

Lenguajes de programación 2022-1

Programación Funcional con Scala

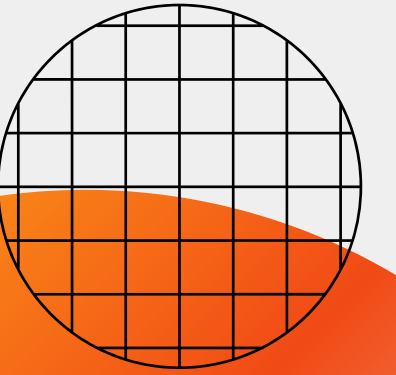
Integrantes:

Francisco Sebastian Dueñas Caicedo

Juan Camilo Gomez Lopez

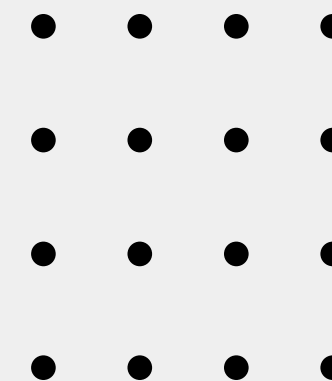
Juan Diego Moreno Mora

Oscar Julian Tinjaca Reyes



Contenido

Primeros pasos
Tour por Scala
Particularidades del lenguaje
Programación Funcional





Primeros pasos

Que es Scala?

Expresa patrones de programación comunes de una forma **concisa**, **elegante**, y con **tipado seguro**

Multiparadigma:

- **Orientación a objetos**
- **Programación Funcional**

Historia



Historia



JVM

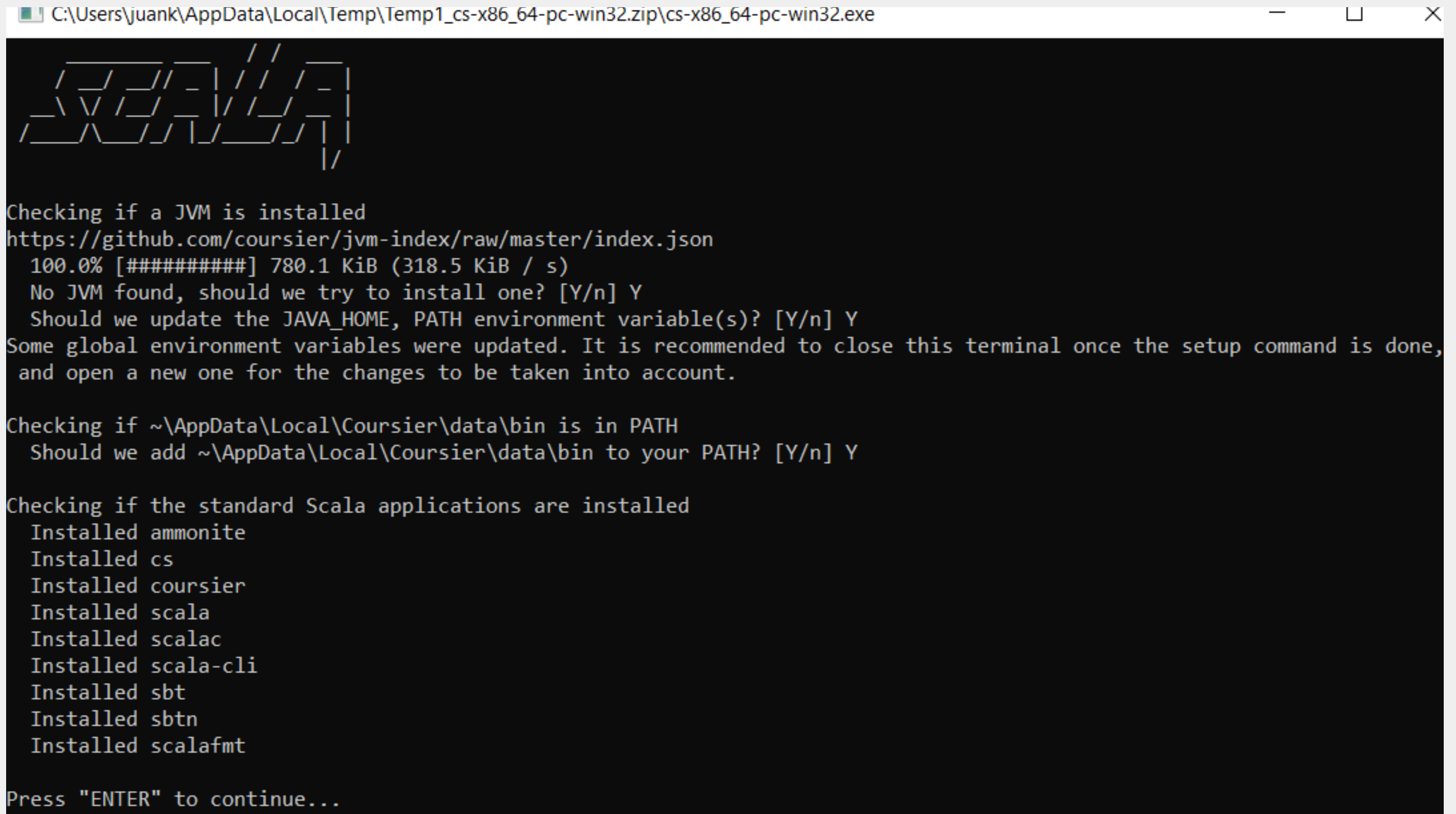
- La plataforma principal de Scala es Java Virtual Machine (JVM)
- A veces, las nuevas versiones de JVM y JDK requieren que actualicemos Scala para que siga siendo compatible
- JDK 8, 11 y 17 son opciones razonables tanto para compilar como para ejecutar código Scala.
- Para mas información dirigirse a <https://docs.scala-lang.org/overviews/jdk-compatibility/overview.html>



Instalación

- Para sistema operativo windows descargar el instalador basado en coursier, que puede ser encontrado en https://github.com/coursier/launchers/raw/master/cs-x86_64-pc-win32.zip o <https://www.scala-lang.org/download/>
- Ejecutar el .exe que se encuentra en el .zip y seguir las instrucciones del programa, el cual se encargara de instalar JVM, Agregar y actualizar variables de entorno, e instalar Scala localmente.

A large, stylized orange circle is positioned on the right side of the page, partially cut off by the edge. The circle has a gradient, transitioning from a lighter orange at the top to a darker orange at the bottom. The background is a solid light gray.

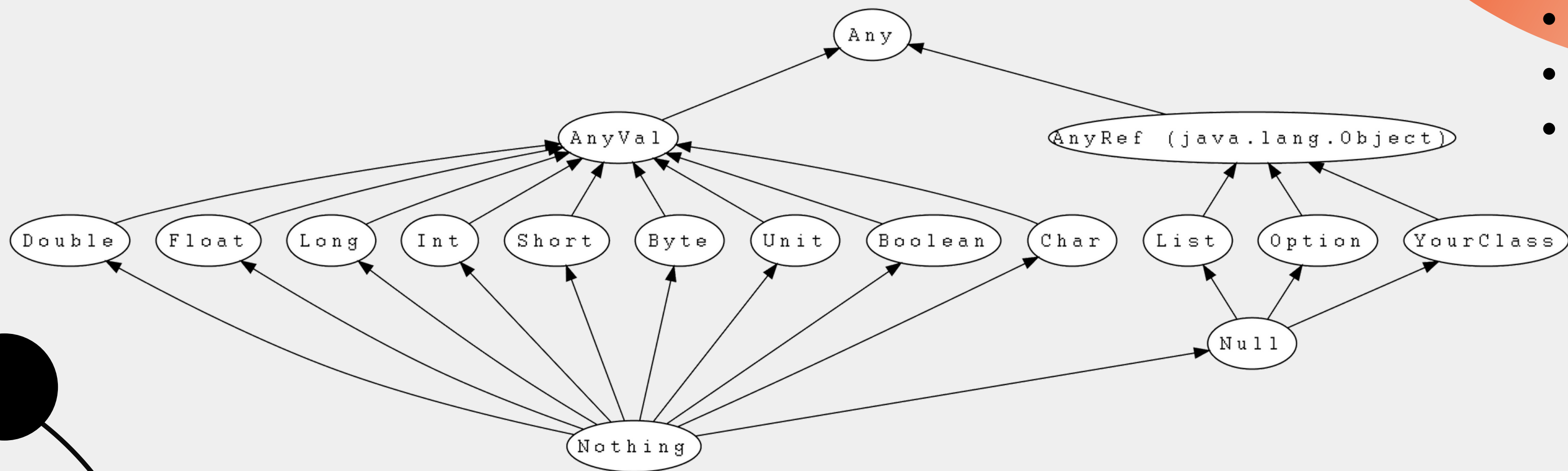


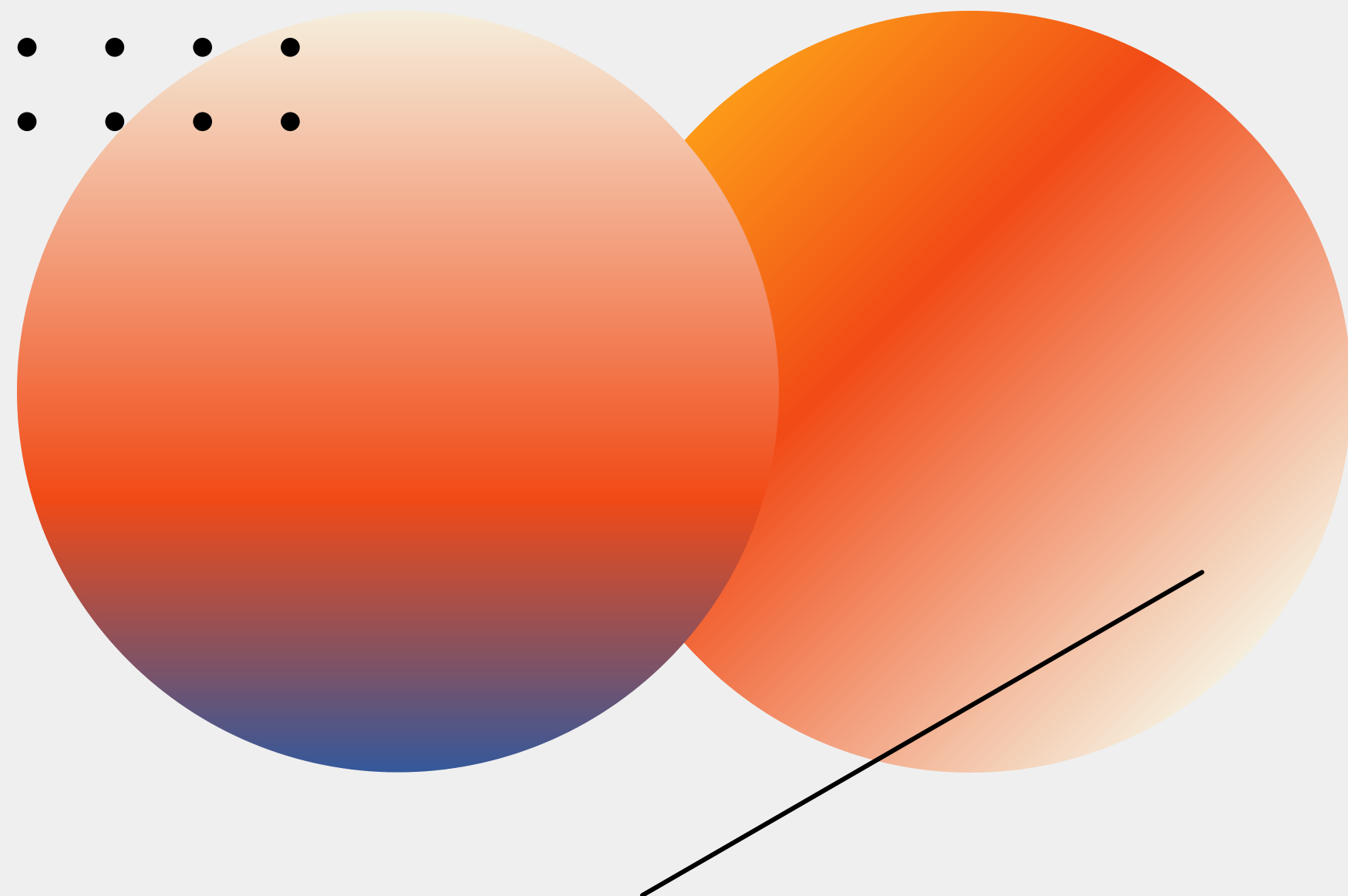
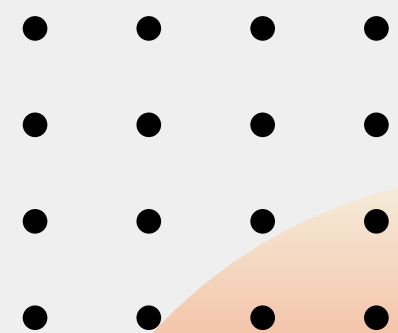


Tour por Scala

Tipos de datos

Subconjunto de la jerarquía de los tipos de datos





Expresiones

Valores y variables

Valores

Hacen referencia a un valor, el cual NO puede ser modificado.

```
val x = 1 + 1  
println(x) // 2  
x = 3 // Esto no compilará
```

```
val x: Int = 1 + 1
```

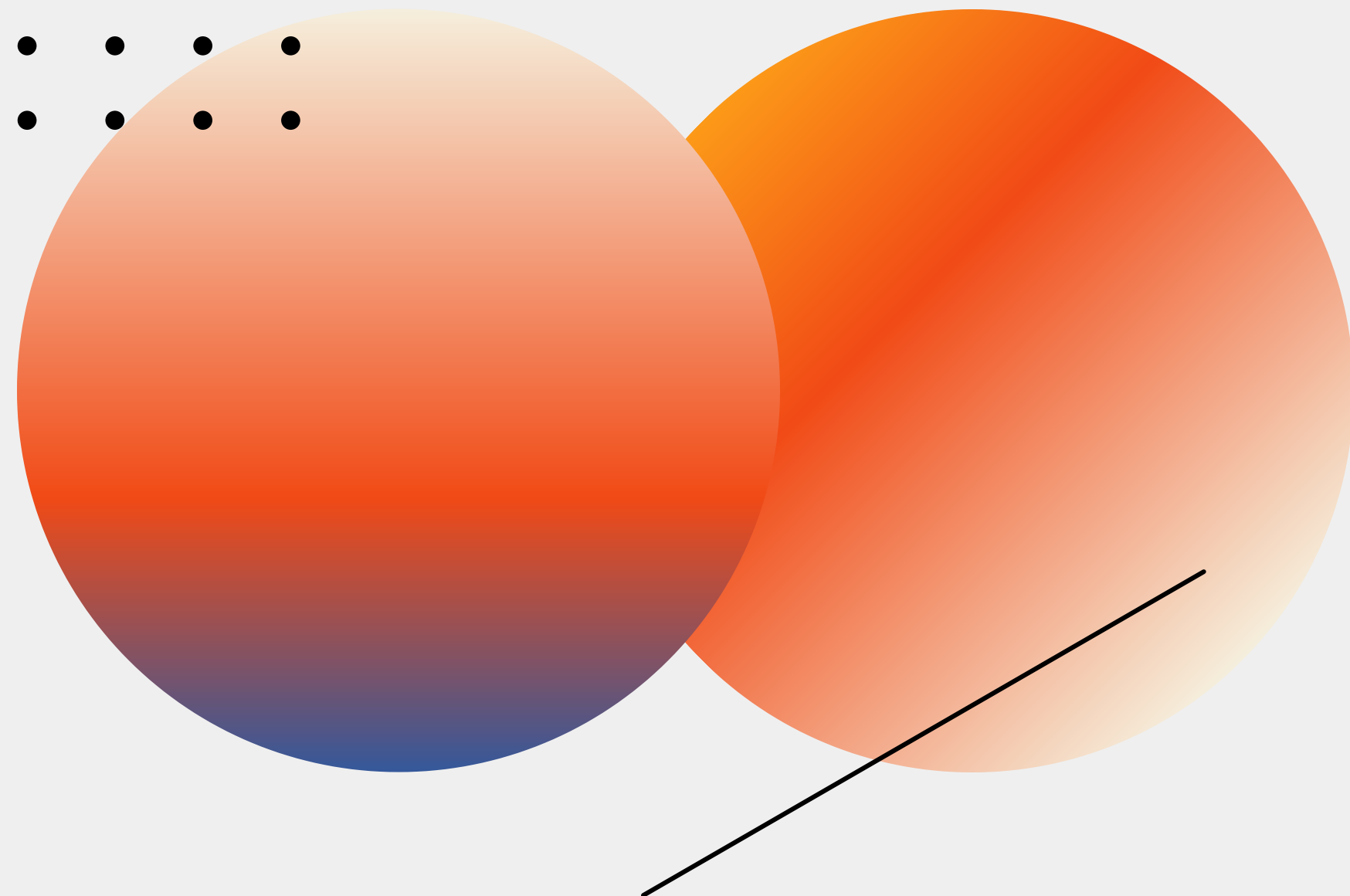
Variables

Hacen referencia a un valor, el cual puede ser modificado.

```
var x = 1 + 1  
x = 3 // Compila, dado que "x"  
      está declarado como "var"  
println(x * x) // 9
```

```
var x: Int = 1 + 1
```





Funciones y métodos

Funciones y métodos

Son expresiones que tienen parámetros y toman argumentos, pero existen algunas diferencias clave entre ellos.

`(x: Int) => x + 1 // Func. anónima`

`val addOne = (x: Int) => x + 1
println(addOne(1)) // 2`

`val add = (x: Int, y: Int) => x + y
println(add(1, 2)) // 3`

`def add(x: Int, y: Int): Int = x + y
println(add(1, 2)) // 3`

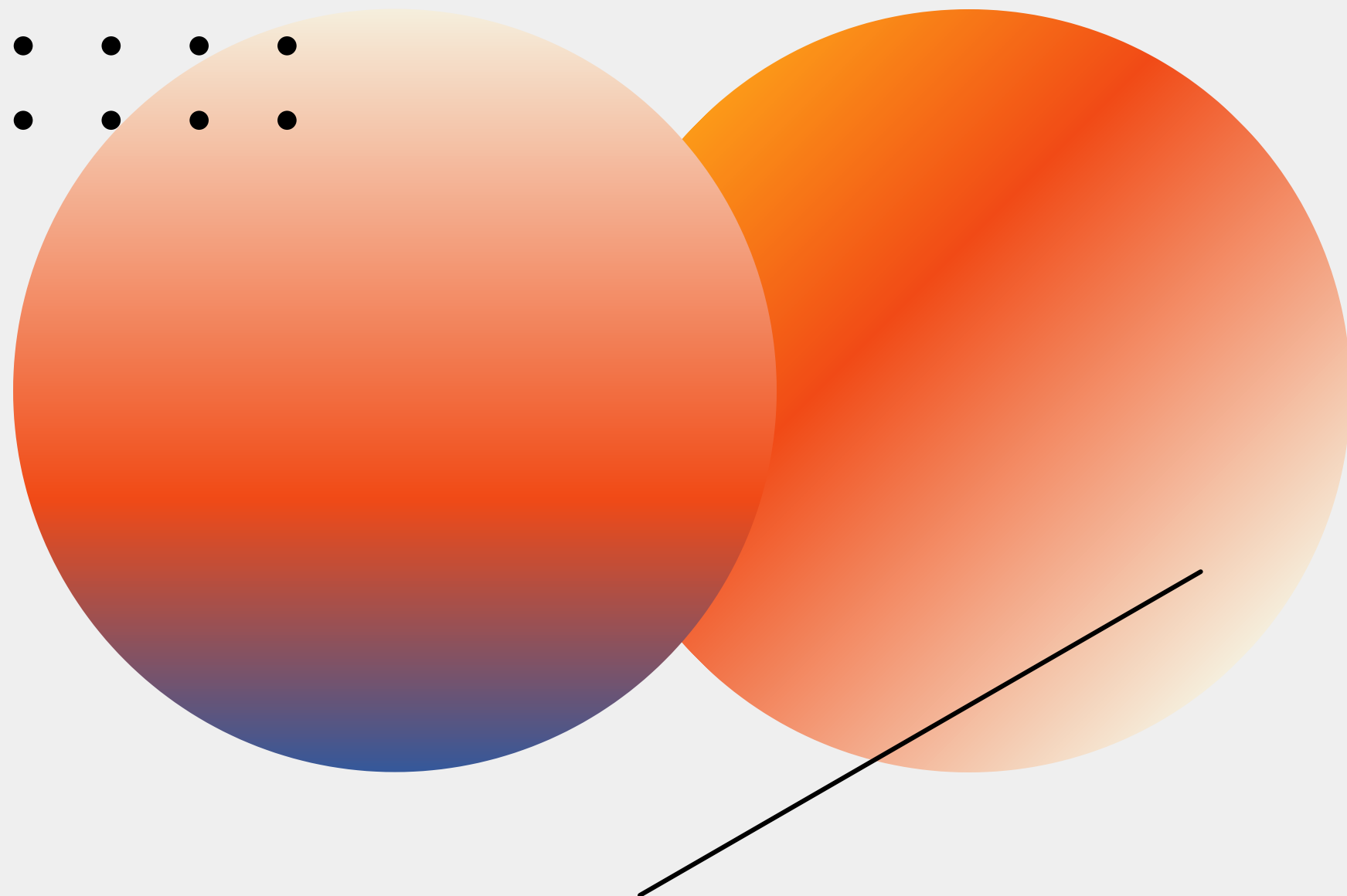
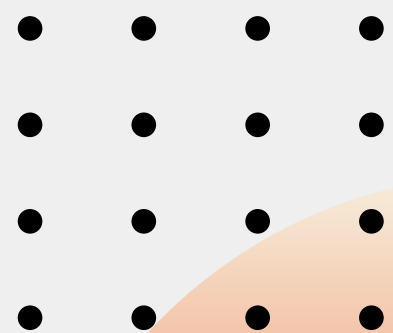
`def addThenMultiply(x: Int, y: Int)
(multiplier: Int): Int = (x + y) * multiplier
println(addThenMultiply(1, 2)(3)) // 9`

Funciones y métodos

```
val getTheAnswer = () => 42  
println(getTheAnswer()) // 42
```

```
def name: String = System.getProperty("user.name")  
println("Hello, " + name + "!")
```

```
def getSquareString(input: Double): String = {  
    val square = input * input  
    square.toString  
}  
println(getSquareString(2.5)) // 6.25
```

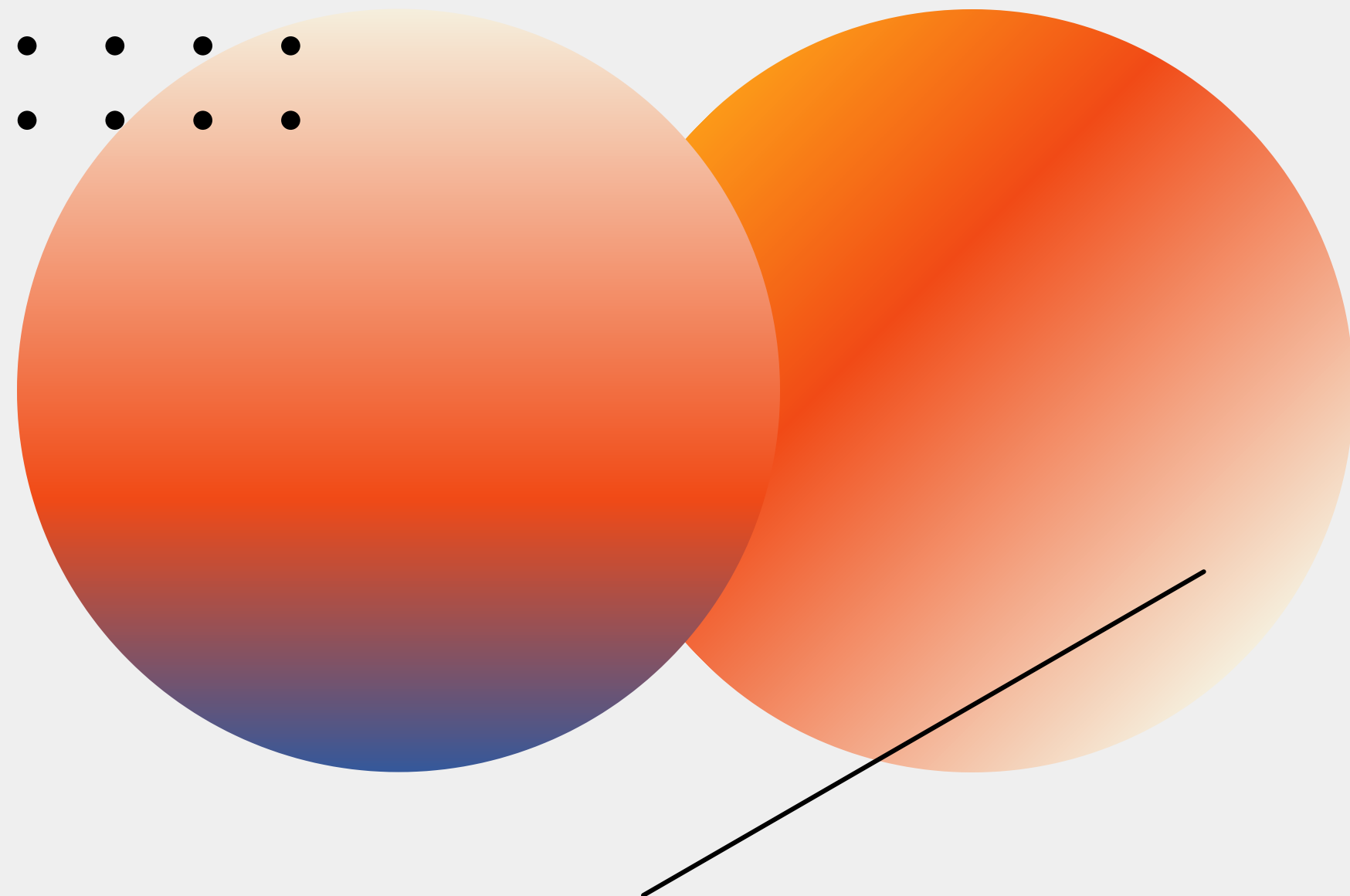
Classes

Classes

Podemos definir una clase de la siguiente manera:

```
class Greeter(prefix: String, suffix: String) {  
  def greet(name: String): Unit =  
    println(prefix + name + suffix)  
}
```

```
val greeter = new Greeter("Hello, ", "!")  
greeter.greet("Scala developer") // Hello,  
Scala developer!
```



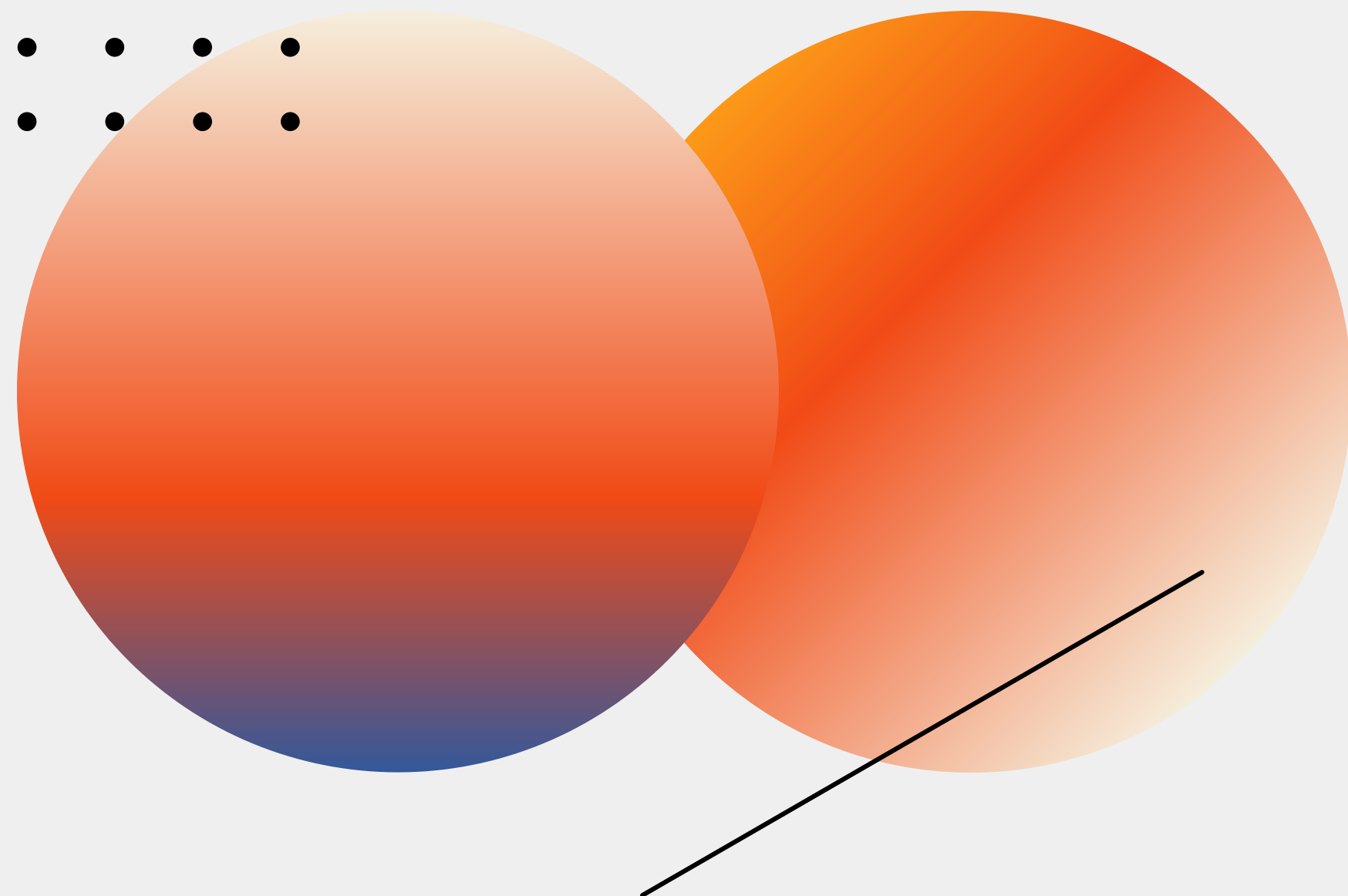
Objetos

Objetos

Son instancias únicas de sus propias definiciones. Se puede pensar en ellos como singletons de sus propias clases.

```
object IdFactory {  
  private var counter = 0  
  def create(): Int = {  
    counter += 1  
    counter  
  }  
}
```

```
val newId: Int = IdFactory.create()  
println(newId) // 1  
val newerId: Int = IdFactory.create()  
println(newerId) // 2
```



Rasgos

Rasgos

Son tipos de datos abstractos que contienen ciertos campos y métodos. En la herencia de Scala, una clase solo puede extender otra clase, pero puede extender múltiples rasgos.

```
trait Greeter {  
  def greet(name: String): Unit  
}
```

```
trait Greeter {  
  def greet(name: String): Unit =  
    println("Hello, " + name + "!")  
}
```

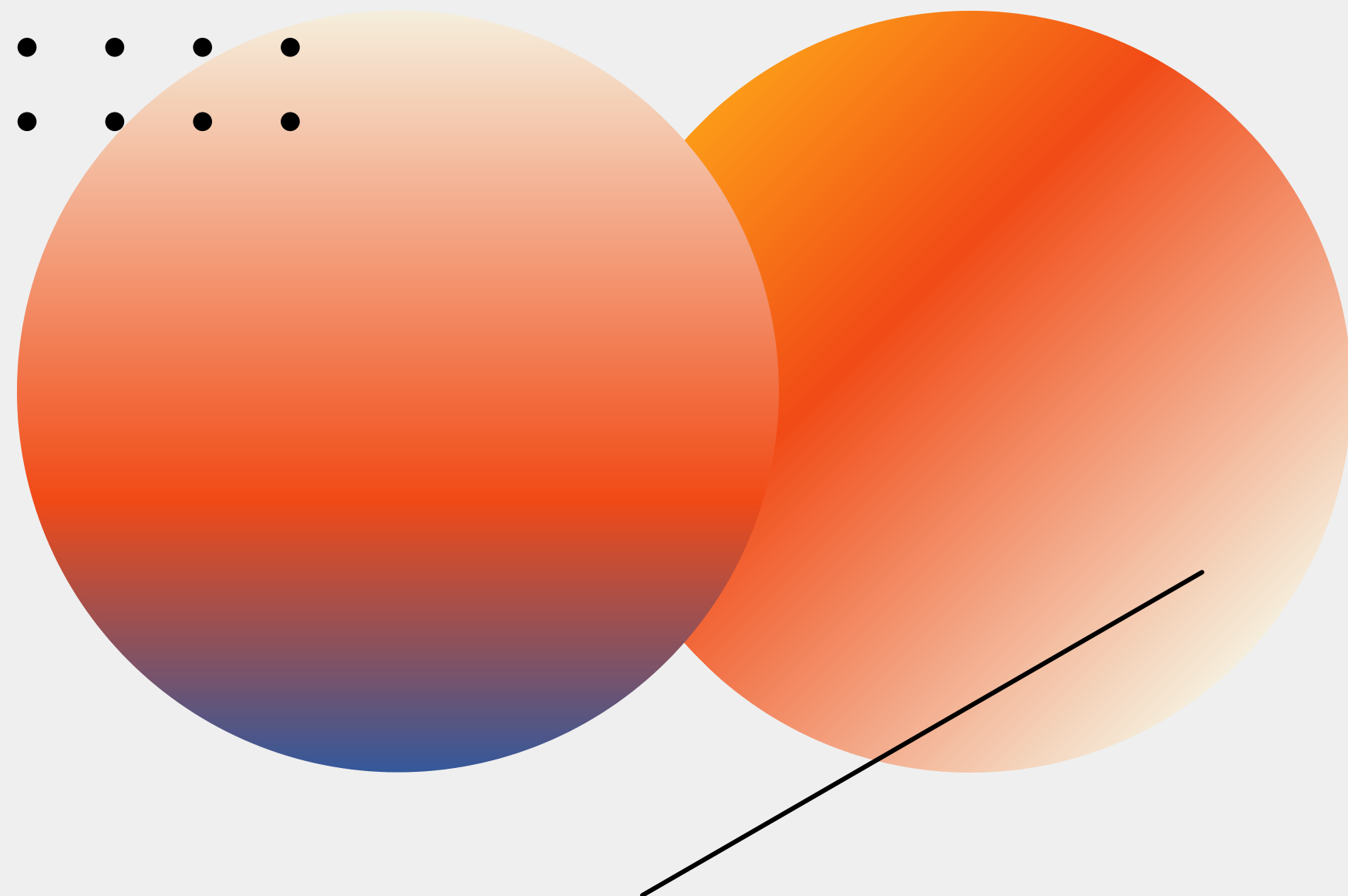
Rasgos

```
class DefaultGreeter extends Greeter
```

```
class CustomizableGreeter(prefix: String, postfix: String) extends Greeter {  
  override def greet(name: String): Unit = {  
    println(prefix + name + postfix)  
  }  
}
```

```
val greeter = new DefaultGreeter()  
greeter.greet("Scala developer") // Hello, Scala developer!
```

```
val customGreeter = new CustomizableGreeter("How are you, ", "?")  
customGreeter.greet("Scala developer") // How are you, Scala developer?
```



Case classes

Case classes

Una case class requiere la palabra clave "**case class**", un identificador y una lista de parámetros.

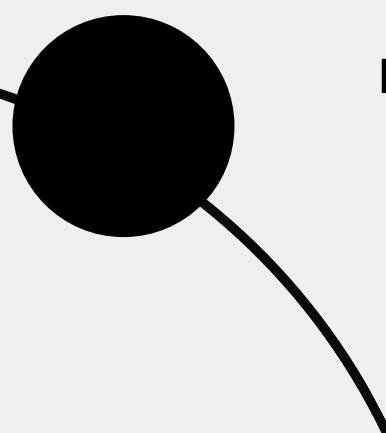
- Case classes son buenas para modelar datos inmutables:
- Instancias de case classes son comparadas por estructura y no por referencia.

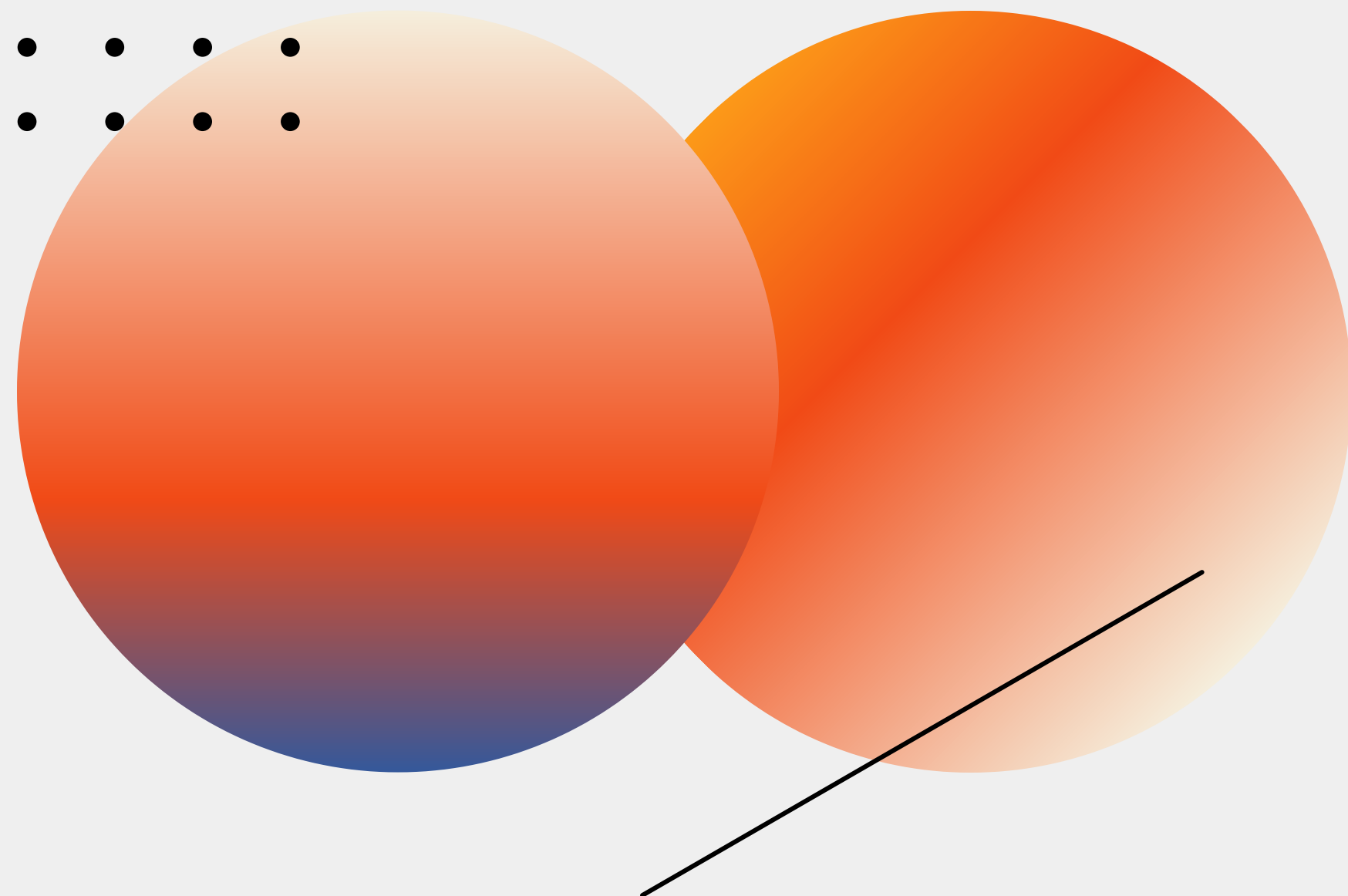
```
case class Message(sender: String,  
recipient: String, body: String)
```

```
val message1 = Message("test@email.ca",  
"test2@email.co", "Ça va ?")
```

```
println(message1.sender) // prints guillaume@quebec.ca
```

```
message1.sender = "travis@washington.us" // this line does  
not compile
```





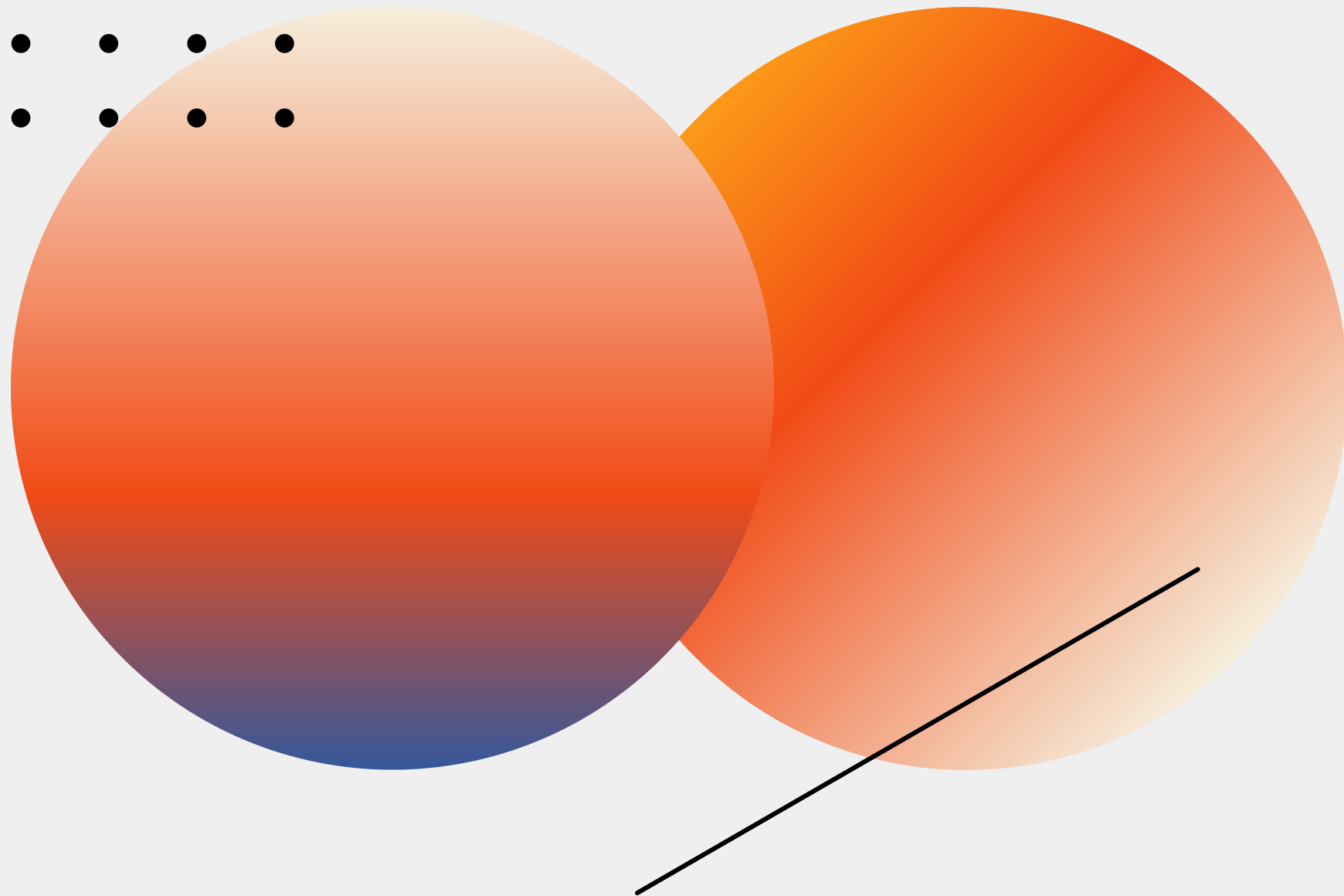
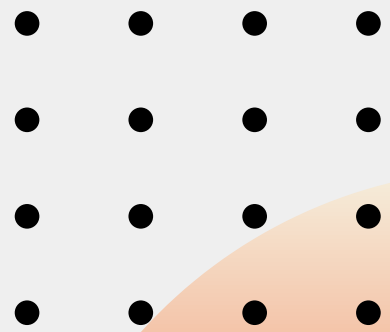
Functors

Functors

Teóricamente, el functor es un tipo de mapeo entre categorías. Dadas dos categorías A y B , un functor F asigna los objetos o entidades de A a los objetos o entidades de B . Podemos llamarlo simplemente una función de objetos o entidades.

$$\text{map} : (A \Rightarrow B) \Rightarrow F[A] \Rightarrow F[B]$$

Funciones de orden superior



Funciones de orden superior

Scala permite la definición de funciones de orden superior. Estas funciones son las que toman otras funciones como parámetros, o las cuales el resultado es una función. Aquí mostramos una función `apply` la cual toma otra función `f` y un valor `v` como parámetros y aplica la función `f` a `v`:

```
def apply(f: Int => String, v: Int) = f(v)
```



Particularidades del lenguaje

Características

Inferencia de Tipos

Por lo general, no es necesario especificar el tipo de una variable, ya que el compilador puede deducir el tipo mediante la expresión de inicialización de la variable

```
object InferenceTest1 extends App
{
  val x = 1 + 2 * 3      // El tipo
  de x es Int
  val y = x.toString()  // El tipo
  de y es String
  def succ(x: Int) = x + 1 // El
  método succ retorna valores Int
}
```

```
x: Int = 7
y: String = "7"
defined function succ
```

Características

Sintáctica Flexible

```
def isMinor(  
    p:  
    Int  
    ) = p < 18  
  
println(isMinor(18))
```

false

Punto y comas no son necesarios.
Las líneas se unen si hay
paréntesis o corchetes sin cerrar
o si hay un token en la siguiente
línea que puede continuar con la
sentencia

Características

Sintáctica Flexible

Los operadores en Scala pueden ser también métodos.

- `val sum = (a + b) * c`
`println(a.+(b).*(c))`



```
println(5.+(6).*(20))
```



```
220
```



Características

Sintáctica Flexible

Los nombres de métodos que terminan en dos puntos esperan que el argumento esté en el lado izquierdo.

```
object Demo {  
  def main(args: Array[String]) {  
    val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))  
    val nums = Nil  
  
    println( "Head of fruit : " + fruit.head )  
    println( "Tail of fruit : " + fruit.tail )  
    println( "Check if fruit is empty : " + fruit.isEmpty )  
    println( "Check if nums is empty : " + nums.isEmpty )  
  }  
}
```

```
Head of fruit : apples  
Tail of fruit : List(oranges, pears)  
Check if fruit is empty : false  
Check if nums is empty : true
```

Características

Todo es una expresión

No hace distinción entre declaraciones y expresiones. Todas las declaraciones son de hecho expresiones que se evalúan a algún valor.

```
def printValue(x: String): Unit = {  
    println("I ate a %s".format(x))  
}  
def printValue2(x: String) =  
    println("I ate a %s" format x)  
  
printValue("Manzana")  
printValue2("Pera")
```

```
I ate a Manzana  
I ate a Pera
```

Características

Match

La coincidencia de patrones es un mecanismo para comprobar un valor con un patrón. Una coincidencia exitosa también puede deconstruir un valor en sus partes constituyentes.

```
def matchTest(x: Int):  
String = x match {  
  case 1 => "one"  
  case 2 => "two"  
  case _ => "other"  
}  
matchTest(3)  
matchTest(1)
```

```
res36_1: String = "other"  
res36_2: String = "one"
```

Características

Agrupar

Se puede definir funciones dentro de una función y las funciones definidas dentro de otras funciones se denominan funciones anidadas o locales .

```
def maxAndMin(a: Int, b: Int) = {  
  def maxValue() = {  
    if(a > b){println("Max is: " + a)}  
    else{  
      println("Max is: " + b)  
    }  
  }  
  def minValue() = {  
    if(a < b){println("Min is: " + a)}  
    else{  
      println("Min is: " + b)  
    }  
  }  
  maxValue();  
  minValue();  
}
```

```
Min and Max from 5, 7  
Max is: 7  
Min is: 5
```

Características

Tail -Recursion

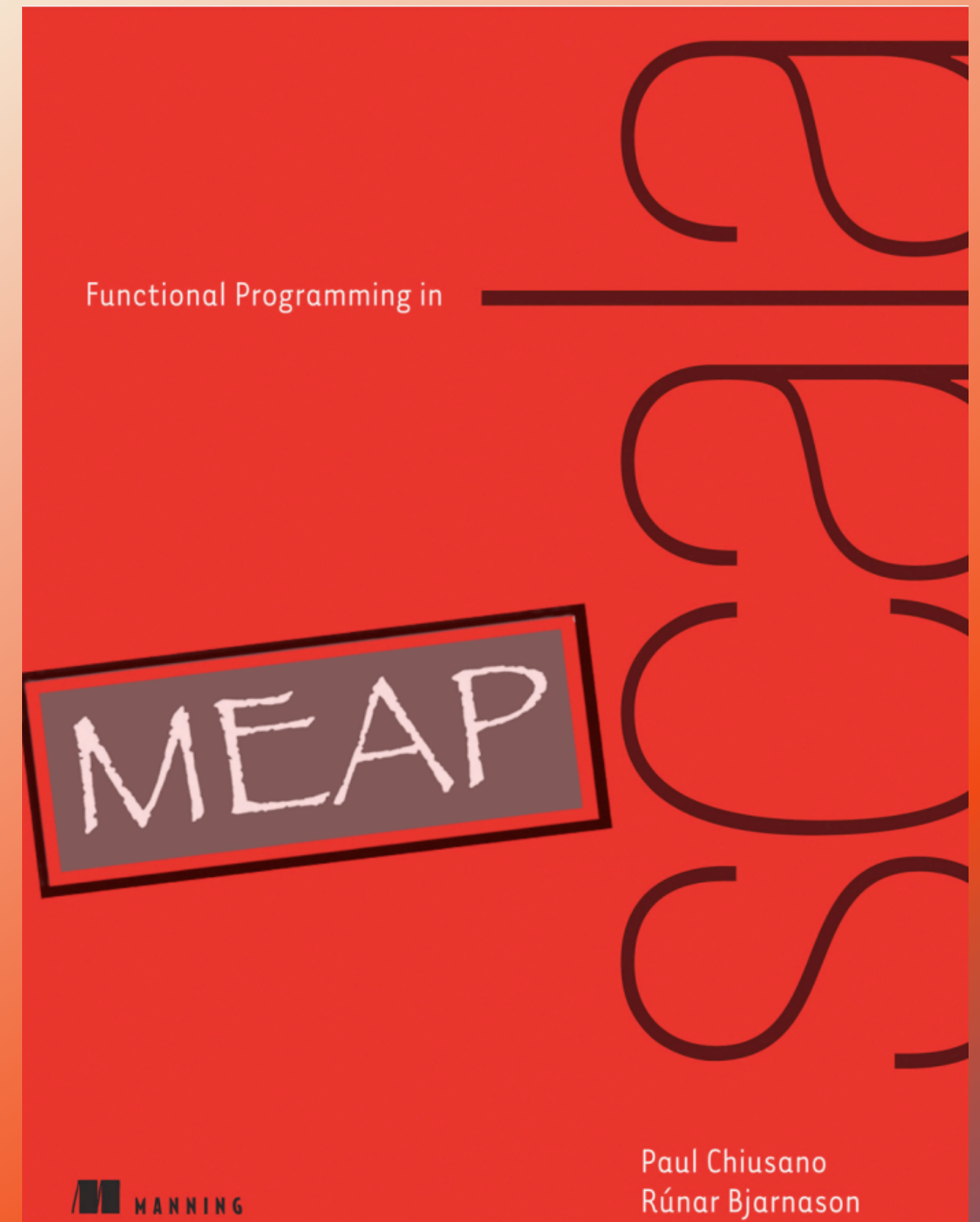
Una función recursiva usa tail recursion si la llamada a sí misma ocurre como la última acción de la función.

```
def sum(n: Int, total: Long = 0):  
    Long =  
    //def sum(n: Int): Long =  
        if (n <= 0)  
            total  
        else  
            sum(n - 1, total + n)  
    //n + sum(n - 1)
```

```
println(sumTR(1000000))
```

```
500000500000
```

Functional Programming in Scala





Iniciando con Programación Funcional

Declaración de Funciones Anónimas

En programación funcional, es tan frecuente hablar de funciones que se debe tener una manera ligera de declarar una función, sin tener que darle nombre.

```
[ ] def formatResult(name: String, n: Int, f: Int => Int) = {  
    val msg = "The %s of %d is %d."  
    msg.format(name,n, f(n))  
}
```

```
println(formatResult("increment", 7, (x: Int) => x + 1))  
println(formatResult("increment2", 7, (x) => x + 1))  
println(formatResult("increment3", 7, x => x + 1))  
println(formatResult("increment4", 7, _ + 1))  
println(formatResult("increment5", 7, x => { val r = x + 1; r })))
```

```
The increment of 7 is 8.  
The increment2 of 7 is 8.  
The increment3 of 7 is 8.  
The increment4 of 7 is 8.  
The increment5 of 7 is 8.
```



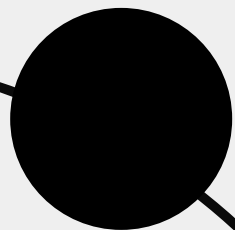
Manejo de errores sin excepciones

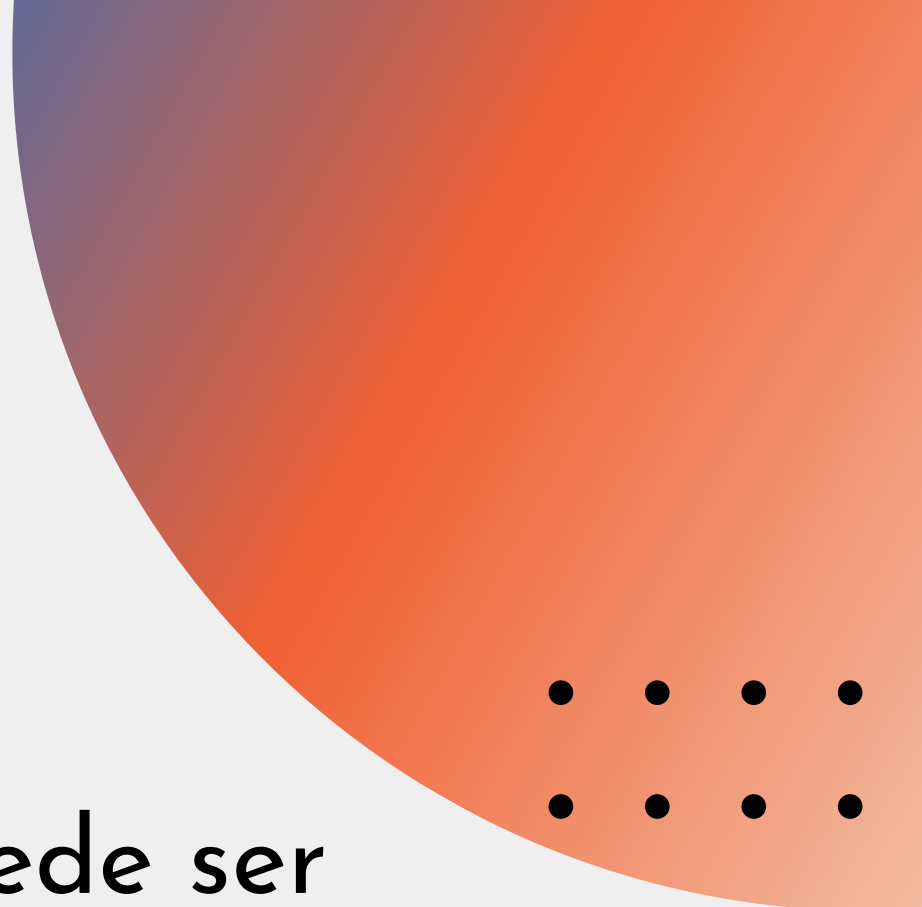
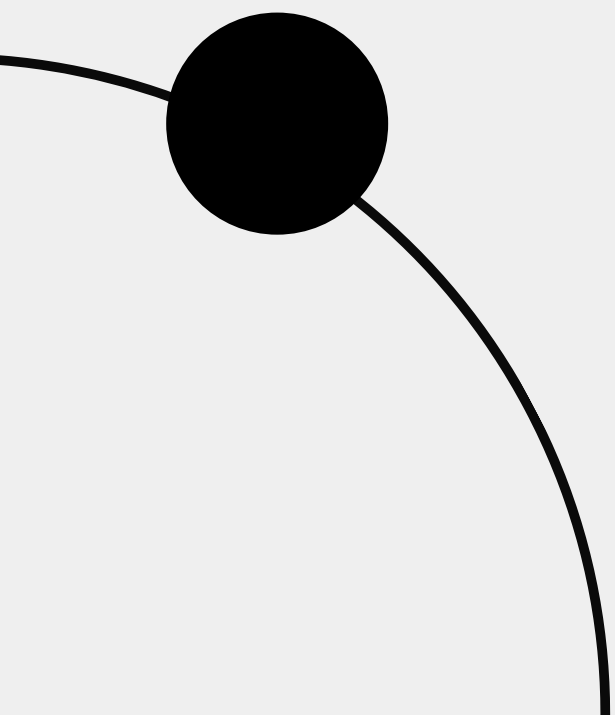
Manejo de errores sin excepciones

Consideremos una función que calcula la media de una lista de números

```
def mean(xs: Seq[Double]): Double =  
  if (xs.isEmpty)  
    throw new ArithmeticException("mean of empty list!")  
  else xs.sum / xs.length
```

- Es una función parcial



- 
- 
- Retornar siempre el valor encontrado (que puede ser NaN)
 - Recibir un argumento que nos indique qué hacer o qué retornar en caso que haya un error

Usar el tipo Option

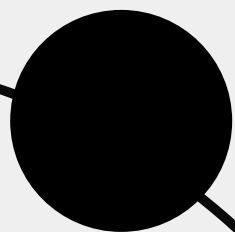
El tipo Option nos permite indicar que el resultado de la función puede o no estar definido

```
def mean_1(xs: Seq[Double]): Option[Double] =  
  if (xs.isEmpty) None  
  else Some(xs.sum / xs.length)
```

- Ahora es una función completa

Option

```
trait Option[+A] {  
  def map[B](f: A => B): Option[B]  
  def flatMap[B](f: A => Option[B]): Option[B]  
  def getOrElse[B >: A](default: => B): B  
  def orElse[B >: A](ob: => Option[B]): Option[B]  
  def filter(f: A => Boolean): Option[A]  
}
```



Option

```
case class Employee(name: String, department: String, manager: Option[String])
```

```
def lookupByName(name: String): Option[Employee] =  
  name match {  
    case "Joe" => Some(Employee("Joe", "Finances", Some("Julie")))  
    case "Mary" => Some(Employee("Mary", "IT", None))  
    case "Izumi" => Some(Employee("Izumi", "IT", Some("Mary")))  
    case _ => None  
  }
```



Option

Vamos a usar la función `lookupByName` para encontrar el departamento de un empleado

```
def getDepartment: (Option[Employee]) => Option[String] = // YOUR CODE GOES HERE  
  
getDepartment(lookupByName("Joe")) == Some("Finances")
```


Option

Vamos a usar la función `lookupByName` para encontrar el departamento de un empleado

```
def getDepartment: (Option[Employee]) => Option[String] = _.map(_.department)

getDepartment(lookupByName("Joe")) == Some("Finances")
```

FlatMap y orElse

Vamos a crear otra función que nos retorne el manager de un empleado o una opción por defecto si no tiene:

```
def getManager(employee: Option[Employee]): Option[String] = employee.flatMap(_.manager)

getManager(lookupByName("Joe")).orElse(Some("Mr. CEO"))
```



Rigurosidad y "pereza" (laziness)

Motivación

Al encadenar funciones como map, filter, fold, ... sobre una lista, Scala crea listas intermedias para cada una:

```
List(1,2,3,4) map (_ + 10) filter (_ % 2 == 0)  
List(11,12,13,14) filter (_ % 2 == 0)  
List(12,14)
```

Definiciones

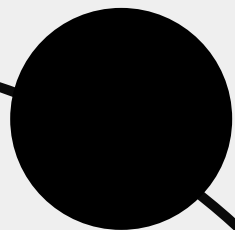
Strictness: Si la evaluación de una expresión lanza un error o es infinita, decimos que esta no termina. Una función $f(x)$ va a ser estricta si evalúa cualquier x cuya evaluación no termine.

Laziness: Hace referencia a una función que no es estricta, es decir que puede no evaluar alguno de sus argumentos.

Ejemplo

```
def square(x: Double): Double = {  
  println("Inside")  
  x * x  
}
```

```
square(41.0 + 1.0)  
square(sys.error("failure"))
```



Los operadores booleanos

```
def returnTrue() = {  
  println("Returning True")  
  true  
}  
if(false && returnTrue()) println("Inside")  
if(true || returnTrue()) println("Inside")
```



Funciones rigurosas en Scala

```
def pair(i: => Int) = (i, i)  
pair{println("Hola"); 40 + 1}
```

```
def pair2(i: => Int) = { lazy val j = i; (j, j) }  
pair2{println("Hola"); 40 + 1}
```



Volviendo al ejemplo inicial

```
Stream(1,2,3,4).map(_ + 10).filter(_ % 2 == 0)
```

```
Stream(1,2,3,4).map(_ + 10).filter(_ % 2 == 0)
11 #:: Stream(2,3,4).map(_ + 10).filter(_ % 2 == 0)
Stream(2,3,4).map(_ + 10).filter(_ % 2 == 0)
(12 #:: Stream(3,4).map(_ + 10).filter(_ % 2 == 0)
12 #:: Stream(3,4).map(_ + 10).filter(_ % 2 == 0)
12 #:: (13 #:: Stream(4).map(_ + 10).filter(_ % 2 == 0)
12 #:: Stream(4).map(_ + 10).filter(_ % 2 == 0)
12 #:: (14 #:: Stream().map(_ + 10).filter(_ % 2 == 0)
12 #:: 14 #:: Stream().map(_ + 10).filter(_ % 2 == 0)
12 #:: 14 #:: Stream()
```

El recolector de basura puede liberar el espacio de los números que no van a hacer parte del resultado final



Estado puramente funcional

Generación de números aleatorios

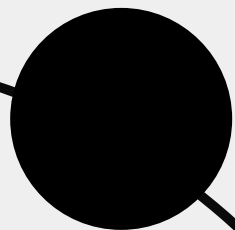
En Java:

`rng.nextDouble` -> 0.9867076608154569

`rng.nextDouble` -> 0.8455696498024141

`rng.nextInt` -> -623297295

`rng.nextInt` -> 1989531047



Generación de números aleatorios

```
trait RNG {  
  def nextInt: (Int, RNG)  
}  
  
object RNG {  
  def simple(seed: Long): RNG = new RNG {  
    def nextInt = {  
      val seed2 = (seed*0x5DEECE66DL + 0xBL)  
&  
        ((1L << 48) - 1)  
      ((seed2 >>> 16).asInstanceOf[Int],  
simple(seed2))  
    }  
  }  
}
```



Generación de números aleatorios

```
def nonNegativeInt(rng: RNG): (Int, RNG) = {  
  val (i, r) = rng.nextInt  
  (if (i < 0) -(i + 1) else i, r)  
}
```

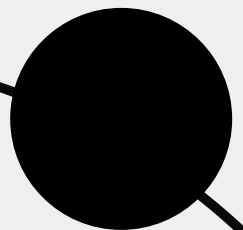
```
val rng = RNG.simple(47)
```

```
val result = nonNegativeInt(rng)  
println(result)
```

```
val result2 = nonNegativeInt(rng)  
println(result2)
```

Generación de números aleatorios

```
val rng = RNG.simple(47)  
val (result1, rng1) = nonNegativeInt(rng)  
val result2 = nonNegativeInt(rng1)._1
```



End

Thank you

Do you have any questions?

