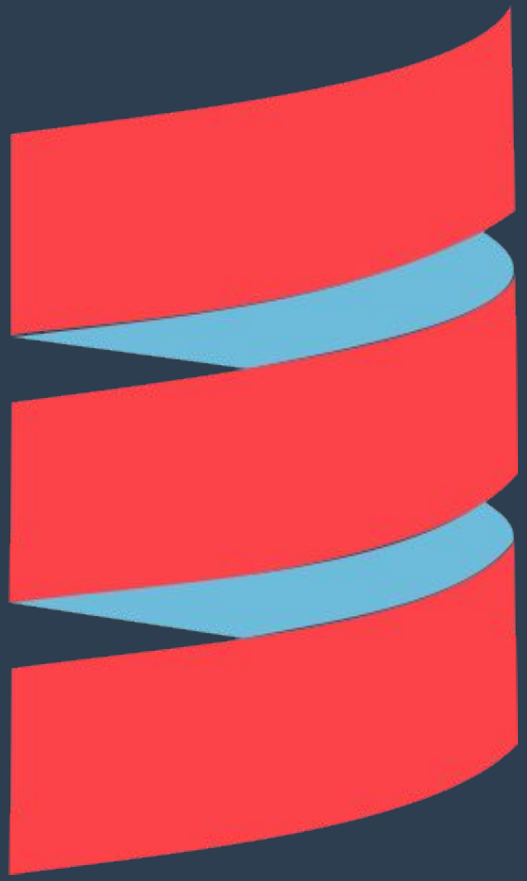
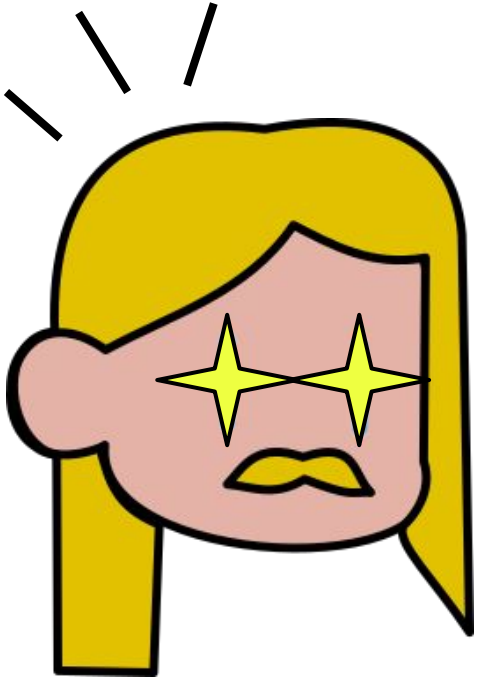


Descubriendo Scala

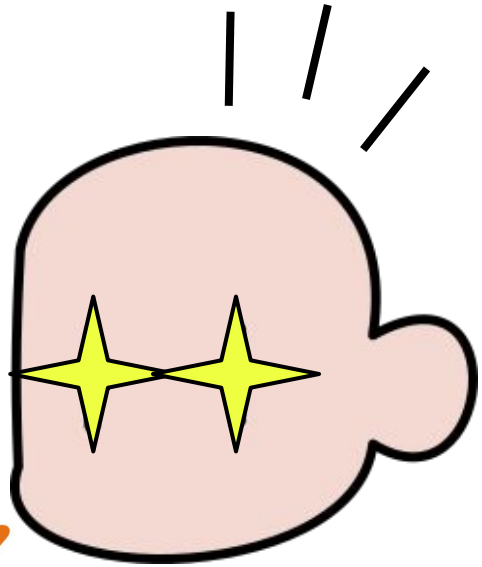
Noe Luaces
@noe_luaces

GDG DevFest 2017

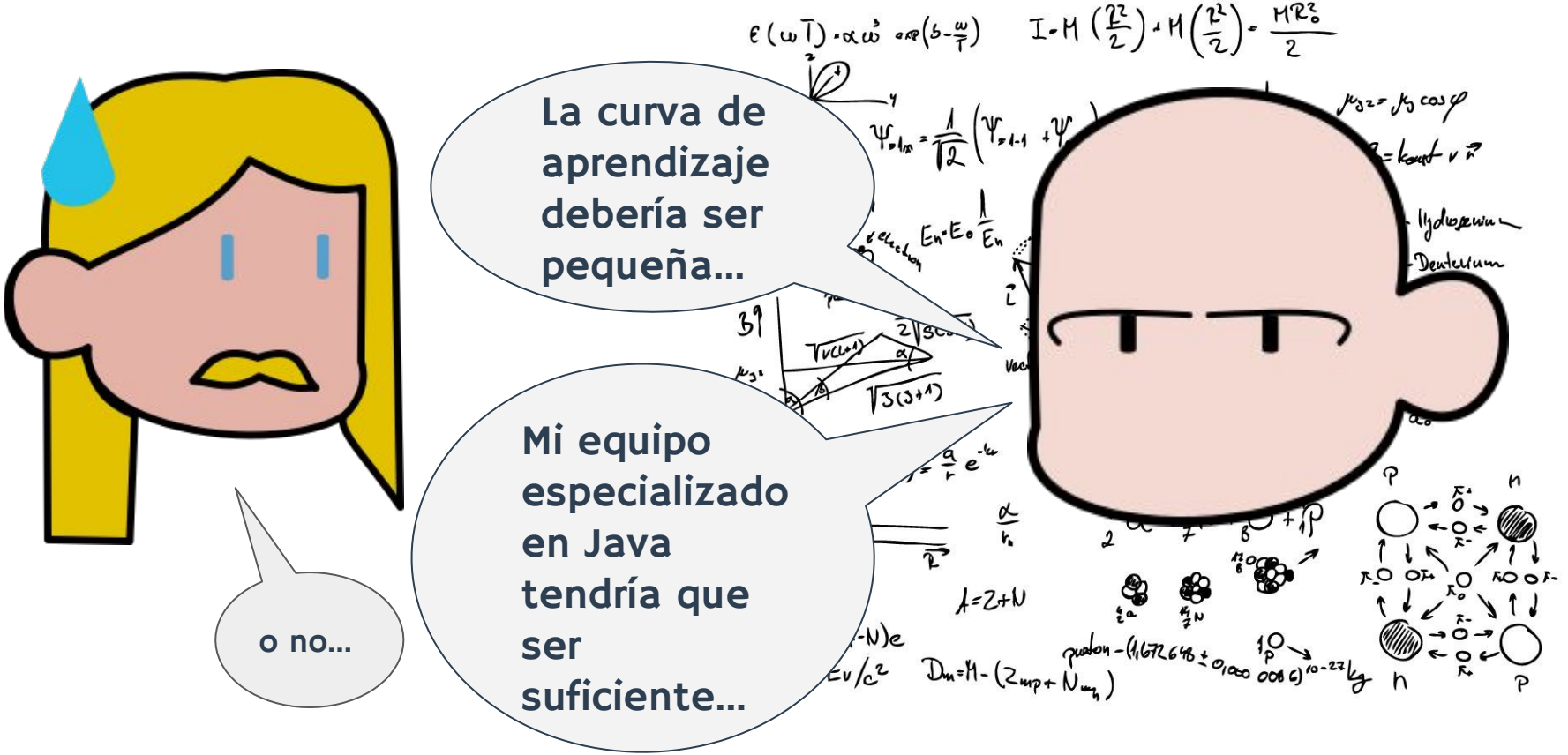
Caso típico: Oferta para nuevo proyecto



Spark



“Si sabes de Java, entonces sabes de Scala”

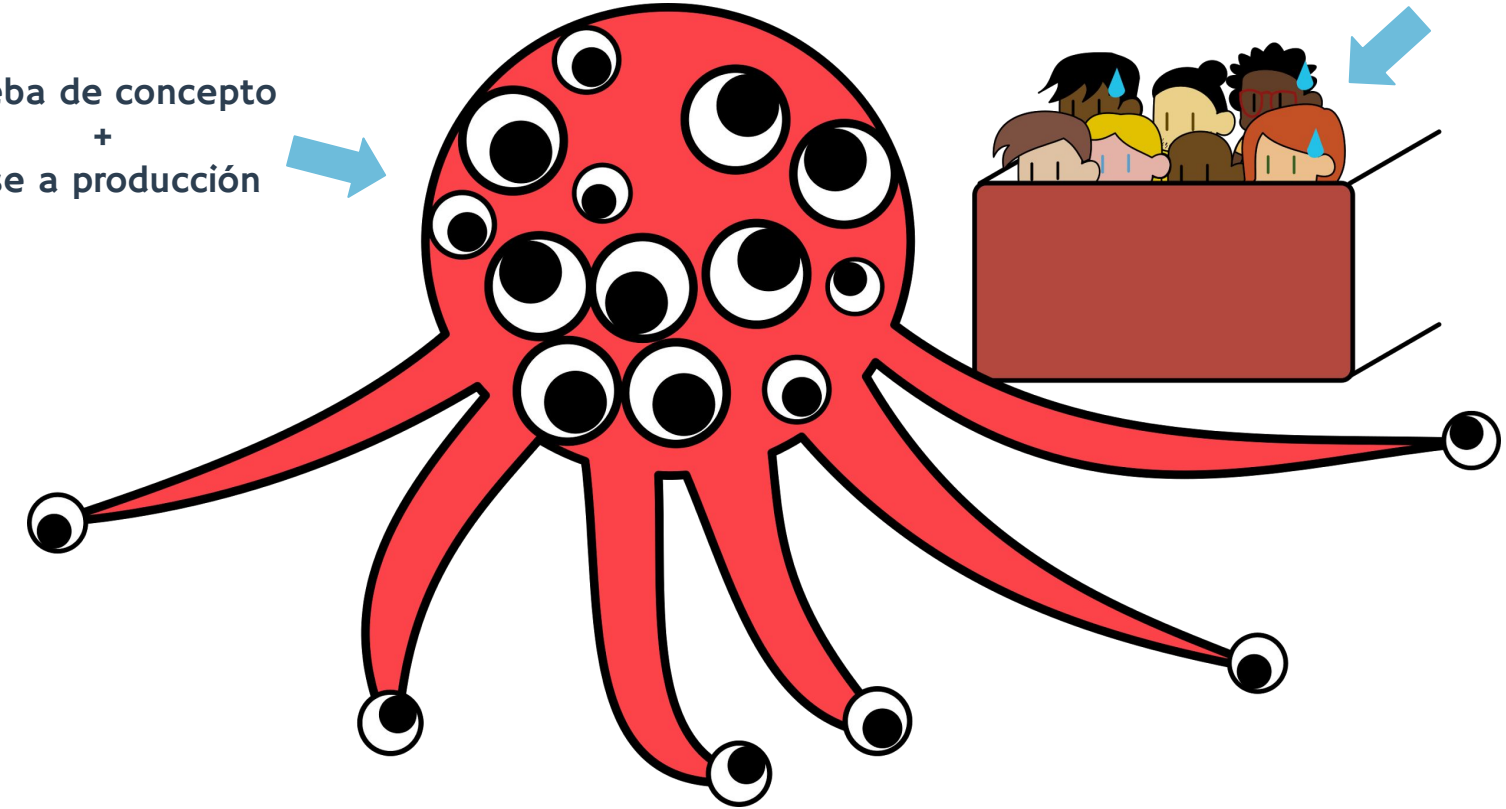


Lo que se pretendía entregar...

...y el resultado real

Desarrolladores asustados

Prueba de concepto
+
Pase a producción



Conclusiones del *Sprint Review* :

- *“Scala resulta ser innecesariamente complejo”*
- *“El código en Java es mucho más legible”*
- *“Sin duda, la programación funcional es obra de Satán”*

Java & Scala

> El compilador de Scala genera un bytecode ejecutable en JVM
> Scala puede ejecutar código en Java
> Scala es un lenguaje de Programación Orientada a Objetos

pero:

>> Scala a su vez es un lenguaje de Programación Funcional que evita variables mutables, asignaciones, bucles y demás estructuras de programación imperativa para consumir menos memoria

>> Adaptarnos a un nuevo paradigma no es trivial y siempre tendrá un coste asociado

¿Qué es Scala?

> Scala es un lenguaje que implementa la mayor parte de los conceptos de la Programación Funcional

> Cada computación es tratada como una función matemática y esto evita el almacenamiento de estados y datos mutables favoreciendo:

>> Escalabilidad en las aplicaciones
>> Reducción considerable de código
>> Programación paralela y distribuida

Evaluación de funciones

Reducción de la expresión a un valor mediante el modelo de sustitución (*substitution model*)

Call-by-value Strategy (por defecto)

```
def test(x:Int, y:Int) = x*x
```

```
test(7,2*4)
```

```
test(7,8)
```

```
7*7
```

```
49
```

Call-by-name Strategy

```
def test(x:=>Int,y:=>Int) = x*x
```

```
test(7,2*4)
```

```
7*7
```

```
49
```

Definición de valores

> La forma `def` sigue la estrategia *by-name* y su parte derecha resulta evaluada en cada uso

> La forma `val` sigue la estrategia *by-value* y su parte derecha se evalúa en la propia definición

```
def bucle: Boolean = bucle /** Definición de función **/

def x = bucle              /** Bucle sólo si se invoca x **/

val x = bucle              /** Ya provoca bucle infinito **/
```

Nesting & Lexical Scope

```
def isGoodEnough(inc:Int,x:Int)=x>inc;

def increasePosition(inc:Int,x:Int):Int=
if (isGoodEnough(inc,x)) return x+inc;
else return x;

def test(x: Int) ={
    increasePosition(1,x);
}
```

```
def test(x: Int) ={

  def isGoodEnough(inc:Int)=x>inc

  def increasePosition(inc:Int):Int=
    if (isGoodEnough(inc)) x+inc else x

  increasePosition(1)
}
```

Recursividad de cola

>> Una función es tail recursive cuando su última acción consiste en llamarse a sí misma

>> La pila de llamadas de la propia función se puede reutilizar

>> Por otro lado, tendremos que evitar usar este tipo de recursividad si la cadena de llamadas va a ser muy profunda

Higher-Order & Anonymous functions (I/3)

> Todas las funciones son valores de tipo A=>B

> El tipo recibe como parámetro A y devuelve un valor B

> Las funciones que aceptan este tipo de valores como parámetros o lo devuelven como resultados son llamadas funciones de orden superior

> Una función anónima es aquella que podemos escribir como un literal sin darle un nombre

>> Mayor flexibilidad a la hora de programar

Higher-Order & Anonymous functions (2/3)

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


def sumaEnteros(a:Int,b:Int):Int =
if(a>b) 0
else a+sumInts(a+1,b)


def sumaCubos(a:Int,b:Int):Int =
if(a>b) 0
else cubo(a)+sumaCubos(a+1,b)
```

```
def suma(f:Int=>Int,a:Int,b:Int):Int=
if(a>b) 0
else f(a) +suma(f,a+1,b)

def sumaEnteros(a:Int,b:Int)=
suma(x=>x,a,b)

def sumaCubos(a:Int,b:Int) =
suma(x=>x*x*x,a,b)

...
```

Higher-Order & Anonymous functions (3/3)

> Podemos hacer que una función devuelva otra función:

```
def suma(f:Int => Int): (Int,Int)=> Int = {
    def sumaConFuncion(a:Int, b:Int):Int =
        if (a>b) 0
        else f(a) + sumaConFuncion(a+1,b)
    sumaConFuncion
}

def sumaEnteros = suma(x =>x)
def sumaCubos = suma(x=>x*x*x)

sumaEnteros(1,10)
sumaCubos(2,3)
```

Currying

> O podemos definirla para que acepte listas de múltiples parámetros:

```
def suma(f: Int => Int)(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + suma(f)(a + 1, b)

suma(x=>x)(1,10)
suma(x=>x*x*x)(2,3)
```

> Gracias a este nivel de abstracción, podemos reducir el código y hacerlo más flexible

Definición de clases

> En Scala todos los tipos extienden clases
> Por defecto, una clase extiende de java.lang.Object

```
class Racional(x:Int, y:Int){
    def numerador = x
    def denominador = y

    override def toString(r: Racional) = r.numerador +"/"+r.denominador

    def add(r:Racional) = … /* implementación de la suma */
    def - (r:Racional) = … /* implementación de la resta*/
}
```

Instanciación de objetos

> Siguiendo el ejemplo anterior:

```
val x = new Racional(1,2)
val y = new Racional(3,5)

x.add(y)
```

> Pero lo podríamos reescribir como:

```
x add y
```

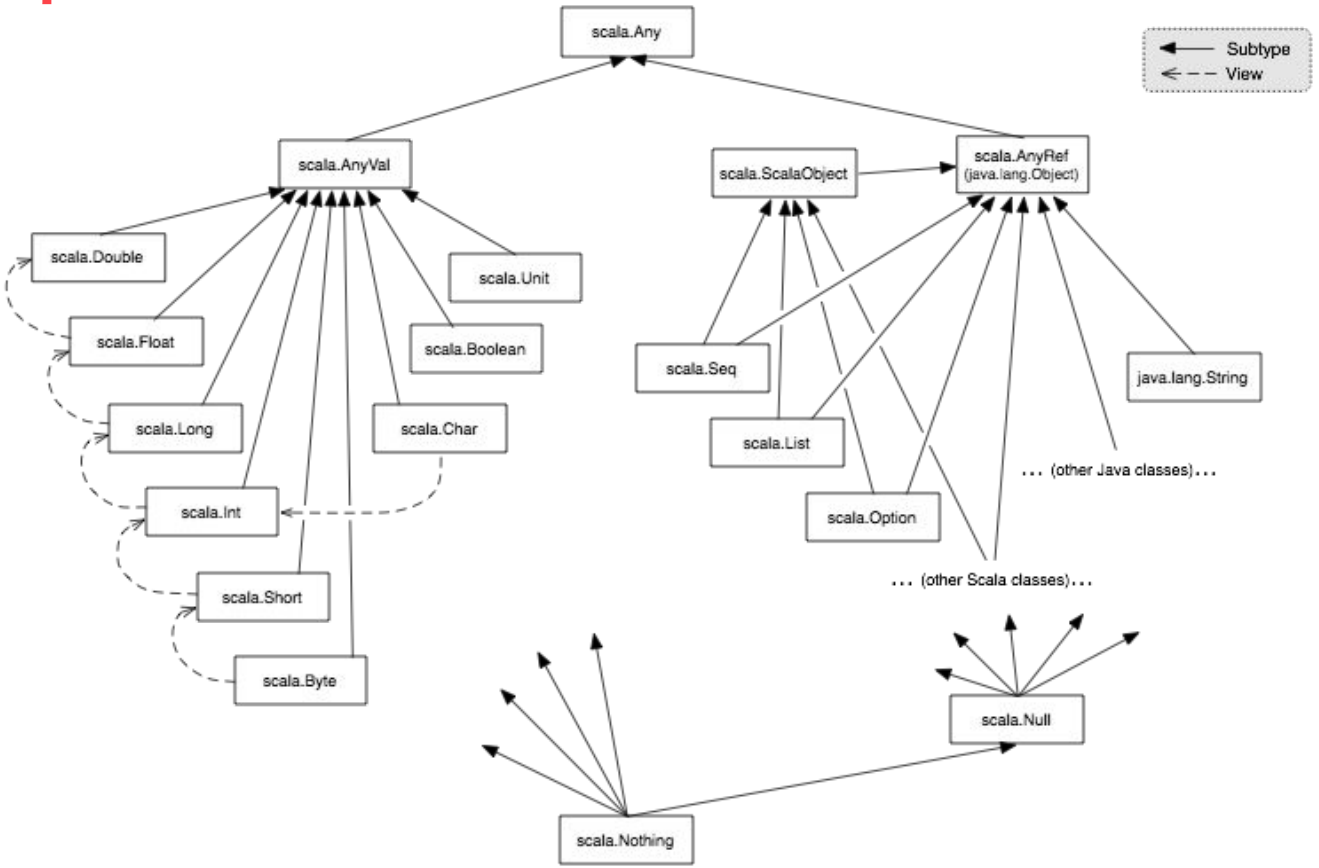
Abstract classes & Traits

> Clase abstracta : métodos sin implementación, y no se puede instanciar objetos de dicha clase
> No hay herencia múltiple
> Un trait es un supertipo que parece una interfaz de Java, pero con la ventaja de tener atributos y métodos implementados
> De esta forma, una clase sólo puede tener una superclase pero puede tener asociados múltiples traits

```
trait Planar{
  def height: Int
  def width: Int
  def surface = height * width
}
```

```
class Square extends Shape with Planar with
Movable...
```

Jerarquía de clases en Scala



Case Classes

> Son como clases normales pero son útiles para modelar datos inmutables
> No es necesario añadir new en la instaciación de objetos
> Se comparan por estructura y no por referencia

```
trait Expresion
case class Numero(n:Int) extends Expresion
case class Suma(e1: Expresion, e2: Expresion) extends Expresion

val numPar = Numero(2)
val numImpar = Numero(3)
val sum = Suma(numPar,numImpar)
val numSonIguales = numPar == Numero(2) /* true */
```

Pattern Matching

> Generalización de switch de Java en la jerarquía de clases
> Sustitución del operador instanceOf (coste alto)
> En cada caso, se asociará una expresión a un patrón:
    >> Constructores, por ej: Numero(n)
    >> Variables, por ej: num
    >> Patrones wildcard _
    >> Constantes, por ej: 1, true

```
def evaluar(e:Expresion):Int = e match {
    case Numero(n) => n
    case Suma(e1,e2) => evaluar(e1)+evaluar(e2)
}
```

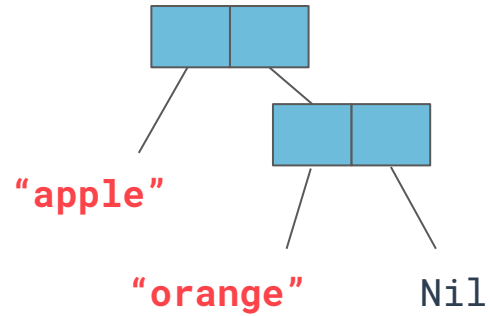
Listas

> Estructuras de datos fundamentales en programación funcional
> Son inmutables: los elementos de la lista no pueden modificarse
> Son recursivas: siguen el patrón **`x::xs`** donde **x** es el primer elemento (head) y **xs** el resto (tail)

```
val fruit = List("apple", "orange")
```

sería lo mismo que

```
val fruit = "apple" :: "orange" :: Nil
```



Parámetros implícitos

> Parámetros que si no se especifican, el compilador se encargará de pasar el argumento correcto en función del tipo especificado

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) = …
val lista = List(6,5,-2,1)
msort(lista)((x,y) => x<y)
```

```
def msort[T](xs: List[T])(ord:Ordering) = …
msort(lista)(Ordering.Int)                    //import math.Ordering
```

```
def msort[T](xs: List[T])(implicit ord:Ordering) = …
msort(lista)
```

Reducción de listas

> Combinar los elementos de la lista utilizando un operador dado

> Teóricamente la suma de los elementos de una lista se definiría como:
suma(Lista(x1,...,xn)) = 0+x1+...+xn

> Pero podremos simplificarlo como:

```
def suma(lista: List[Int]) = lista reduceLeft ((x,y) => x+y)
```

o aún más:

```
def suma(lista:List[Int]) = lista reduceLeft(_+_)
```

> Otras reducciones como reduceLeft: foldLeft, reduceRight, foldRight (para listas cuyo árbol tiende a la derecha)

Mapas (I/2)

> Otro tipo de colección fundamental en Scala
> Un mapa de tipo Map[Key, Value] es una estructura de datos que asocia claves de tipo Key a valores de tipo Value

```
val capitalPais = Map("Francia" ->"París", "Noruega"->"Oslo")
```

> Son iterables y soportan las mismas operaciones que otros iterables como Listas

> Los mapas a su vez son funciones:

```
capitalPais("Francia")  //"París"
```

Mapas (2/2)

> Dos operaciones muy útiles de SQL para manejar mapas son groupBy y orderBy

```
val fruta = List("manzana","pera", "cereza")
fruta sortWith (_.length < _.length)  // List("pera", "cereza","manzana")
fruit.sorted                          //List("cereza","manzana","pera")
```

> groupBy particiona una colección en un mapa de acuerdo a una función discriminadora

```
val fruta = List("manzana","pera", "cereza","piña")
fruta groupBy (_.head)

// Map(p->List(pera, piña), c ->List(cereza),m->List(manzana))
```

Conclusiones:

- Scala es un lenguaje con un paradigma muy distinto a lo que la programación en Java se refiere
- Tiene muchas características muy potentes que reducen drásticamente el nivel de código en un proyecto
- Proporciona flexibilidad y herramientas de optimización, lo que beneficia la programación escalable y distribuida