

# Programación funcional con Scala

Presentado por:

Diego Felipe Solorzano

Laura Andrea Castiblanco

Mateo López Ruiz

Felipe Esteban Riaño





# ¿Quieres intentarlo tú también?

mybinder.org

binder

Have a repository full of Jupyter notebooks? With Binder, open those notebooks in an executable environment, making your code immediately reproducible by anyone, anywhere.

New to Binder? Get started with a [Zero-to-Binder tutorial](#) in Julia, Python, or R.

Build and launch a repository

GitHub repository name or URL

GitHub GitHub repository name or URL

Git ref (branch, tag, or commit)

Path to a notebook file (optional)

HEAD Path to a notebook file (optional) File search

Copy the URL below and share your Binder with others:

Fill in the fields to see a URL for sharing your Binder.

Expand to see the text below, paste it into your README to show a binder badge

How it works

Ir a <https://mybinder.org/>



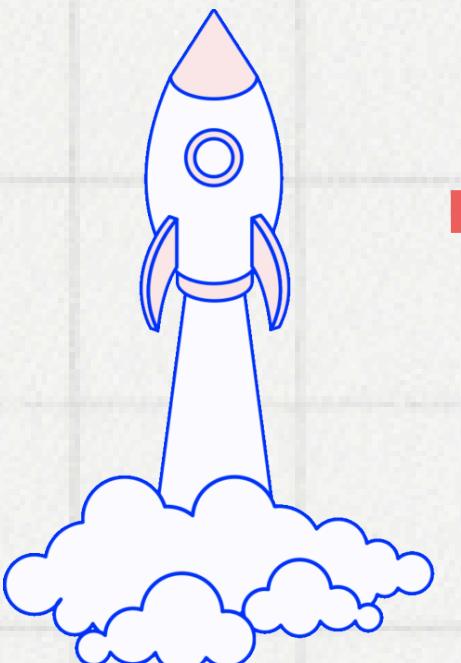
GitHub repository name or URL

GitHub

GitHub repository name or URL



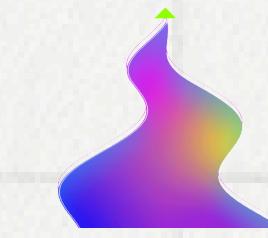
Coloca el siguiente repo:  
<https://github.com/felipe1297/PL2024>



launch



Oprime launch  
y ¡listo!



# Contenido

01. Primeros pasos

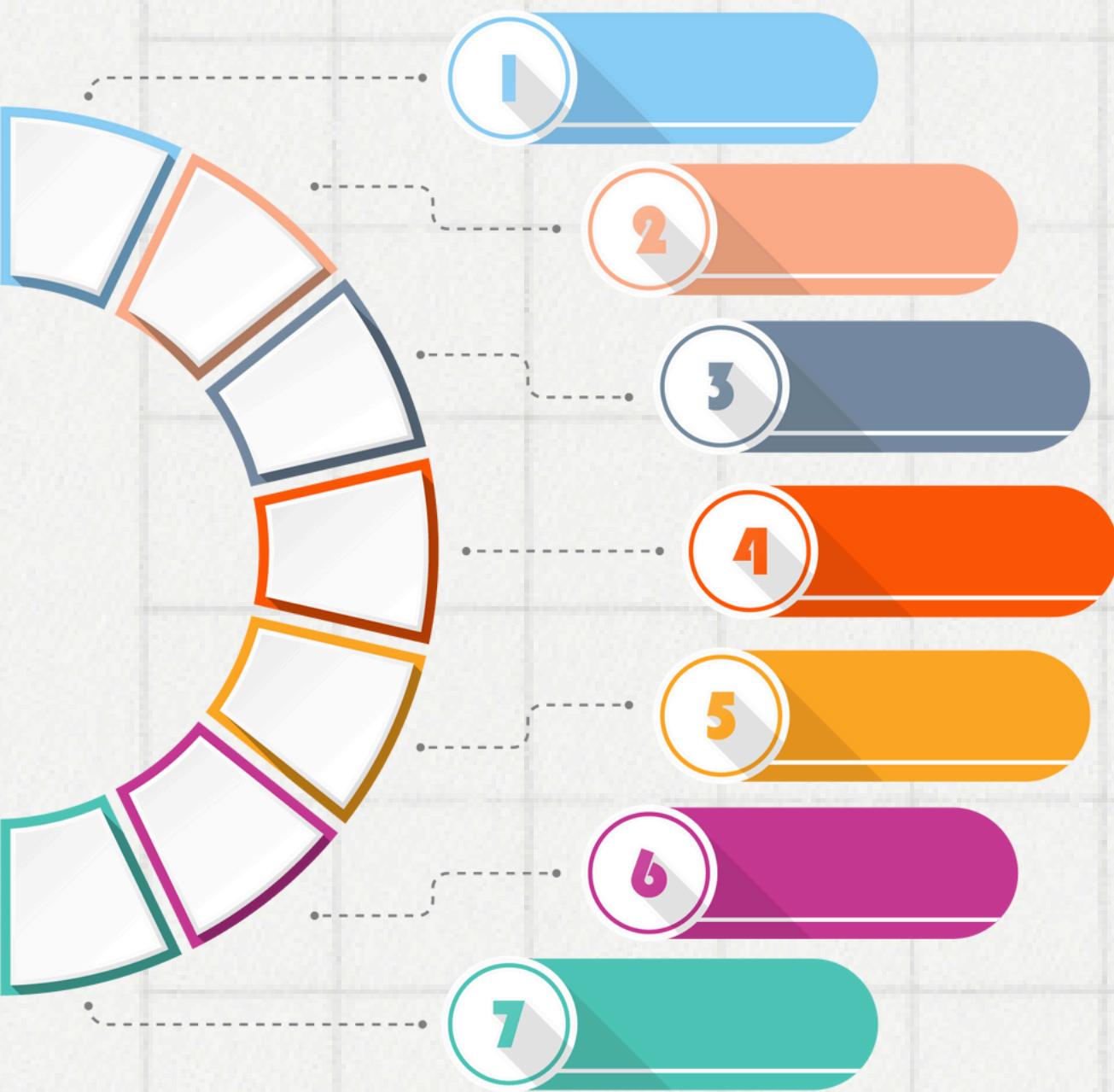
02. Tour por Scala

03. Programación  
funcional con Scala

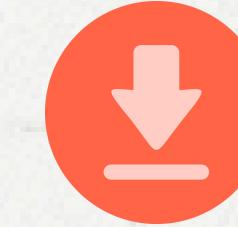
04. Ejemplo práctico

05. Conclusiones

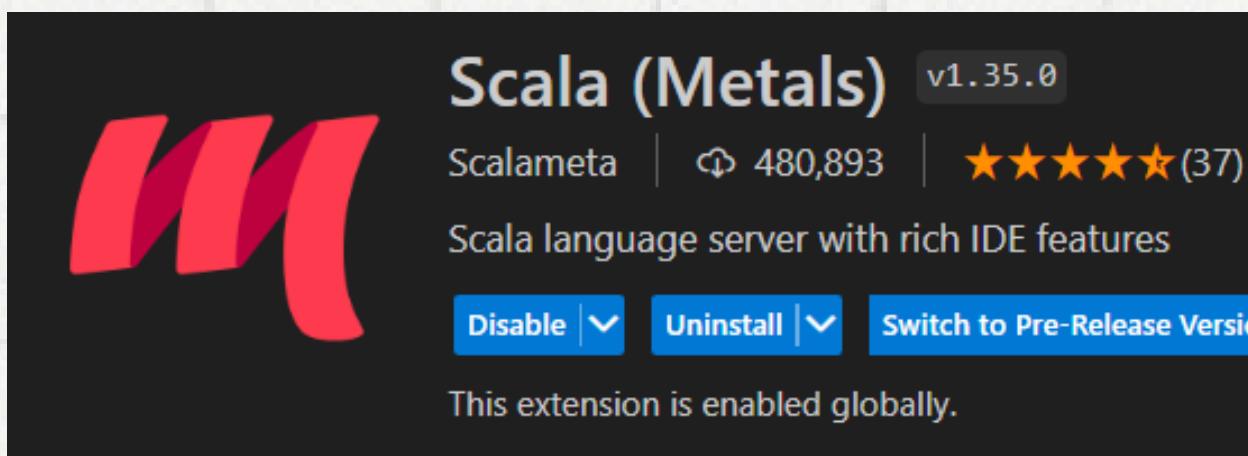
# PRIMEROS PASOS



# INSTALACIÓN DE SCALA



Ir a la página oficial de Scala <https://www.scala-lang.org/download/> y seguir las instrucciones para el sistema operativo que se necesita.



```
C:\Users\lalac\AppData\Local> 
[Progress bar]
Checking if a JVM is installed
Found a JVM installed under C:\Program Files\Java\jdk-17.

Checking if ~\AppData\Local\Coursier\data\bin is in PATH
Should we add ~\AppData\Local\Coursier\data\bin to your PATH? [Y/n] y

Checking if the standard Scala applications are installed
Installed ammonite
Installed cs
Installed coursier
Installed scala
Installed scalac
Installed scala-cli
Installed sbt
Installed sbt
Installed scalafmt

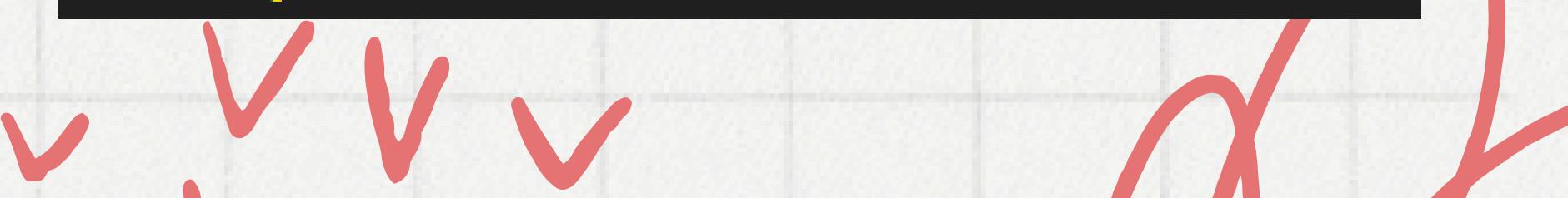
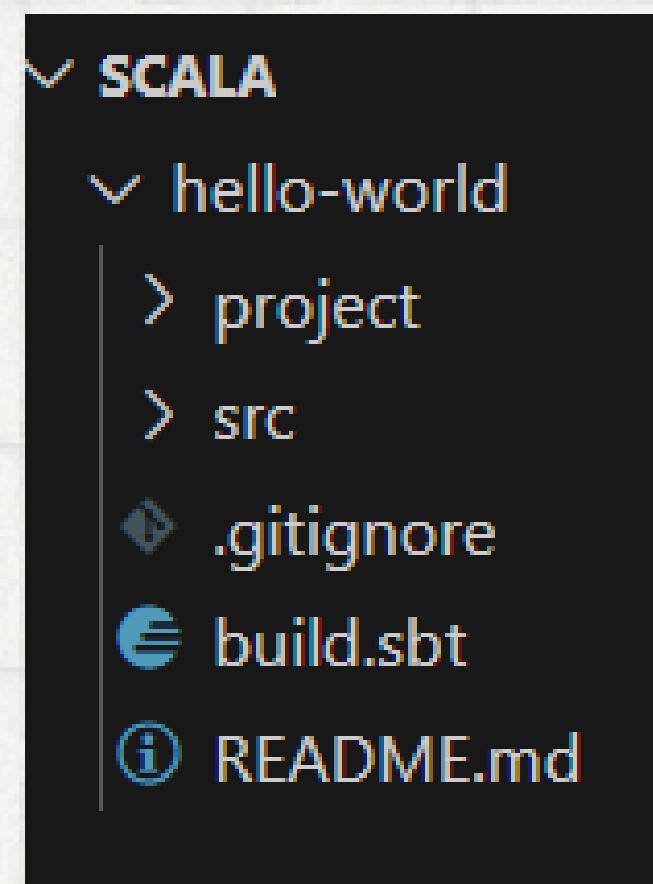
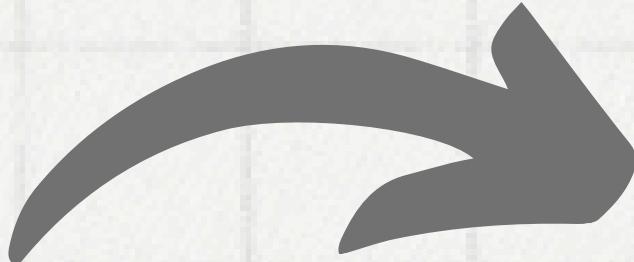
Press "ENTER" to continue...
```

# Hello World

```
PS C:\Users\lalac\Desktop\Scala> sbt new scala/scala3.g8
[info] [launcher] getting org.scala-sbt sbt 1.10.0 (this may take some time)...
[info] [launcher] getting Scala 2.12.19 (for sbt)...
[info] resolving Giter8 0.16.2...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
A template to demonstrate a minimal Scala 3 application

name [Scala 3 Project Template]: hello-world

Template applied in C:\Users\lalac\Desktop\Scala\.hello-world
```



```
Main.scala ×

hello-world > src > main > scala > Main.scala > ...
1 @main def hello(): Unit =
2   println("Hello world!")
3   println(msg)
4
5 def msg = "I was compiled by Scala 3. :)"
6
```

```
Main.scala ×

hello-world > src > main > scala > Main.scala > ...
1 object HolaMundo {
2   def main(args: Array[String]): Unit = {
3     println("¡Hola, mundo!")
4   }
5 }
```



**EXPRESIVO**

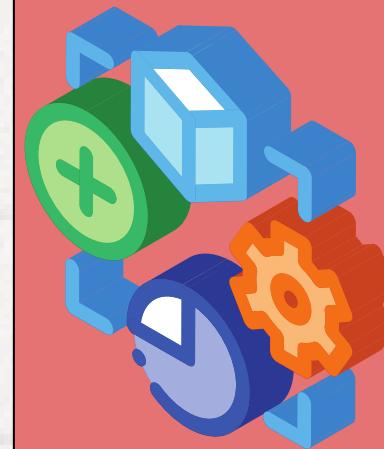


**ESCALABLE**



**SISTEMAS  
SEGUROS**

**¿QUÉ ES  
SCALA?**



**MULTI -  
PARADIGMA**

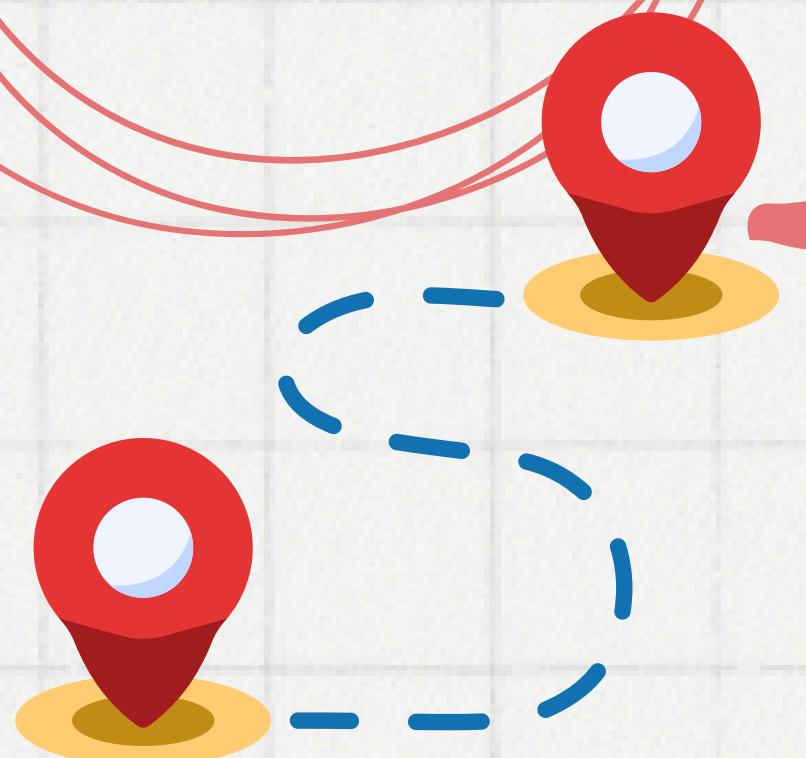


**EXTENSI-  
BILIDAD**



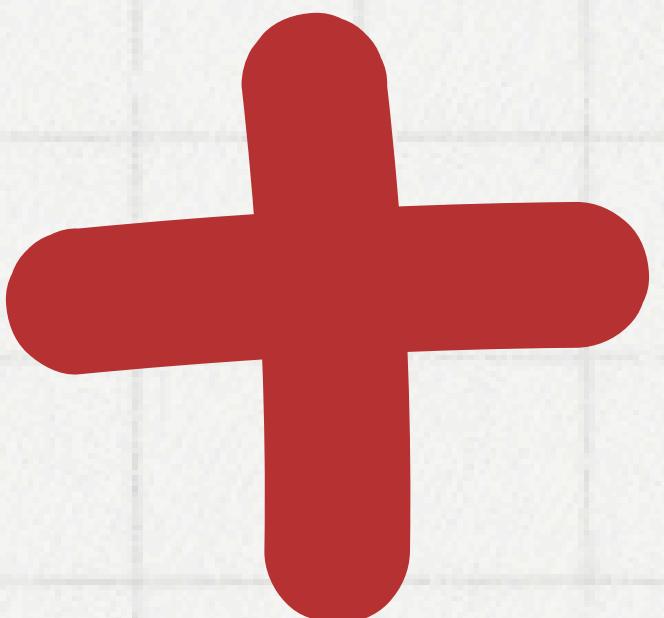
**CONCISO**

# TOUR POR SCALA



# LENGUAJE MULTIPARADIGMA

PROGRAMACIÓN  
ORIENTADA A  
OBJETOS



PROGRAMACIÓN  
FUNCIONAL



# Definición de variables

Cada tipo de variable es una clase

```
val num1: Int = 10 //val es inmutable  
var num2: Int = 20
```



# Tipos de datos

```
var bool: Boolean = true  
var char: Char = 'a'  
var string: String = "Hola"  
var float: Float = 1.0f  
var double: Double = 1.0  
var long: Long = 1L
```

También están las estructuras de datos características de Java

# If, for y while

## Estructura if

```
// Estructura if-else
val numero = 5
if (numero > 0) {
    println(s"$numero es positivo")
} else {
    println(s"$numero es negativo o cero")
}
```

## Estructura for

```
// Estructura for
for (i <- 1 to 5) {
    println(s"Iteración $i")
}
```

## Estructura while

```
// Estructura while
var contador = 3
while (contador > 0) {
    println(s"Contador: $contador")
    contador -= 1
}
```

# Case classes y Pattern matching

```
// Definición de una case class
case class Persona(nombre: String, edad: Int)

val persona1 = Persona("Alice", 25)
val persona2 = Persona("Bob", 30)
var persona3 = Persona("Charlie", 17)
```

Esta es una característica de Scala que destruye datos. Es similar a una estructura switch

Son clases inmutables y son muy útiles para la representación de datos y para la programación funcional

```
// Función que utiliza pattern matching
def describirPersona(persona: Persona): String = persona match {
    case Persona("Alice", 25) => "Esta es Alice y tiene 25 años."
    case Persona("Bob", 30) => "Este es Bob y tiene 30 años."
    case Persona(nombre, edad) if edad < 18 => s"$nombre es menor de edad."
    case Persona(nombre, edad) => s"$nombre tiene $edad años."
}
```

# Definición de funciones

```
// Definición de una función
def suma(a: Int, b: Int): Int = {
    a + b
}

// Llamada a la función
val resultado = suma(3, 4)
println(s"Resultado de la suma: $resultado")
```

Y funciones con argumentos por defecto que aceptan llamados con y sin argumento

Se tienen funciones con argumentos

```
// Función con parámetros por defecto
def saludo(nombre: String = "Mundo"): String = {
    s"Hola, $nombre!"
}

// Llamada a la función con y sin argumento
println(saludo("Scala"))
println(saludo())
```

$f(x)$ 

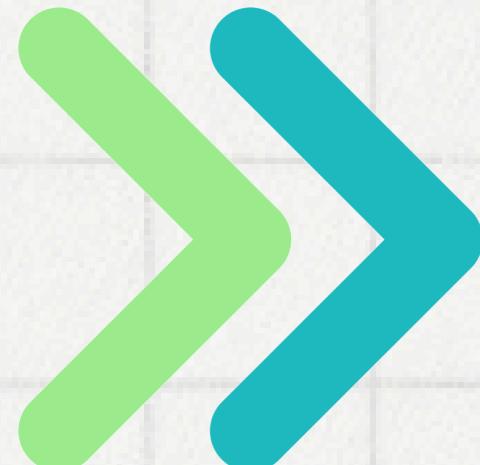
# PROGRAMACIÓN FUNCIONAL CON SCALA



# CAMBIO DE PARADIGMA



```
List<Integer> numeros =  
List.of(e1:18,e2:6,e3:4,e4:17,  
e5:4, e6:45, e7:90);  
  
int contador = 0;  
  
for (int numero : numeros){  
    if (numero > 10){  
        contador++;  
    }  
}
```



```
val numeros =  
List(18,6,4,17, 4, 45, 90)  
  
val contador =  
numeros.filter(_>10).length
```



01. INMUTABILIDAD

02. FUNCIONES  
PURAS

03. FUNCIONES DE  
PRIMERA CLASE

04. COMPOSICIÓN  
DE FUNCIONES

05. EVALUACIÓN  
PEREZOSA

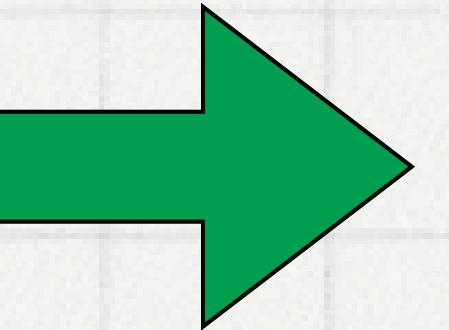
# PILARES DE LA PROGRAMACIÓN FUNCIONAL CON SCALA



# INMUTABILIDAD

## ¿QUÉ ES?

Los datos son inmutables, una vez creados no se pueden cambiar. Esto evita efectos secundarios no deseados

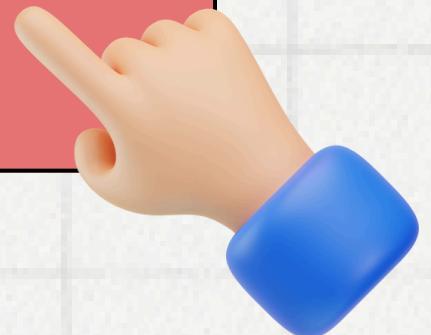


## ¿CÓMO SE GRANTIZA?

- Todas las variables se crean como campos **val**
- Solo se utilizan clases de colecciones inmutables, como **List**, **Vector** y las clases inmutables **Map** y **Set**.
- Las clases se crean como **case class**, cuyos parámetros de constructor son val de forma predeterminada

¿Pero si todo es  
inmutable, cómo  
puede cambiar  
algo?

Una colección  
existente no se  
modifica, sino que  
se le aplican  
funciones para crear  
una nueva



# EJEMPLOS INMUTABILIDAD

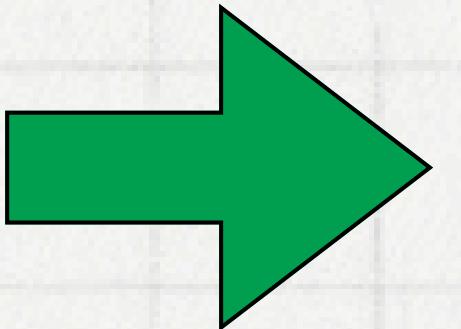
```
val a = List("jane", "jon", "mary")  
_____  
  
val b = a.filter(_.startsWith("j")).map(_.capitalize)  
_____
```

```
case class Person(firstName: String, lastName: String)  
_____  
  
val reginald = Person("Reginald", "Dwight")  
_____  
  
val elton = reginald.copy(  
  _____  
  firstName = "Elton",  
  _____  
  lastName = "John"  
  _____  
)
```

# FUNCIONES PURAS

## ¿QUÉ ES?

Una función pura es aquella que produce el mismo resultado para los mismos datos de entrada y no tienen efectos secundarios fuera de la función.



## EJEMPLOS

### FUNCIONES PURAS

- abs
- ceil
- max
- isEmpty
- length
- substring

### FUNCIONES IMPURAS

- println
- currentTimeMillis
- sys.error

Pero las funciones impuras también se necesitan...

Una aplicación se ve muy limitada si no puede interactuar con el mundo exterior



## RECOMENDACIÓN

Escriba el núcleo de su aplicación utilizando funciones puras y luego escriba un "envoltorio" impuro alrededor de ese núcleo para interactuar con el mundo exterior.

# EJEMPLOS DE FUNCIONES PURAS/RECUSIVIDAD

1

```
def double(i: Int): Int = i * 2
```

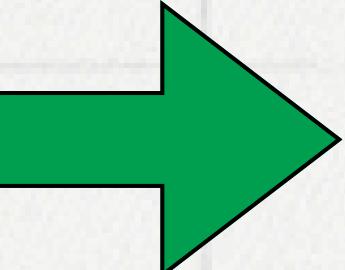
2

```
def sum(xs: List[Int]): Int = xs match
  case Nil => 0
  case head :: tail => head + sum(tail)
```

# FUNCIONES COMO CIUDADANOS DE PRIMERA CLASE

## ¿QUÉ SON?

Las funciones son tratadas como valores de datos comunes. Se pueden usar como argumentos o retornos válidos de otras funciones



## BENEFICIOS

- Puede definir métodos para aceptar parámetros de funciones
- Puede pasar funciones como parámetros a métodos.

# EJEMPLOS FUNCIONES DE PRIMERA CLASE

```
val nums = (1 to 10).toList
def double(i: Int): Int = i * 2
def underFive(i: Int): Boolean = i < 5
val doubles = nums.filter(underFive).map(double)
```

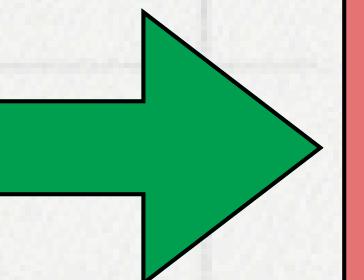
```
List("bob", "joe").map(_.toUpperCase)
List("bob", "joe").map(_.capitalize)
List("plum", "banana").map(_.length)
```

```
val nums = List(5, 1, 3, 11, 7)
nums.takeWhile(_ < 6).sortWith(_ < _) // List(1, 3, 5)
```

# COMPOSICIÓN DE FUNCIONES

## ¿QUÉ ES?

Combinación de funciones simples para crear funciones más complejas.



## BENEFICIOS

- Promueve la reutilización de código y la modularidad.
- Facilitan el trabajar con estructuras de datos y transformaciones de datos, como filtrado, mapeo y reducción.

# EJEMPLO DE COMPOSICIÓN DE FUNCIONES

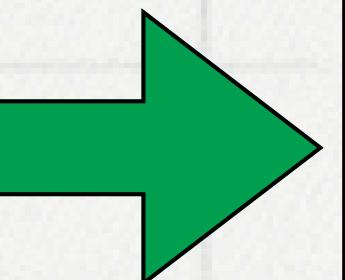
```
def double(x: Int): Int = x * 2
def intToString(x: Int): String = x.toString
val nums = (1 to 10).toList

val numsResult = nums.filter(_ > 5).map(double)
    .map(intToString).reduce(_ + ", " + _)
```

# FUNCIONES DE ORDEN SUPERIOR

## ¿QUÉ ES?

Son funciones que reciben como argumento funciones y/o devuelven funciones como resultado



## BENEFICIOS

Gracias a este tipo de funciones se puede **reutilizar código**, hacer **composición de funciones** y ofrece **flexibilidad** en el comportamiento de cualquier función

# EJEMPLO DE FUNCIONES DE ORDEN SUPERIOR

```
// Definición de una función de orden superior
def applyTwice(f: Int => Int, x: Int): Int = {
    f(f(x))
}

// Definición de una función simple que se utilizará como argumento
def addOne(n: Int): Int = n + 1

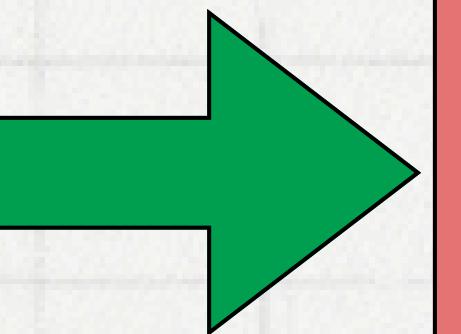
// Uso de la función de orden superior
val result = applyTwice(addOne, 5)

// Imprime el resultado
println(s"El resultado de aplicar la función dos veces es: $result")
```

# EVALUACIÓN PEREZOSA

## ¿QUÉ ES?

Es una estrategia de evaluación en la que la variable no se evalúa hasta que su valor sea necesario.



## BENEFICIOS

- **Eficiencia:** Se evitan cálculos innecesarios para las computaciones costosas.
- Capacidad de Manejar Estructuras de Datos Infinitas.
- **Modularidad:** Facilita la definición de funciones más modulares.

# EJEMPLO DE EVALUACIÓN PEREZOSA

```
// Definición de una función de orden superior
def applyTwice(f: Int => Int, x: Int): Int = {
    f(f(x))
}

// Definición de una función simple que se utilizará como argumento
def addOne(n: Int): Int = n + 1

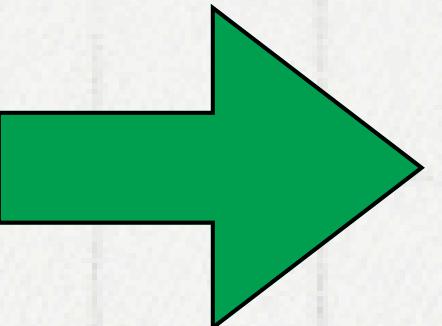
// Uso de la función de orden superior
val result = applyTwice(addOne, 5)

// Imprime el resultado
println(s"El resultado de aplicar la función dos veces es: $result")
```

# FUNCTIONAL ERROR HANDLING

## ¿QUÉ ES?

Estructuras de datos funcionales para manejar errores y excepciones de manera segura y expresiva.



```
def makeInt(s: String): Option[Int] = {  
    try {  
        Some(Integer.parseInt(s.trim))  
    } catch {  
        case e: Exception => None  
    }  
  
    val a = makeInt("1")      // Some(1)  
    val b = makeInt("one")   // None  
  
    makeInt("0") match {  
        case Some(i) => println(i)  
        case None => println("That didn't work.")  
    }  
}
```

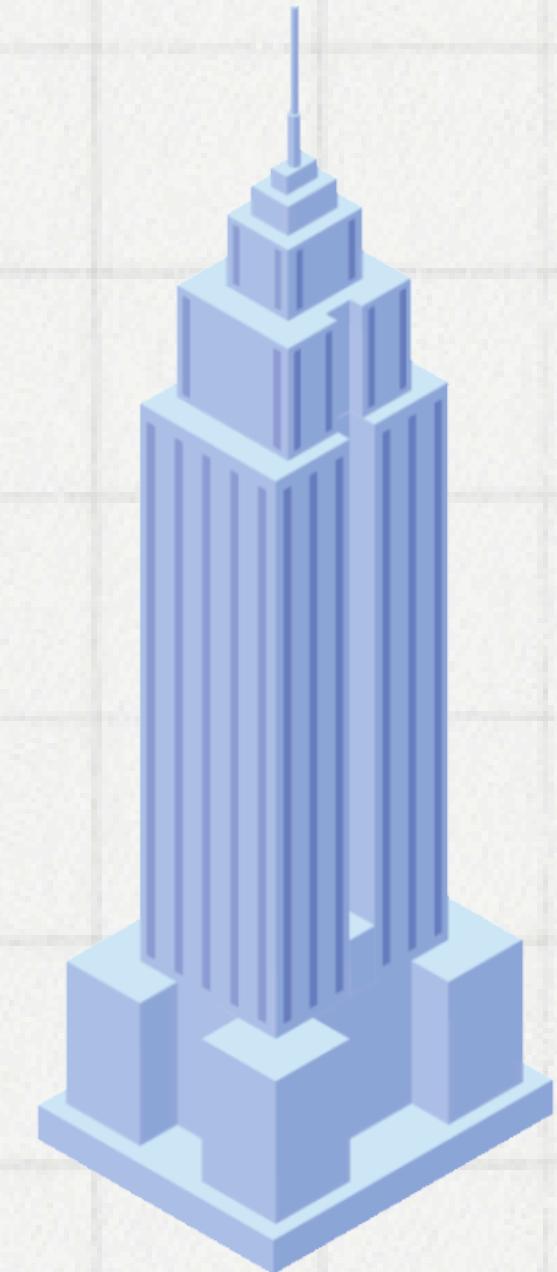
# Tipos de Orden Superior (HKT)

Los tipos de orden superior son tipos que toman otros tipos como parámetros. Permiten la definición de abstracciones sobre estructuras de datos

# Origen

Estructura proveniente de la teoría de categorías

- Functor: Mapear funciones
- Applicative: Aplicación de funciones encapsuladas
- Monad: Encadenar operaciones
- Traversable: Iterar y manipular estructuras
- Foldable: Reducción de estructuras



# Functors

Los functors son una abstracción que aplica una función a un valor envuelto en un contexto sin alterar el contexto en sí



Permiten mapear funciones sobre estructuras de datos de manera uniforme, facilitando la manipulación de datos dentro de contextos

## Función

## Abs

```
trait Functor[F[_]] {  
  def map[A, B](fa: F[A])(f: A => B): F[B]  
}
```

# Uso

```
val list: List[Option[Int]] = List(Some(1), None, Some(3))

def incrementOptions(list: List[Option[Int]]): List[Option[Int]] = {
    list.map {
        case Some(value) => Some(value + 1)
        case None => None
    }
}

val incrementedList = incrementOptions(list)
println(incrementedList)

val numbers: List[Int] = List(1, 2, 3, 4)

def incrementList(nums: List[Int]): List[Int] = {
    nums.map(_ + 1)
}

val incrementedNumbers = incrementList(numbers)
println(incrementedNumbers)
```

```
trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A => B): F[B]
}

implicit val listFunctor: Functor[List] = new Functor[List] {
    def map[A, B](fa: List[A])(f: A => B): List[B] = fa.map(f)
}

implicit val optionFunctor: Functor[Option] = new Functor[Option] {
    def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa.map(f)
}

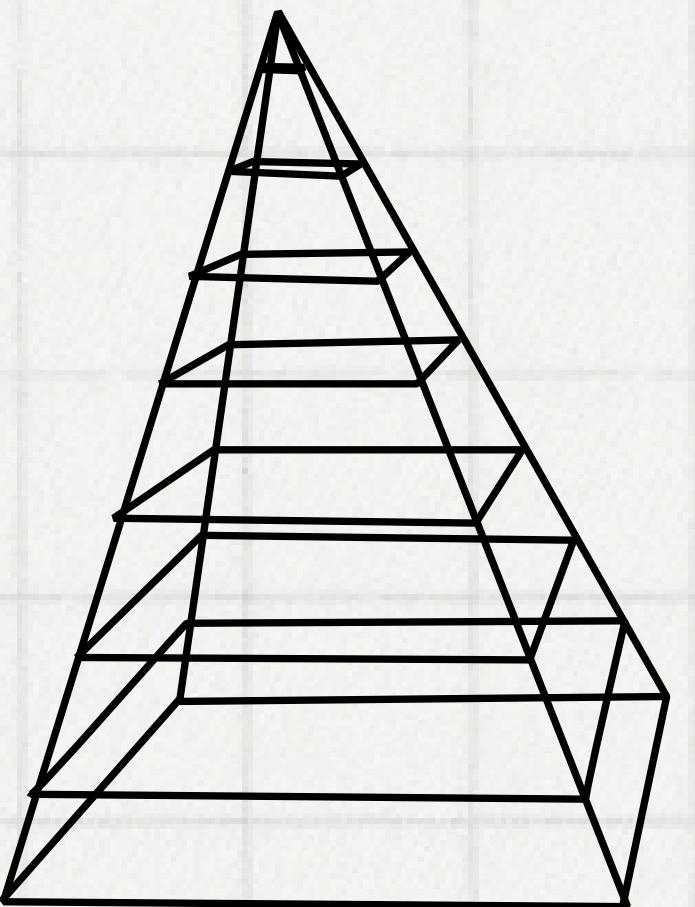
def increment[F[_]: Functor](container: F[Int]): F[Int] = {
    implicitly[Functor[F]].map(container)(_ + 1)
}

val list: List[Option[Int]] = List(Some(1), None, Some(3))
val incrementedList = list.map(increment(_))
println(incrementedList)

val numbers: List[Int] = List(1, 2, 3, 4)
val incrementedNumbers = increment(numbers)
println(incrementedNumbers)
```



# Monads



Estructura que representa cálculos como una secuencia de pasos. Proporciona una forma de encadenar operaciones mientras maneja contextos como fallos, efectos secundarios y asíncronía

Monads ?



Simplificación en el manejo de efectos secundarios y permite componer operaciones de manera limpia

**Función**

**Abs**

```
trait Monad[M[_]] {  
    def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B]  
    def unit[A](a: => A): M[A]  
}
```

# Uso

```
case class Superhero(name: String, base: String, powerLevel: Int)
case class Villain(name: String, powerLevel: Int)

def getSuperhero(id: Int): Option[Superhero] = {
  if (id == 1) Some(Superhero("Superman", "Metropolis", 100))
  else if (id == 2) Some(Superhero("Batman", "Gotham", 85))
  else None
}

def getBase(superhero: Superhero): Option[String] = {
  Some(superhero.base)
}

def calculatePowerLevel(superhero: Superhero): Option[Int] = {
  if (superhero.powerLevel > 0) Some(superhero.powerLevel)
  else None
}

def getArchEnemy(superhero: Superhero): Option[Villain] = {
  superhero.name match {
    case "Superman" => Some(Villain("Lex Luthor", 90))
    case "Batman" => Some(Villain("Joker", 70))
    case _ => None
  }
}

def calculateFightRisk(heroPower: Int, villainPower: Int): Option[String] = {
  if (heroPower > villainPower) Some("Low Risk")
  else if (heroPower == villainPower) Some("Medium Risk")
  else Some("High Risk")
}
```

```
def getSuperheroDetails(id: Int): Option[(String, Int, Villain, String)] = {
  val heroOpt = getSuperhero(id)
  if (heroOpt.isEmpty) return None

  val baseOpt = getBase(heroOpt.get)
  if (baseOpt.isEmpty) return None

  val powerLevelOpt = calculatePowerLevel(heroOpt.get)
  if (powerLevelOpt.isEmpty) return None

  val enemyOpt = getArchEnemy(heroOpt.get)
  if (enemyOpt.isEmpty) return None

  val riskOpt = calculateFightRisk(powerLevelOpt.get, enemyOpt.get.powerLevel)
  if (riskOpt.isEmpty) return None

  Some((baseOpt.get, powerLevelOpt.get, enemyOpt.get, riskOpt.get))
}

println("Without Monads")
val details = getSuperheroDetails(1)
println(details)
```

```
def getSuperheroDetails(id: Int): Option[(String, Int, Villain, String)] = {
  for {
    hero <- getSuperhero(id)
    base <- getBase(hero)
    powerLevel <- calculatePowerLevel(hero)
    enemy <- getArchEnemy(hero)
    risk <- calculateFightRisk(powerLevel, enemy.powerLevel)
  } yield (base, powerLevel, enemy, risk)
}

print("With Monads")
val details = getSuperheroDetails(1)
println(details)
```

# Patrón Typeclass

Abstracción en la programación funcional que permite definir comportamiento genérico que puede ser implementado por diferentes tipos

# Propósitos

En Scala, las clases de tipos se representan mediante traits con métodos abstractos

Proveer una forma de extender funcionalidad a tipos existentes sin modificar su definición

Facilitar la separación de la lógica de comportamiento de la lógica de datos

Permitir el polimorfismo ad-hoc, es decir, definir funciones que pueden operar sobre diferentes tipos sin conocerlos de antemano

# Uso

```
def showInt(x: Int): String = x.toString
def showString(x: String): String = x

println(showInt(123))
println(showString("hello"))
```



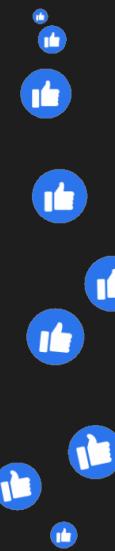
```
trait Show[A] {
  def show(a: A): String
}

implicit val intShow: Show[Int] = new Show[Int] {
  def show(a: Int): String = a.toString
}

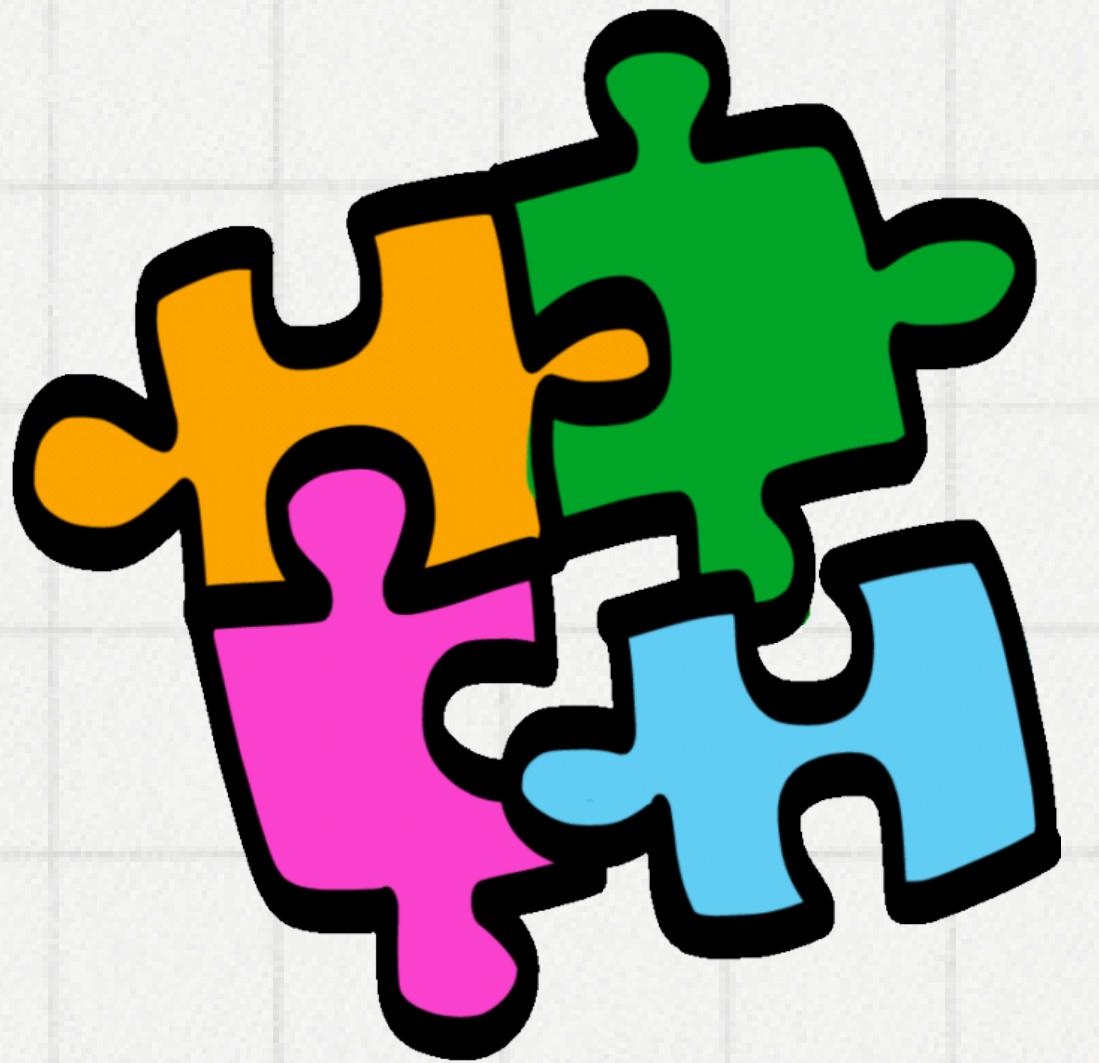
implicit val stringShow: Show[String] = new Show[String] {
  def show(a: String): String = a
}

def printShow[A](a: A)(implicit s: Show[A]): Unit = {
  println(s.show(a))
}

printShow(123)
printShow("hello")
```



# Tipos de datos algebraicos (ADT)



Los ADTs son tipos de datos que se forman combinando otros tipos mediante productos (tuplas) y sumas (uniones disjuntas)

# ~~¿Por y para qué?~~

Modelado datos  
Composición tipos  
Manejo de escenarios  
Estructura

Seguridad en tipado de datos

Patrón de Coincidencia

Inmutabilidad

Claridad y Mantenibilidad

## Beneficios

# Uso



```
sealed trait PaymentMethod
case class CreditCard(number: String, name: String, expiry: String) extends PaymentMethod
case class PayPal(email: String) extends PaymentMethod

def processPayment(payment: PaymentMethod): String = payment match {
  case CreditCard(number, name, expiry) =>
    s"Processing credit card payment for $name: $number $expiry"
  case PayPal(email) =>
    s"Processing PayPal payment for $email"
}

val creditCardPayment = CreditCard("1234-5678-9876-5432", "Bruce Wayne", "12/23")
val paypalPayment = PayPal("tony.stark@starkindustries.com")
// val unknownPayment = Bitcoin("1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa") // Error

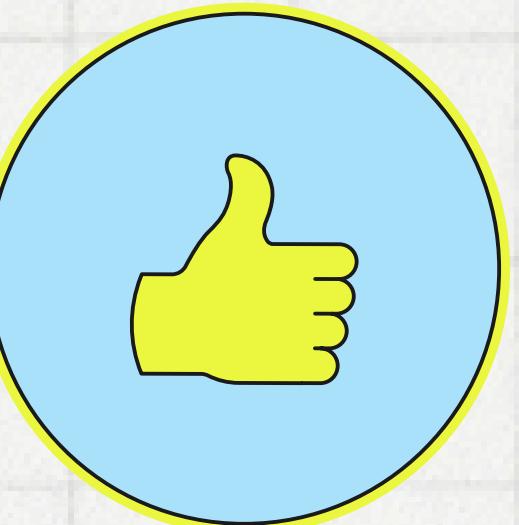
println(processPayment(creditCardPayment))
println(processPayment(paypalPayment))
// println(processPayment(unknownPayment)) // Error
```

```
def processPayment(payment: (String, String)): String = {
  val (paymentType, details) = payment

  if (paymentType == "CreditCard") {
    s"Processing credit card payment: $details"
  } else if (paymentType == "PayPal") {
    s"Processing PayPal payment: $details"
  } else {
    throw new IllegalArgumentException("Unknown payment type")
  }
}

val creditCardPayment = ("CreditCard", "1234-5678-9876-5432 Bruce Wayne 12/23")
val paypalPayment = ("PayPal", "tony.stark@starkindustries.com")
val unknownPayment = ("Bitcoin", "1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa")

println(processPayment(creditCardPayment))
println(processPayment(paypalPayment))
println(processPayment(unknownPayment))
```





# EJEMPLO PRÁCTICO

# Conclusiones

- Permite combinar los paradigmas funcional y orientado a objetos, proporcionando una buena flexibilidad para el programador.
- La inmutabilidad que provee Scala ayuda a evitar errores que serían comunes en la programación concurrente.
- Scala permite una mayor abstracción y composición de funciones, permitiendo el desarrollo de un código más modular y reutilizable.
- Las funciones de orden superior o de primera clase simplifican la construcción de programas complejos a partir de componentes más simples



# Recursos

- <https://docs.scala-lang.org/es/tutorials/scala-for-java-programmers.html>
- <https://www.youtube.com/watch?v=i9o70PMqMGY>
- <https://geekytheory.com/curso-scala-parte-4-variables-y-tipos-de-datos/>

**Muchas  
gracias**