



THE FUNCTIONAL  
ARCHITECTURE STUDIO



# Introducción a Scala

*Habla Computing*  
[info@hablapps.com](mailto:info@hablapps.com)  
[@hablapps](#)

# Índice

- Introducción
- Características de Orientación a Objetos
- Características de Programación Funcional
- Azúcar Sintáctico
- Implícitos
- Características de Programación Genérica
- Conclusiones



# Curso de Introducción a Scala

## *1. Introducción*

# Introducción

## *Temario*

- Audiencia
  - Programadores OO con curiosidad por Scala
- Objetivos
  - Tener nociones básicas de las características principales de Scala para los diversos paradigmas
  - Mejorar las habilidades del alumno para comprender código fuente y documentación en Scala
- Temario
  - Orientación a Objetos
  - Programación Funcional
  - + Azúcar Sintáctico
  - Programación Genérica
- Organización
  - Demostraciones “en vivo” siempre que sea posible
  - Ejercicios al finalizar cada módulo
  - Punteros a contenido más avanzado

# Introducción

## *Sobre Scala*

- Creado por Martin Odersky
- Aparece en 2003
- EPFL / Lightbend
- Lenguaje multi-paradigma
- Tipado: estático / fuerte
- JVM + Javascript
- Ecosistema de moda (spark, akka...)
- *Todo* es un objeto
- Scala 3 (dotty) finales de 2020





# **Curso de Introducción a Scala**

## *2. Características de Orientación a Objetos*



# Introducción

## *Hello World!*



```
object HolaMundo extends App {  
    println("Hola Mundo!")  
}
```



# Características de OO

## *Objetivos*

- Poder aplicar los aspectos fundamentales del paradigma OO en Scala: clases, atributos, métodos...
- Identificar otros conceptos que Scala introduce para lidiar con este paradigma: objetos, traits...
- Aprender a declarar herencia múltiple muy básica


# Características de OO

## *Guión*


- ❖ Clases, Atributos y Constructores
- ❖ Métodos
- ❖ Herencia simple
- ❖ (Singleton & Companion) Objects
- ❖ Traits

# Características de OO

## *Clases, Atributos y Constructores (1/4)*




```
class Bicicleta(  
    _cadencia: Int,  
    _marcha: Int,  
    _velocidad: Int) {  
    var cadencia: Int = _cadencia  
    var marcha: Int = _marcha  
    var velocidad: Int = _velocidad  
}
```




section2-oo/code/Bicicleta.scala

# Características de OO

## *Clases, Atributos y Constructores (2/4)*



```
class Bicicleta(  
    _cadencia: Int,  
    _marcha: Int,  
    _velocidad: Int) {  
    val cadencia: Int = _cadencia  
    val marcha: Int = _marcha  
    val velocidad: Int = _velocidad  
}
```



section2-oo/code/Bicicleta.scala

# Características de OO


## *Clases, Atributos y Constructores (3/4)*




```
class Bicicleta(  
  val cadencia: Int,  
  val marcha: Int,  
  val velocidad: Int)
```

# Características de OO

## *Clases, Atributos y Constructores (4/4)*



```
class Bicicleta(  
    val cadencia: Int,  
    val marcha: Int,  
    val velocidad: Int) {  
    def this(_cadencia: Int, _marcha: Int) = {  
        this(_cadencia, _marcha, 1)  
    }  
}
```



section2-oo/code/Bicicleta.scala

# Características de OO

## *Métodos*

```
class Bicicleta(  
    val cadencia: Int,  
    val marcha: Int,  
    val velocidad: Int) {  
  
    def frenar(decremento: Int): Bicicleta = {  
        new Bicicleta(  
            cadencia,  
            marcha,  
            velocidad - decremento)  
        }  
    }  
}
```



# Características de OO

## *Singleton Objects*


```
object FabricaDeBicicletas {  
    val cadenciaInicial = 0  
    val marchaInicial = 1  
    val velocidadInicial = 0  
  
    def crear: Bicicleta = {  
        new Bicicleta(  
            cadenciaInicial,  
            marchaInicial,  
            velocidadInicial)  
        }  
    }  
}
```





# Características de OO

## *Companion Objects*



```
object Bicicleta {  
  def crear(  
    cadencia: Int,  
    marcha: Int,  
    velocidad: Int): Bicicleta = {  
    new Bicicleta(cadencia, marcha, velocidad)  
  }  
}
```

# Características de OO

## *Herencia Simple*



```
class BicicletaDeMontaña(  
    val alturaSillin: Int,  
    cadencia: Int,  
    marcha: Int,  
    velocidad: Int)  
    extends Bicicleta(cadencia, marcha, velocidad)
```

# Características de OO

## *Traits*

```
trait Motor {  
    val revoluciones: Int  
    val cilindrada: Int = 55  
}
```

```
class Motocicleta(  
    cadencia: Int,  
    marcha: Int,  
    velocidad: Int)  
    extends Bicicleta(cadencia, marcha, velocidad) with Motor {  
    val revoluciones = 2500  
}
```

section2-oo/code/Bicicleta.scala

# Características OO

## *Ejercicios*

Los ejercicios para este módulo se encuentran en:

**`src/main/.../curso/exercise1-oo/Ejercicios.scala`**

# Características de OO

## *Takeaways y cómo seguir*

- Takeaways
  - *Scala as a better Java*
  - La existencia de *objects* elimina la necesidad de utilizar el modificador *static* para crear miembros de clase
  - Se permite la herencia múltiple con traits
- ¿Por dónde seguir?
  - Resolución de herencia múltiple [Linearization](#)



# Curso de Introducción a Scala

## *4. Azúcar Sintáctico 1*

# AZÚCAR MORENO

desde el principio



# Azúcar Sintáctico

## *Objetivos*

- Mejorar las habilidades del alumno para comprender código fuente en Scala
- Conocer diversas alternativas para declarar una misma instrucción
- Adquirir nociones básicas sobre qué estilo es preferible para según qué tarea



# Azúcar Sintáctico

## *Guión*

- ❖ Invocación de Métodos
- ❖ Parámetros por Defecto
- ❖ Métodos *Variadic*

# Azúcar Sintáctico

## *Invocación de Métodos (1/3)*



```
class Azucar {  
  def f1(a: Int): Int = a  
}
```

```
scala> val azucar = new Azucar // Sin paréntesis  
azucar: Azucar = ...
```

```
scala> azucar.f1(1)  
res0: Int = 1
```

```
scala> azucar.f1 { 1 }           // similar a azucar.f1({1})  
res1: Int = 1
```

```
scala> azucar f1 1  
res2: Int = 1
```

# Azúcar Sintáctico

## *Invocación de Métodos (2/3)*

```
class Azucar {  
  def f2(a: Boolean, b: String, c: String): String =  
    if (a) b else c  
}
```

```
scala> azucar.f2(true, "then", "else")  
res3: String = then
```

```
scala> azucar f2 (true, "then", "else")  
res4: String = then
```

```
scala> azucar.f2(a=true, b="then", c="else")  
res5: String = then
```

```
scala> azucar.f2(b="then", c="else", a=true)  
res6: String = then
```

# Azúcar Sintáctico

## *Invocación de Métodos (3/3)*

```
class Credentials(name: String, secret: String)
```

```
class Password(secret: String)
```

```
class User(name: String) {  
  def ++(p: Password): String =  
    new Credentials(name, p.secret)  
}
```

```
scala> val usuario = new User("myName")
```

```
usuario: User = User("myName")
```

```
scala> val pass = new Password("my super secret")
```

```
pass: Password = Password("mySuperSecret")
```

```
scala> val cred = usuario ++ pass
```

```
pass: Credentials = Credentials("myName","my super secret")
```



# Azúcar Sintáctico

## *Parámetros por Defecto*




```
class Azucar {  
  def f3(  
    a: Boolean,  
    b: String = "then",  
    c: String = "else"): String = {  
    if (a) b else c  
  }  
}
```

```
scala> azucar.f3(true, "txt1", "txt2")  
res0: String = txt1
```

```
scala> azucar.f3(true)  
res1: String = then
```

# Azúcar Sintáctico

## *Métodos Variadic*



```
class Azucar {  
  def f4(a: Int*): Int = a.reduce { (a1, a2) =>  
    a1 + a2  
  }  
}
```

```
scala> azucar.f4(1)  
res0: Int = 1
```

```
scala> azucar.f4(1, 2, 3, 4, 5)  
res1: Int = 15
```

# Azúcar Sintáctico

## *Takeaways y cómo seguir*

- Takeaways
  - Scala despliega una gran variedad de azúcar sintáctico, lo que lo convierte en un lenguaje muy flexible
  - Es importante tener unas nociones básicas sobre estas técnicas para poder comprender código escrito por terceros
  - Controlar estas técnicas permite adecuar nuestro estilo de programación al posible lector (incluso no expertos)
- ¿Por dónde seguir?
  - [Programming in Scala](#): **Implicits**, **for-comprehensions**, Lambdas, Currying, etc.
  - [String Interpolation](#)
  - [Scala Style Guide](#)
  - [DSLs in Action](#) ([ScalaTest](#), [Spray](#), [Embedded BASIC](#), etc.)
  - [Scala Macros](#)



# Curso de Introducción a Scala

## *4. Novedades en objetos en dotty (scala 3)*



# Intersection types

No es novedad, pero simplifica su sintaxis.

```
val l1: List[String] = List("hola","adios")
```

```
val l2: List[Int] = List(1,2,3)
```

```
val l3: List[String | Int] = l1 :: l2
```