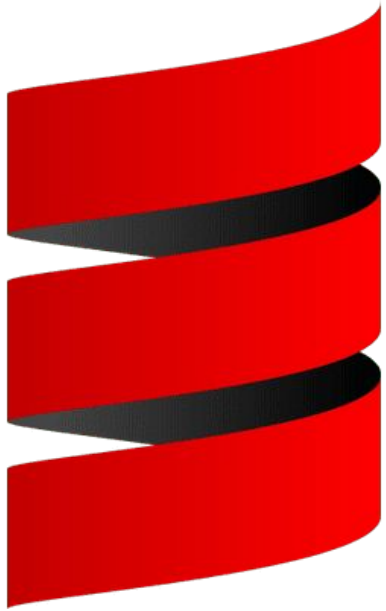
Scala

KEEPCODING
Tech School

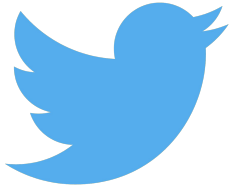
Antonio Gutiérrez López
gutierrez.lopez.ant@gmail.com

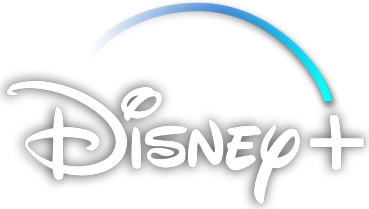
¿Qué es Scala?

- Liberado en 2003
- Desarrollo en EPFL liderado por Martin Odersky
- Lenguaje de propósito general
- Multi-paradigma:
  - Programación Orientada a Objetos
  - Programación Funcional
- Ejecuta sobre la JVM
- Fuertemente tipado



¿Quién usa Scala?





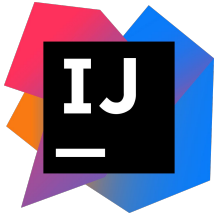
Proyectos más conocidos

Herramientas

- **REPL** (Read-Eval-Print-Loop): https://docs.scala-lang.org/overviews/repl/overview.html

- **SBT** (Simple Build Tool): https://www.scala-sbt.org/

- **Intellij IDEA**: https://www.jetbrains.com/idea/



Indice

- Basic Foundations
  - val/var, methods/functions, class, trait, objects…
  - logical expressions, if statements & loops
- Pattern Matching
- Standard Library
  - seq, list, option
  - map, filter, fold, flatmap, flatten
  - for comprehension
  - try & either types
- More Features
  - Implicits, generics, variance
- Futures

Basic Fundations

KEEPCODING
Tech School

val *vs* var

- **Type Inference**

- **val**: Immutable values

```
val nameWontChange = "foo"
nameWontChange = "bar" // compilation error:
reassignment to val

val namesMayChange: Array[String] = new Array(1)
namesMayChange(0) = "foo" // It's works

val namesWontChange = Seq("foo")
namesWontChange(0) = "foo" // error: value update
is not a member of Seq[String]
```

- **var**: Mutable variables

```
var numMayChange: Int = 3
numMayChange = 5
println(5) // 5, it's works
```

- **lazy vals**: evaluated when it’s call

```
lazy val itWillResolved = 10 // itWillResolved = <lazy>
println(itWillResolved) // itWillResolved = 10
```


Methods / Functions

- **Type Inference**
- **Last line as return**

```
def greet(name: String): Unit = println(s"Hello $name")
def greet(name: String): String = s"Hello $name"
def sum(n: Int, m: Int): Int = n + m
```

- **Nested Methods:**

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

- Variable args number

```
def greet(names: String*) =
    println(s"Hello ${names.mkString(", ")}")
def sum(n: Int*): Int = n.reduce((n, m) => n+m)
```

- Methods vs Functions

```
def m(x: Int) = 2 * x
val f = (x: Int) => 2 * x
```

Class

```
class User(val  name: String, val  lastName: String, var active: Boolean = false) {
  val fullName: String = s"$name $lastName"
  val role: String = {
    if (name == "Carlos") "admin" else "guess"
  }
  println("Init block expression") println(s"Role is: $role")
  def greet():  Unit = println(s"Hello i'm $fullName")
}

val user = new User("John", "Doe")
println(user.fullName)
user.active = true
println(user.greet())
```

Abstract Class

```
abstract class Pet(val name: String, val age: Int){
  val greeting: String
  protected val ageMultiplier: Int = 1

  def greet(): Unit = {
    println(greeting)
  }

  def getHumanAge(): Int = ageMultiplier * age
  def sleep(): Unit
}
```

```
class Dog(name: String, age: Int) extends Pet(name, age){
  val greeting = "I'm a Dog"
  override protected val ageMultiplier = 7

  override def sleep(): Unit = {
    println("Sleeping")
  }
}

val dog = new Dog("Rex", 5)
dog.greet()
dog.sleep()
println(dog.getHumanAge())
```

Trait

```scala
trait Bird {
  val name: String
  override def toString(): String = s"My name is $name"
}

trait Flying {
  def fly(): Unit = println("Flying...")
  override def toString(): String = s"I can fly"
}

trait Swimmer {
  def swim(): Unit = println("Swimming...")
  override def toString(): String = s"I can swim"
}
```

```scala
class Duck extends Bird with Flying with Swimmer {
  override val name = "Duck"
  override def toString():  String =
    s"${super[Bird].toString} and
      ${super[Flying].toString} and
      ${super[Swimmer].toString}"
}

class Pigeon extends Bird with Flying {
  override val name = "Pigeon"
}

class Penguin extends Bird with Swimmer {
  override val name = "Penguin"
}
```

Object

- Singletons
- Lazy construction

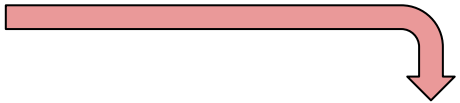
```
object Maths {
  val Pi: Double = 3.1416
  def sum(nums: Long*): Long = nums.reduce((n, m) => n + m)
  def max(nums: Long*): Long = nums.reduce((n, m) => if (n else m)
}

Maths.sum(5, 5) // 10
Maths.Pi // 3.1416


object App {
  def main(args: Array[String]): Unit = {
    println("Running...")
  }
}
```

Case Class

```scala
case class User(name: String, age: Int)
val user = User("John", 30)
```



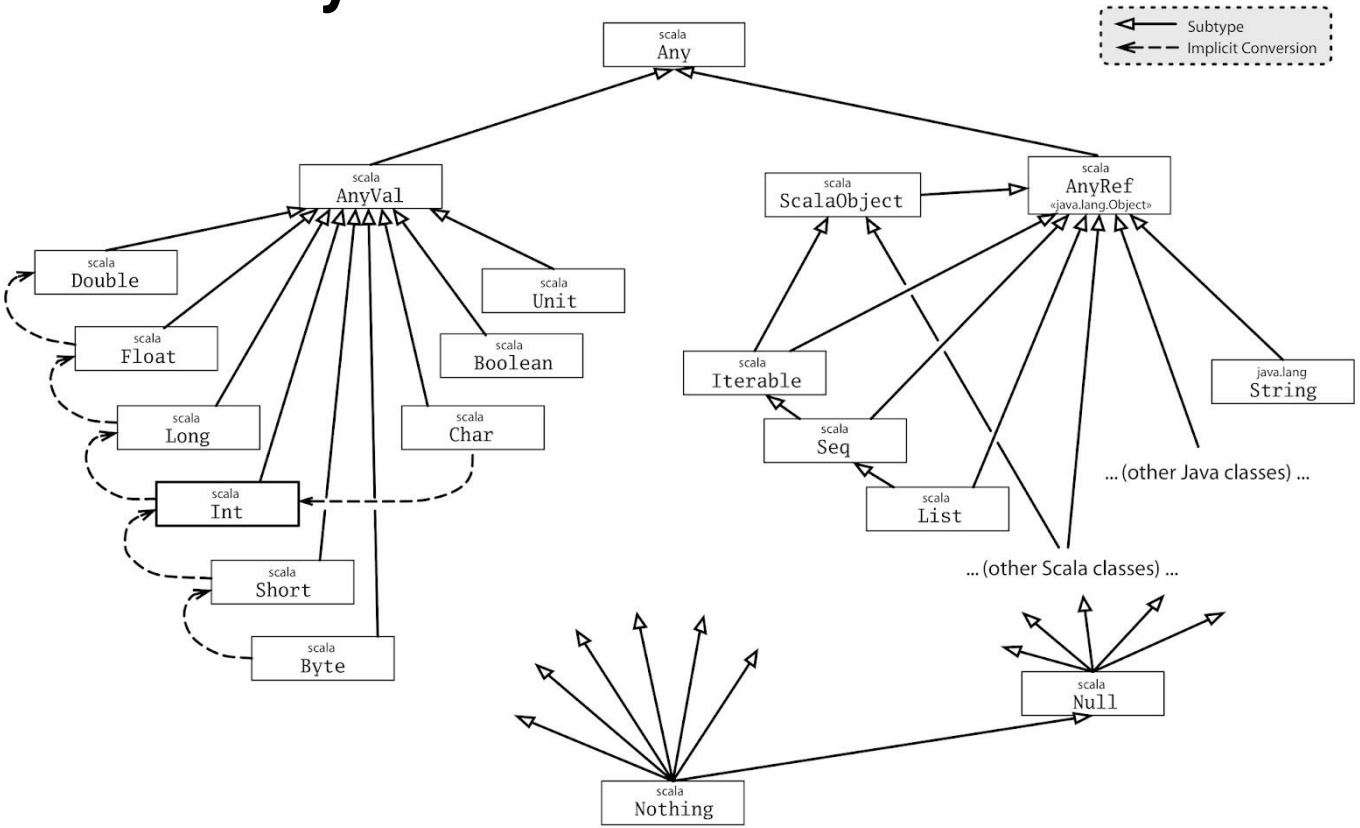
- Parameter accessors
- Serializable
- Copy method
- Apply and unapply
- NO inheritable

```scala
class User(val name: String, val age: Int) extends Serializable
{
  def copy(name: String = this.name, age: Int = this.age) =
    new User(name, age)

  override def toString():  String = s"User($name, $age)"
}

object User {
  def apply(name: String, age: Int): User = new User(name, age)
  def unapply(user: User): Option[(String, Int)] =
    Some((user.name,  user.age))
}
```


Type Hierarchy



Exercise 1

OOP Modeling

High Order Functions

```
object Calculator{
  def sum(n1: Int, n2: Int): Int = n1 + n2
  def subtract(n1: Int, n2: Int): Int = n1 - n2
}


def calculate(n1: Int, n2: Int)(operation:(Int, Int) => Int) = operation(n1, n2)
type Operation = (Int, Int) => Int

def calculate(n1: Int, n2: Int)(operation: Operation) = operation(n1, n2)
```


High Order Functions II

```
def calculate(n1: Int, n2: Int)(operation: Operation) = operation(n1, n2)

//Remember the calculator
def sum(n1: Int, n2: Int) = n1 + n2
calculate(1, 2)(Calculator.sum) //returns 3

calculate(1, 2)((n1: Int, n2: Int) => (n1 + n2)*2) //returns 6
```


Logical Expressions

Constants	`true false`
Negation	`!a`
Conjunction	`a && b`
Disjunction	`a || b`
Comparison	`a <= b, a >=b, a < b, a > b, a == b, a != b`

```
!true => false
!false => true
```

```
true && a => a
false && a => false
```

```
true || a => true
false || a => a
```


If, then, else

```
if ( condition ) ... else ...

if ( x > 0 ) println("x mayor que 0") else println("x menor/igual 0")

val value = if (x >= 0) x else 0
def abs(x: Double) = if (x >= 0) x else -x
```


Loops: for, foreach, while

```
for(w <- range){ // Code.. }

for( w <- 0 to 10)
{
  println(w);
}

for( w <- 0 until 10)
{
  println(w);
}
```

```
for( w <- List(1,2,3))
{
  println(w);
}

List(1,2,3).foreach(x => println(x))
List(1,2,3).foreach(println(_))
List(1,2,3).foreach(println)
```

Loops: for, foreach, while

```
while(condition){ statement(s); }

var a = 10

// while loop execution
while( a < 20 ) {
  println( "Value of a: " + a )
  a = a + 1
}
```


Pattern Matching

KEEPCODING
Tech School

Basic Pattern Matching

```
def defcon(level: Int): String = {
  level match {
    case 1 => "Nuclear war is imminent"
    case 2 => "Next step to nuclear war"
    case 3 => "Increase in force readiness above that required for normal readiness"
    case 4 => "Increased intelligence watch and  strengthened security measures"
    case 5 => "Lowest state of readiness"
    case _ =>  throw new Exception("Invalid defcon value")
  }
}
```

```
def canSwim(bird: Bird): Boolean =
  bird match {
    case _: Swimmer => true
    case _ =>  false
  }
```

```
def isFullEquip(bird: Bird): Boolean =
  bird match {
    case b if b.isInstanceOf[Swimmer] &&  b.isInstanceOf[Flying] =>  true
    case _ =>  false
  }
```


Pattern Matching Extractors

```
case class User(name: String, email: String)
def isGmailUser(user: User): Boolean = user match {
  case User(_, email) if email.endsWith("@gmail.com") => true
  case _ => false
}

def parseArg(arg : String, value: Any) = (arg, value) match {
  case ("-l", lang: String) => setLanguageTo(lang)
  case ("-i", iterations: Int) => setIterationsTo(iterations)
  case ("-h" | "--help", _) => displayHelp()
  case unk => throw new Exception(s"Unsupported argument ${unk._1}${unk._2}")
}
```

Pattern Matching Extractors II

```
sealed trait Expression
case object X extends Expression
case class Const(value: Int) extends Expression
case class Add(left: Expression, right: Expression) extends Expression
case class Mult(left: Expression, right: Expression) extends Expression

def eval(expression : Expression, xValue: Int): Int = expression match {
  case X => xValue
  case Const(value) => value
  case Add(left, right) => eval(left, xValue) + eval(right, xValue)
  case Mult(left, right) => eval(left, xValue) * eval(right, xValue)
}

val expr = Add(Const(1), Mult(Const(2), Mult(X, X)))
eval(expr, 2) // 9
```


Pattern Matching with extractors & @

```
case class User(name: String, email: String, active: Boolean)

user match {
  case user @ User(_, _, true) => save(user)
  case _ => _
}
```


Exercise 2

Pattern Matching

Standard Library

KEEPCODING
Tech School

Seq's, List's, Set's & Option's

```
val mySeq = Seq(1, 2, 3, 4, 5)  //As a Java List (interface)

val myList = List(1, 2, 3, 4, 5) //As a Java Linked List (implementation)

val mySet = Set(1, 2, 3, 4, 5)

val myOption = Option(1)
```


filter, map & fold

```
List(1, 2, 3, 4, 5).filter(_ > 2) //List(3, 4, 5)

List(1, 2, 3, 4, 5).map(_ + 1) //List(2, 3, 4, 5, 6)

List(1, 2, 3).fold(0)(_ + _) //6
```

map, flatMap & flatten

```
List(1, 2, 3).map(value => List(value, value + 1))
//List(List(1, 2), List(2, 3), List(3, 4))

List(1, 2, 3).map(value => List(value, value + 1)).flatten
//List(1, 2, 2, 3, 3, 4)

List(1, 2, 3).flatMap(value => List(value, value + 1))
//List(1, 2, 2, 3, 3, 4)
```


Options

```
Option(1) match {
  case Some(value) => s"I have the value $value"
  case None => "I don't have value"
}

val myValue: Int = Option(1).getOrElse(0)

val myOption: Option[Int] = Option(1).orElse(0)
```


Exercise 3

Standard Library

For Comprehension

```
for {
  x <- List(1, 2, 3)
  y <- List(true, false)
} yield (x, y)

//List((1, true), (1, false), (2, true), (2, false), (3, true), (3, false))


List(1, 2, 3).flatMap(x =>
  List(true, false).map( y => (x, y))
)
```


For Comprehension II

```
for {
  x <- List(1, 2, 3, 4, 5, 6, 7)
  if x < 3
  y <- List("a", "b")
} yield (x, y)

//result: List((1,a), (1,b), (2,a), (2,b))


List(1, 2, 3, 4, 5, 6, 7).withFilter(_ < 3).flatMap( x =>
  List("a", "b").map(
    y => (x, y)
  )
)
```


Exercise 4

For Comprehension

Try Type

```
Try(myFunction) match {
  case Success(result) => println("All right!")
  case Failure(exception) => println("Ouch! " + exception.getMessage)
}

Try(myFunction).getOrElse(defaultValue)
```

Either Type

```
def getHead(l: List[Int]): Either[Exception, Int] =
  if (l.isEmpty)
    Left(new Exception("empty collection"))
  else
    Right(l.head)


getHead(myList) match {
  case Right(head) => println(s"head: $head")
  case Left(exception) => println(exception.getMessage)
}
```


More features

KEEPCODING
Tech School

Implicit Parameters

```
def printMessage(text: String)(implicit prefix: String) =
  println(s"$prefix $text)


implicit val pre = "Sam says: "

printMessage("Hello World!")
// Sam says: Hello World!
```


Implicit conversion

```
case class Point(x: Int, y: Int)

implicit def tupleToPoint(tuple: (Int, Int)) =
    Point(tuple._1, tuple._2)

def sumPoints(p1: Point, p2: Point) =
    Point(p1.x + p2.x, p1.y + p2.y)

val p1: Point = Point(1, 2)
val p2: (Int, Int) = (3, 4)
sumPoints(p1, p2)
```


Implicit Class

```
class ExternalClass {
  private val list: List[String] = List("1","2")
  def getSize(): Int = list.size
}
val instance = new ExternalClass()

if (instance.getSize == 0) ....

def isEmpty(ex: ExternalClass): Boolean = instance.getSize == 0
if (isEmpty(ex)) ....
```


Implicit Class II

```
class ExternalClass {
  private val list: List[String] = List(.....)
  def getSize(): Int = list.size
}

implicit class EnrichExternalClass(ex: ExternalClass){
  def isEmpty: Boolean =ex.getSize == 0
}

if (ex.isEmpty) ....
```


Exercise 5

Implicit Class

Generics

```
class MyAwesomeCollection[T](initCollection: Seq[T]) {

  val myList: Seq[T] = initCollection

  def contains(element: T): Boolean = myList.contains(element)

  def addNewEelement(element: T): Seq[T] = myList :+ element

  def hasMoreElementsThan[S](otherCollection: Seq[S]): Boolean =
    myList.size > otherCollection.size
}
```


Futures

KEEPCODING
Tech School

Future[T]

- By default, non-blocking operations
- They will hold a T value at some point
- So a future may be uncompleted (it has no value yet) or completed
- Completion will be treated as a scala.util.Try value. It will have two possible values:
  - Success(t: T)
  - Failure(t: Throwable)

Future[T] II

```
import scala.concurrent._
import ExecutionContext.Implicits.global

val firstPrimeNumbers: Future[List[Int]] = Future {
  List(1, 2, 3, 4, 7, 11, 13)
  // what if 'calculateFirstPrimeNumbers(100000000000)'
}

val thisWillFail: Future[Int] = Future(2 / 0)
```

Execution context

- A future, once it’s completed, it never change of value
- An ExecutionContext
  - execute tasks submitted to them
  - They can be seen as thread pools
  - Most of future ops require an implicit ExecutionContext

Future: Catch Results

- Expecting results:
  - Blocker way (discouraged but sometimes mandatory)
  - Non-blocker way: using callbacks

Future: Blocking - Await

```
import scala.concurrent._
import scala.concurrent.duration._
import ExecutionContext.Implicits.global

val f: Future[Int] = Future {
  Thread.sleep(10000)
  2
}

println(Await.result(f, 12 seconds))
```


Future: Blocking - Await

```
import scala.concurrent._
import scala.concurrent.duration._
import ExecutionContext.Implicits.global

val f: Future[Int] = Future {
  Thread.sleep(10000)
  2
}

println(Await.result(f, 5 seconds)) =>
                java.util.concurrent.TimeoutException
```


Future: Callbacks

```
import scala.concurrent._
import ExecutionContext.Implicits.global

val f: Future[Int] = Future {
  Thread.sleep(10000)
  2
}

f.onComplete(n => println(n))
```


Future: Callbacks II

- onComplete

```
f.onComplete( (t: Try[Int]) => println(t) )

import scala.util._
f.onComplete {
  case Success(t) => println(t)
  case Failure(e) => pritnln(e.getMessage)
}
```

- onSuccess/foreach

```
f.onSuccess{case n => println(n) }
f.foreach(n => println(n) }
```

- onFailure

```
f.onFailure{ case throwable => println(throwable.getMessage) }
f.failed.foreach(e => println(e.getMessage))
```


Future: map & flatMap

```
def getFirstMillionOfPrimes(): Future[List[Int]] = ???

getFirstMillionOfPrimes().map(
  (list: List[Int]) => list.head
)  //Future[Int]

def concatenate(l: List[Int]): Future[String] = ???

getFirstMillionOfPrimes().flatMap(
  (list: List[Int]) => concatenate(list))
)  //Future[String]
```


Future: recover & recoverWith

```
val f: Future[Int] = Future {
  1 / 0
}.recover {
  case e: ArithmeticException => 0
}


val f2: Future[Int] = Future {
  1 / 0
}.recoverWith {
  case e: ArithmeticException => Future(0)
}
```


Exercise 6

Futures

Futures: For Comprehension

```
getFirstMillionOfPrimes().flatMap(
  (list: List[Int]) => concatenate(list))
)  //Future[String]
```

is equal to:

```
for {
  primes <- getFirstMillionOfPrimes()
  primeString <- concatenate(primes)
} yield primeString
```


Exercise 7

Future with For Comprehension

KEEPCODING

Tech School

Madrid | Barcelona | Bogotá

Datos de contacto