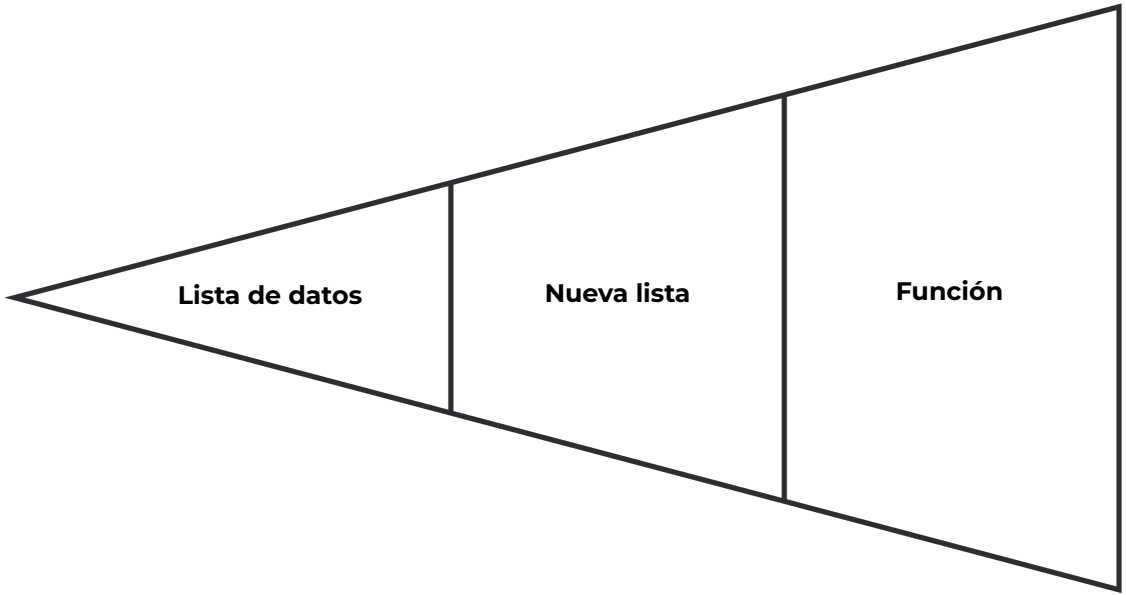
Se indica **paso a paso** cómo cambiar el estado del programa. El programador gestiona explícitamente las variables y el flujo de ejecución.

- Variables mutables
- Bucles y condiciones
- Efectos secundarios frecuentes

Funcional

Los datos se **transforman mediante funciones**. No se modifica el estado, se producen nuevos valores a partir de los anteriores.

- Valores inmutables
- Composición de funciones
- Código más predecible



Imperativo vs. Funcional: ejemplo

IMPERATIVO

```
var contador = 0
for (n <- List(1, 8,
6, 4, 17, 45, 90)) {
  if (n > 10) contador
+= 1
}
```

FUNCIONAL

```
val numeros = List(1,
8, 6, 4, 17, 45, 90)
val contador =
numeros.filter(_ >
10).length
```

En el estilo funcional se **evita modificar variables** y se expresan las transformaciones de forma declarativa y compacta.

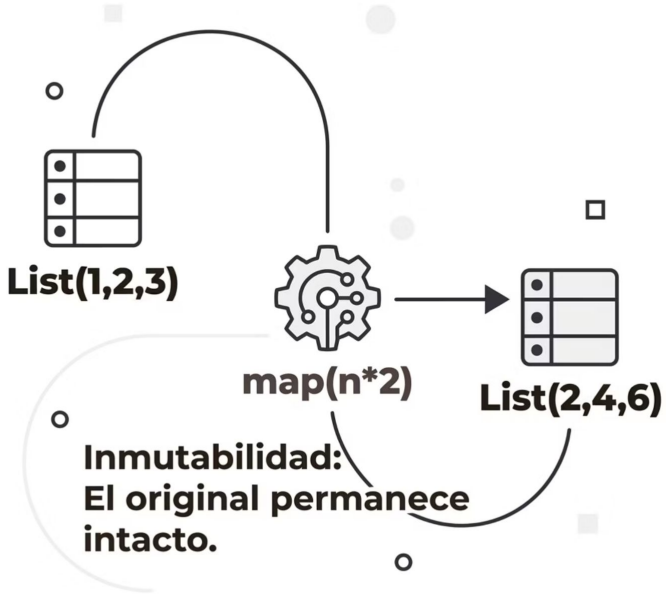
Inmutabilidad

¿Qué significa?

Un valor, una vez creado, **no se modifica**. En lugar de alterar una colección existente, se genera una **nueva colección transformada**.

```
val numeros   = List(1, 2, 3)
val duplicados = numeros.map(n => n * 2)
```

`numeros` sigue siendo `List(1, 2, 3)`.
`duplicados` es la nueva `List(2, 4, 6)`.



Funciones puras

Una función pura **siempre devuelve el mismo resultado** para los mismos datos de entrada y **no produce efectos secundarios**.

✓ PURA

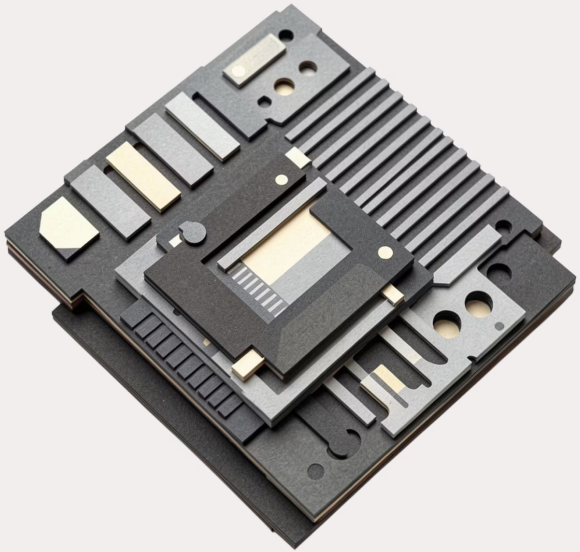
```
def cuadrado(x: Int): Int = x * x
```

Mismo input → mismo output. No interactúa con el exterior.

⚠ IMPURA

```
def mostrar(x: Int): Unit = println(x)
```

`println` escribe en consola: interactúa con el exterior y produce un **efecto secundario**.



Funciones como valores

En Scala, las funciones son **ciudadanas de primera clase**: pueden guardarse en variables, pasar como argumentos o devolverse como resultado.

```
val doble = (x: Int) => x * 2
doble(5)  // resultado: 10
```

Guardar en val

`doble` es un valor que contiene una función.

Pasar como argumento

Se puede enviar a otra función que la ejecute.

Devolver como resultado

Una función puede retornar otra función.

Funciones anónimas

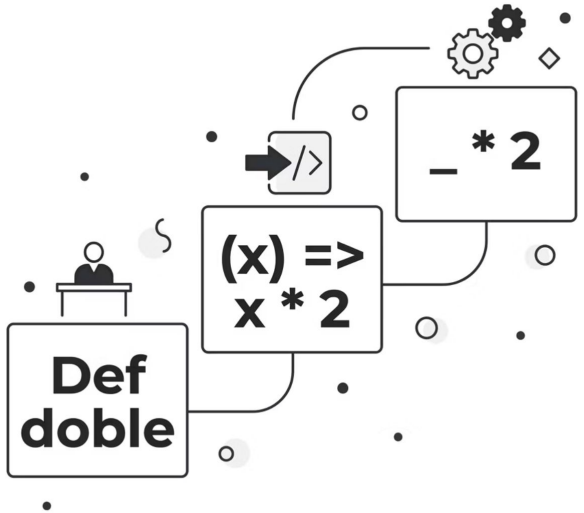
¿Qué son?

Son funciones **sin nombre**, definidas directamente donde se usan. Son muy comunes al trabajar con colecciones.

```
val numeros  = List(1, 2, 3, 4)
val resultado = numeros.map(x => x * 2)
```

Versión abreviada con **wildcard**:

```
val resultado = numeros.map(_ * 2)
```



Scala permite escribir transformaciones de manera compacta gracias al símbolo `_`.

Funciones de orden superior

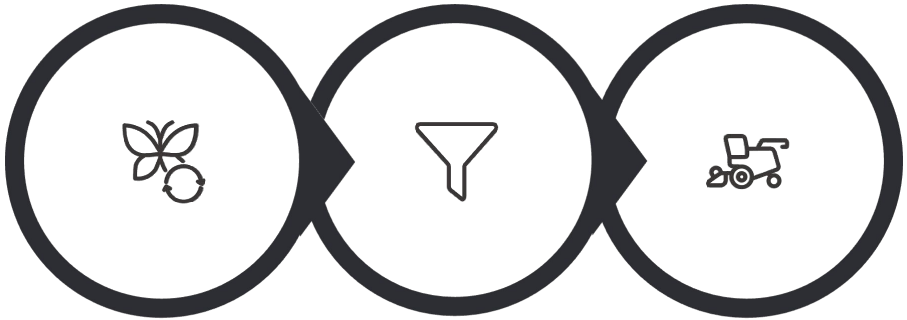
Una función de orden superior **recibe otra función como parámetro** o **devuelve una función** como resultado.

```
def aplicarOperacion(x: Int, operacion: Int =>
Int): Int = {
  operacion(x)
}

aplicarOperacion(5, x => x * 2)  // resultado: 10
```

> `map`, `filter` y `reduce` son funciones de orden superior integradas en las colecciones de Scala.

map, filter y reduce

Operación	¿Qué hace?	Ejemplo
`map`	Transforma cada elemento de la colección	`List(1,2,3).map(_ * 2)` → `List(2,4,6)`
`filter`	Selecciona los elementos que cumplen una condición	`List(1,2,3,4).filter(_ % 2 == 0)` → `List(2,4)`
`reduce`	Combina todos los elementos en un único valor	`List(1,2,3,4).reduce(_ + _)` → `10`



Map Filter Reduce

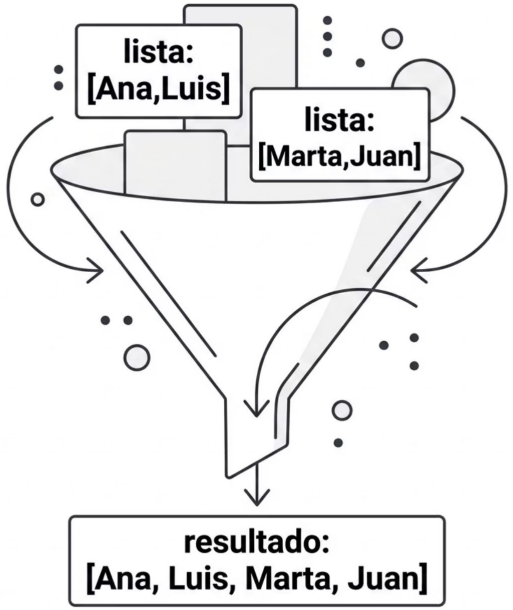
flatMap

¿Qué hace flatMap?

Transforma cada elemento **y aplana** el resultado en una sola colección.

```
val grupos = List(
  List("Ana", "Luis"),
  List("Marta", "Juan")
)
val estudiantes = grupos.flatMap(grupo => grupo)
// List("Ana", "Luis", "Marta", "Juan")
```

Es especialmente útil para trabajar con colecciones anidadas.



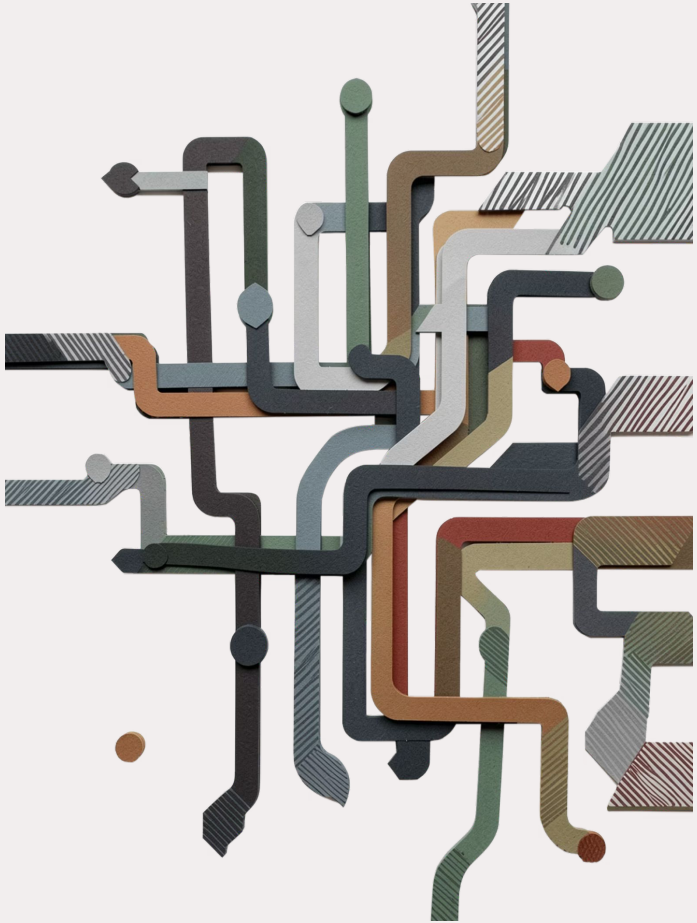
Con estos conceptos ya podemos construir un **ejemplo práctico completo** usando datos reales o simulados. ¡Continuamos en el siguiente bloque!

TUTORIAL PRÁCTICO

Ejemplo práctico y comparación con Java

Aplicando programación funcional con Scala

Ahora se utilizara **listas, case class, map, filter, maxBy** y **pattern matching** en un caso real.



CASO PRÁCTICO

Análisis de notas de estudiantes

Trabajaremos con una lista de estudiantes y sus calificaciones. El objetivo es aplicar las herramientas funcionales de Scala para responder cinco preguntas clave:

Identificar aprobados

¿Quiénes superaron la nota mínima?

Obtener nombres

Extraer solo el nombre de cada aprobado

Calcular promedio

Media aritmética del grupo completo

Mejor nota

¿Quién obtuvo la calificación más alta?

Clasificar desempeño

Etiquetar a cada estudiante según su nota

PASO 1

Modelar datos con `case class`

```
case class Estudiante(
  nombre: String,
  nota: Double
)

val estudiantes = List(
  Estudiante("Ana",   4.5),
  Estudiante("Luis",  2.8),
  Estudiante("Marta", 3.9),
  Estudiante("Juan",  5.0),
  Estudiante("Sofía", 2.5)
)
```

¿Por qué case class?

Una `case class` representa datos de forma **concisa, inmutable y expresiva**. Scala genera automáticamente:

- Constructor con parámetros nombrados
- Método `toString` legible
- Igualdad estructural por valor
- Soporte nativo para pattern matching

Sin boilerplate: no se necesitan getters, setters ni constructores manuales.

PASO 2

Filtrar estudiantes aprobados con `filter`

```
val aprobados =
  estudiantes
    .filter(_.nota >= 3.0)

// Resultado:
// List(
//   Estudiante("Ana",   4.5),
//   Estudiante("Marta", 3.9),
//   Estudiante("Juan",  5.0)
// )
```

¿Cómo funciona `filter`?

`filter` recorre cada elemento de la lista y conserva únicamente aquellos que cumplen la condición dada como función anónima.

La lista original **no se modifica**. Se crea una nueva lista.

Lista original

5 estudiantes (inmutable)

Lista resultado

Ana, Marta, Juan ✓

PASO 3

Transformar datos con `map`

```
val nombresAprobados = aprobados.map(_.nombre)
// Resultado: List("Ana", "Marta", "Juan")
```

`map` transforma cada elemento aplicando una función: aquí extrae el campo `nombre` de cada `Estudiante` aprobado. El resultado es una nueva lista del tipo transformado.

PASO 4

Promedio y mejor nota

```
// Promedio del grupo
val promedio =
  estudiantes
    .map(_.nota)
    .sum / estudiantes.length

// Resultado: 3.74

// Estudiante con mayor nota
val mejor =
  estudiantes.maxBy(_.nota)

// Resultado:
// Estudiante("Juan", 5.0)
```

Encadenando operaciones

1 `map(_.nota)`

Extrae sólo los valores numéricos de cada estudiante.

2 `sum`

Suma todos los valores de la nueva lista de notas.

3 `maxBy`

Devuelve el objeto `Estudiante` con la nota más alta.

PASO 5

Clasificación con `pattern matching`

```
def clasificar(nota: Double): String =
  nota match {
    case n if n >= 4.5 => "Excelente"
    case n if n >= 3.0 => "Aprobado"
    case _             => "Reprobado"
  }

val reporte = estudiantes
  .map(e => (e.nombre, clasificar(e.nota)))

// List(
//   ("Ana",   "Excelente"),
//   ("Luis",  "Reprobado"),
//   ("Marta", "Aprobado"),
//   ("Juan",  "Excelente"),
//   ("Sofía", "Reprobado")
// )
```

¿Por qué `match`?

`match` evalúa condiciones de forma **legible y exhaustiva**, eliminando cadenas de `if-else`. Cada caso (`case`) es una rama clara.

≥ 4.5
Excelente

≥ 3.0
Aprobado

resto
Reprobado

MINI RETO INTERACTIVO

¿Qué contiene `resultado`?

```
val resultado = estudiantes
  .filter(_.nota >= 3.0)
  .map(_.nombre)
```

⚠ ⏸️ Pausa — Piensa la respuesta antes de pasar a la siguiente línea.

> **Respuesta:** `List("Ana", "Marta", "Juan")`
>
> Con solo dos líneas encadenadas se filtran y transforman datos **sin modificar la lista original**. Esto es programación funcional aplicada.

PROGRAMACIÓN FUNCIONAL CON SCALA

Scala vs Java

- *Comparación*
- *Ventajas y Desventajas*

COMPARATIVA

Scala frente a Java

Java — Crear y transformar una lista

```
List<String> lista =
  new ArrayList<>();
lista.add("1");
lista.add("2");
lista.add("3");

List<Integer> enteros =
  lista.stream()
       .map(Integer::parseInt)
       .collect(
         Collectors.toList()
       );
```

Scala — Equivalente conciso

```
val lista = List("1","2","3")

val enteros =
  lista.map(_.toInt)
```

Mensaje clave

Java no debe entenderse como inutil o inferior, en las versiones modernas incluye herramientas funcionales, como streams y expresiones lambda, pero Scala integra este estilo de manera más natural en sus sintasis.

Mismo ecosistema. Diferente filosofía.

Ventajas y desventajas

✅ Ventajas

- Código conciso y expresivo
- Programación funcional de primera clase
- Tipado estático robusto
- Interoperabilidad total con Java
- Colecciones potentes e inmutables

⚠️ Desventajas

- Curva de aprendizaje pronunciada
- Abstracciones avanzadas complejas
- Difícil transición desde Java imperativo (clases tradicionales, valores mutables y ciclos)

Scala no solo **reduce código repetitivo**; propone una forma distinta de pensar: transformar datos mediante funciones, evitar mutaciones y construir programas más expresivos.

PROGRAMACIÓN FUNCIONAL CON SCALA

Conclusiones

- *Lo aprendido en el tutorial*
- *Instrucciones a transformaciones*
- *Recursos recomendados*

¿Qué aprendimos en este tutorial?

A lo largo de esta exposición, exploramos cómo Scala combina la programación funcional con la potencia de la JVM para escribir código más expresivo y robusto.

Inmutabilidad con `val`

Valores que no cambian, código más predecible.

Funciones puras y de orden superior

`se aplicaron operaciones como map`, `filter`, `reduce` y `flatMap`, para la transformación de datos

Case class y pattern matching

Modelado y clasificación de datos elegante.

Interoperabilidad con Java

Todo el ecosistema JVM a tu alcance (librerías y herramientas)

> "Scala no solo cambia la sintaxis: **cambia la forma de pensar el programa.**"

De instrucciones paso a paso a transformaciones de datos

Enfoque imperativo (término de pasos)

- → Modificar variables
- → Ciclos explícitos
- → Cambiar estado
- → Pensar en pasos
- → Código verboso

Enfoque funcional en Scala (transformación de datos)

- → Crear nuevos valores
- → Usar `map`, `filter`, `reduce`
- → Transformar datos
- → Pensar en funciones
- → Código declarativo

La programación funcional busca que el código sea más **predecible, modular y fácil de razonar**, reduciendo mutaciones innecesarias.

Recursos recomendados

Documentación y herramientas

Documentación oficial

`scala-lang.org/documentation`

Scastie (editor online)

`scastie.scala-lang.org`

Tutorial Scala para Java

Guía oficial de migración y comparación.

Material del curso

Página del profesor, notebook y repositorio del grupo.

Gracias

¿Preguntas?

https://wayground.com/join?gc=493656&source=liveDashboard

493656