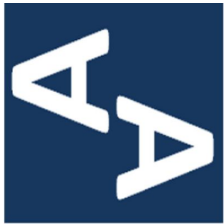




Introducción a Scala



Jesús López González

[Habla Computing](#)

jesus.lopez@hablapps.com

[@jeslg](#)

Índice

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



Curso de Introducción a Scala

1. Introducción

Introducción

Sobre el Curso

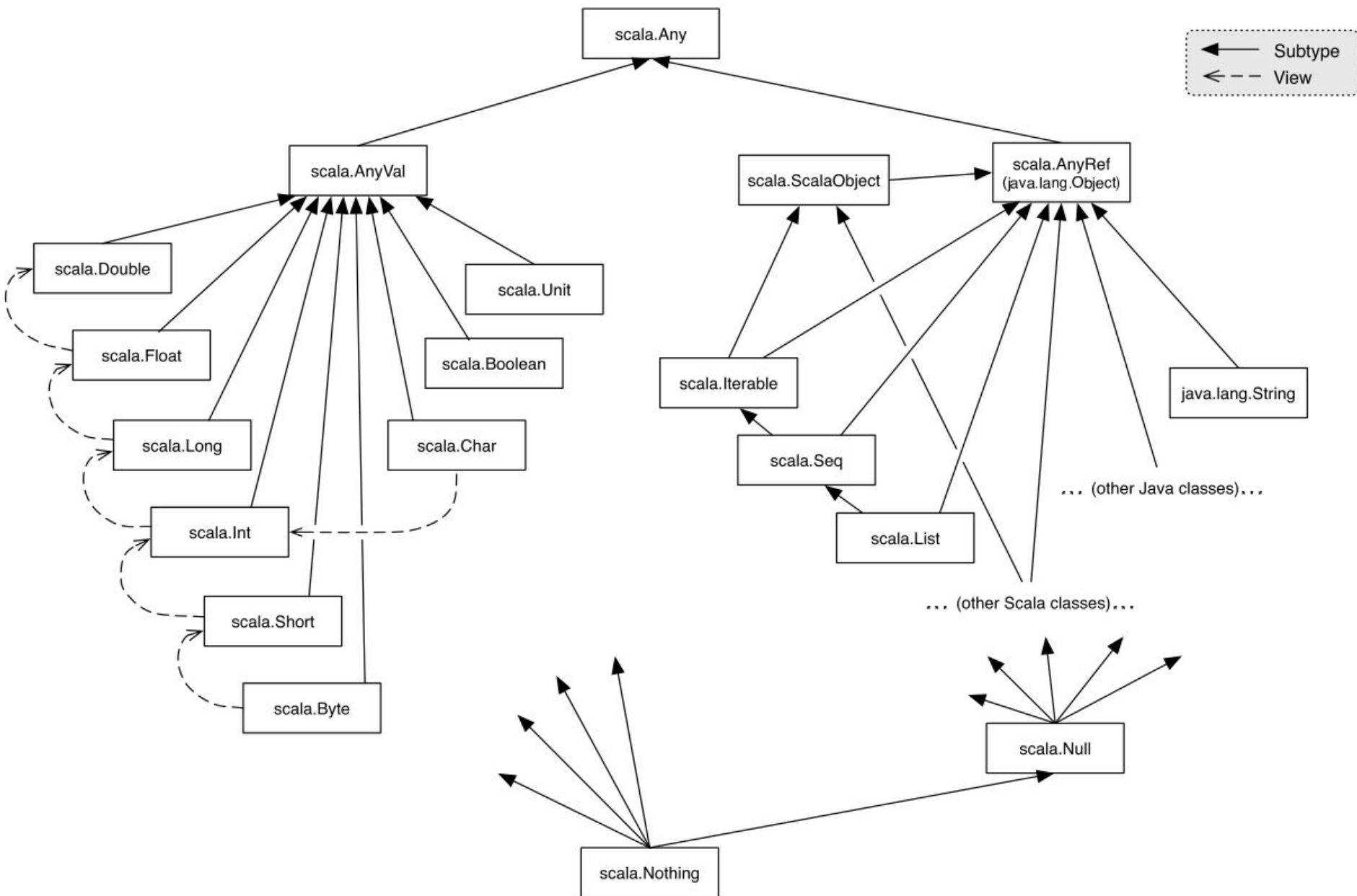
- Audiencia
 - Programadores OO con curiosidad por Scala
- Objetivos
 - Tener nociones básicas de las características principales de Scala para los diversos paradigmas
 - Mejorar las habilidades del alumno para comprender código fuente y documentación en Scala
- Temario
 - Orientación a Objetos
 - Programación 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 / Typesafe
- Lenguaje multi-paradigma
- JVM
- Ecosistema de moda
- Todo es un objeto





Introducción

Hello World!



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

[org/hablapps/cursos/intro/HolaMundo.scala](https://github.com/hablapps/cursos/blob/master/intro/HolaMundo.scala)



Curso de Introducción a Scala

2. Características de Orientación a Objetos

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

- ❖ 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  
}
```

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  
}
```

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)  
    }  
}
```

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)  
        }  
    }  
}
```



org/hablapps/curso/oo/bicicleta.scala

Características de OO

Singleton Objects




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

org/hablapps/curso/oo/bicicleta.scala

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  
}
```

Características OO

Ejercicios

1. Crea un trait ``Fruta``, que tenga como métodos: ``color`` (String) y ``precio`` (Double)
2. Crea un companion object para este trait, que contenga un método ``crearFruta``, que reciba el color como parámetro y que fije el precio a 3.0.
3. Añade una clase ``Platano``, que herede de ``Fruta``, fijando el color a "amarillo" y recibiendo el precio en el propio constructor.
4. Crea una fruta de cualquier color y un plátano con cualquier precio. Imprime por pantalla el precio de la fruta (debería ser 3.0) y el color del plátano (debería ser "amarillo")

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](#)
 - Inyección de dependencias [Cake Pattern](#)



Curso de Introducción a Scala

3. Características de Programación Funcional

Características de PF

Objetivos

- Adquirir unas nociones básicas sobre qué es la programación funcional
- Aprender técnicas fundamentales para el trabajo con el paradigma funcional: inmutabilidad, pattern matching, etc.
- Descubrir las lambdas, funciones que se tratan como valores (*first-order citizens*)

Características de PF

- ❖ ¿Qué es la PF?
- ❖ Inmutabilidad
- ❖ Case Classes & Pattern Matching
- ❖ Lambdas

Características de PF

¿Qué es la PF? (1/2)

Programar con Funciones Puras

Una Función Pura es aquella que realiza únicamente lo que declara su signature. Es decir, transforma unos valores de entrada en unos valores de salida, sin llevar a cabo ningún efecto de lado adicional que sea observable desde el exterior

Características de PF

¿Qué es la PF? (2/2)

```
def pure(a: Int, b: Int): Int = a + b
```

```
var res: Int = 0
```

```
def impure(a: Int, b: Int): Int = {  
  res = a + b  
  a + b  
}
```





Características de PF

Inmutabilidad (1/2)

```
val x = 0
```

```
x = 1
```



Características de PF

Inmutabilidad (2/2)



```
sealed trait Lista {  
  def insertar(elemento: Int): Lista = {  
    new Cons(elemento, this)  
  }  
}
```

```
class Cons(  
  val cabeza: Int,  
  val resto: Lista) extends Lista
```

```
class Nada() extends Lista
```


Características de PF

Case Classes

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

Características de PF

Pattern Matching



```
def matchTest(x: Any): Any = x match {  
  case 1          => "one"  
  case "two"      => 2  
  case y: Int     => "scala.Int"  
  case _          => 4.0  
}
```

Características de PF

Case Classes & Pattern Matching



```
sealed trait Lista {  
  def suma: Int = this match {  
    case Cons(cabeza, resto) => cabeza + resto.suma  
    case Nada() => 0  
  }  
}  
  
case class Cons(cabeza: Int, resto: Lista) extends Lista  
  
case class Nada() extends Lista
```


Características de PF

Lambdas (1/2)



```
(x: Int) => x + 1
```

```
(x: Int, y: Int) => "(" + x + ", " + y + ")"
```

Características de PF

Lambdas (2/2)



```
sealed trait Lista {  
  def map(f: Int => Int): Lista = this match {  
    case Cons(cabeza, resto) => {  
      Cons(f(cabeza), resto.map(f))  
    }  
    case Nada => Nada  
  }  
}  
  
case class Cons(cabeza: Int, resto: Lista) extends Lista  
  
case object Nada extends Lista
```

Características de PF

Ejercicios

1. Implementa el método ``concatenar`` (`Lista.scala`) que recibe dos listas como entrada y genera la concatenación de las mismas.
2. Implementa el método ``existe``, que recibe una lista y una función de tipo ``Int => Boolean`` como parámetro, y que devolverá un booleano para indicar si existe algún elemento en la lista que cumpla tal condición.
3. Implementa el método ``contiene``, que recibe una lista y un entero para indicar si existe dicho elemento en la lista. Esta implementación deberá apoyarse en el resultado del ejercicio 2.
4. Implementa ``tirarMientras`` que “tira” todos aquellos elementos de la cabeza que no cumplan con la condición establecida por una función de entrada.

Características de PF

Takeaways y cómo seguir

- Takeaways
 - La Programación Funcional se basa en el uso de Funciones Puras
 - Inmutabilidad, pattern matching y lambdas son patrones de diseño habituales para trabajar en el paradigma de PF
 - Hemos visto la punta (de la punta) del iceberg
 - Es un paradigma complejo, pero merece la pena
- ¿Por dónde seguir?
 - “El libro rojo” [Functional Programming in Scala](#)
 - Librerías: [scalaz](#), [cats](#), etc.
 - [Haskell](#)
 - [Category Theory](#)



Curso de Introducción a Scala

4. Azúcar Sintáctico

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

- ❖ Invocación de Métodos
- ❖ Parámetros por Defecto
- ❖ Métodos *Variadic*
- ❖ El método *apply*
- ❖ *Placeholder* Lambdas

Azúcar Sintáctico

Invocación de Métodos (1/3)

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

```
scala> val azucar = new Azucar  
azucar: Azucar = ...
```

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

```
scala> azucar.f1 { 1 }  
res1: Int = 1
```

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

org/hablapps/curso/azucar/Azucar.scala

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
```

org/hablapps/curso/azucar/Azucar.scala

Azúcar Sintáctico

Invocación de Métodos (3/3)



SC

```
ala> val lista1 = Cons(1, Cons(2, Nada()))
```

```
lista1: Cons = Cons(1,Cons(2,Nada()))
```

```
scala> lista1 contiene 2
```

```
res0: Boolean = true
```

```
scala> lista1 ++ lista1
```

```
res1: Lista = Cons(1,Cons(2,Cons(1,Cons(2,Nada()))))
```

Azúcar Sintáctico

Parámetros por Defecto (1/2)

```
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
```

org/hablapps/curso/azucar/Azucar.scala



Azúcar Sintáctico


Parámetros por Defecto (2/2)

```
case class Cons(  
  cabeza: Int,  
  resto: Lista = Nada()) extends Lista
```

```
scala> Cons(1, Cons(2))  
res0: org.hablapps.curso.azucar.Cons = Cons(1,Cons(2,Nada()))
```

Azúcar Sintáctico

Métodos Variadic (1/2)



```
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
```

org/hablapps/curso/azucar/Azucar.scala

Azúcar Sintáctico

Métodos Variadic (2/2)



```
object Lista {  
  def crear(es: Int*): Lista = {  
    if (es.isEmpty)  
      Nada()  
    else  
      Cons(es.head, crear(es.tail: _*))  
  }  
}
```

```
scala> Lista.crear(1,2,3)  
res0: Lista = Cons(1, Cons(2, Cons(3, Nada())))
```

org/hablapps/curso/azucar/lista.scala

Azúcar Sintáctico

Método apply (1/2)




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

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

```
scala> azucar(1)  
res1: Int = 1
```

Azúcar Sintáctico

Método apply (2/2)



```
object Lista {  
  def apply(es: Int*): Lista = {  
    if (es.isEmpty)  
      Nada()  
    else  
      Cons(es.head, apply(es.tail: _*))  
  }  
}
```

```
scala> Lista(1,2,3)  
res0: Lista = Cons(1,Cons(2,Cons(3,Nada())))
```


Azúcar Sintáctico

Placeholder Lambdas (1/2)



```
class Azucar {  
  def f5(f: Int => String): Unit = ()  
}
```

```
scala> azucar.f5((x: Int) => x.toString)
```

```
scala> azucar.f5(x => x.toString)
```

```
scala> azucar.f5(_.toString)
```

Azúcar Sintáctico

Placeholder Lambdas (2/2)



```
scala> val l = List(1, 2, 3)
```

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

```
scala> l.map(_ + 1)
```

```
res0: List[Int] = List(2, 3, 4)
```



Azúcar Sintáctico

Ejercicios (1/2)

1. El fichero de configuración de un proyecto SBT (*build.sbt*) es en sí mismo un fichero Scala. ¿Qué crees que está ocurriendo cuando se define la siguiente propiedad?

```
name := "scalaintrocourse"
```

2. El siguiente fragmento pertenece a la [sección de routing](#) del tutorial oficial del framework Play. ¿Qué elementos podrías identificar?

```
def show(id: Long) = Action{  
  Client.findById(id).map { client =>  
    Ok(views.html.Clients.display(client))  
  }.getOrElse(NotFound)  
}
```

Azúcar Sintáctico

Ejercicios (2/2)

3. ¿Qué comprueba este test extraído del [Quick Start de ScalaTest](#)?

```
val emptyStack = new Stack[Int]
a [NoSuchElementException] should be thrownBy {
  emptyStack.pop()
}
```

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](#) (obsoleto): **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

5. Características de Programación Genérica

Características de PG

Objetivos

- Conocer los parámetros tipo, para poder construir código genérico
- Tener nociones sobre varianza y su implicación en la documentación de apis
- Empezar a trastear con la api de colecciones

Azúcar Sintáctico

- ❖ Clases Genéricas
- ❖ Métodos Polimórficos
- ❖ Varianza
- ❖ API de Colecciones

Características de PG

Clases Genéricas (1/2)



```
sealed trait Lista
```

```
case class Cons(  
  cabeza: Int,  
  resto: Lista = Nada()) extends Lista
```

```
case class Nada() extends Lista
```

Características de PG

Clases Genéricas (2/2)



```
sealed trait Lista[A]
```

```
case class Cons[A] (
```

```
  cabeza: A,
```

```
  resto: Lista[A] = Nada[A]() extends Lista[A]
```

```
case class Nada[A]() extends Lista[A]
```

Características de PG

Métodos Polimórficos (1/2)



```
object Lista {  
  def apply(is: Int*): Lista = {  
    if (is.isEmpty)  
      Nada()  
    else  
      Cons(is.head, apply(is.tail: _*))  
  }  
}
```

Características de PG

Métodos Polimórficos (2/2)



```
object Lista {  
  def apply[A](as: A*): Lista[A] = {  
    if (as.isEmpty)  
      Nada()  
    else  
      Cons(as.head, apply(as.tail: _*))  
  }  
}
```

Características de PG

Varianza (1/4)

- Invarianza (por defecto)
 - `class Lista[A]`
 - Las relaciones de herencia del parámetro tipo **no afectan** a las relaciones de herencia de la clase genérica
- Covarianza
 - `class Lista[+A]`
 - Si `A >::> B` entonces `Lista[A] >::> Lista[B]`
 - `Lista[Fruit] >::> Lista[Apple]`
- Contravarianza
 - `class Lista[-A]`
 - Si `A >::> B` entonces `Lista[A] <::< Lista[B]`
 - `Lista[Fruit] <::< Lista[Apple]`

Características de PG

Varianza (2/4)



```
trait Funcion[?A, ?B] {  
  def apply(a: A): B  
}
```

Características de PG

Varianza (2/4)



```
trait Funcion[-A, +B] {  
  def apply(a: A): B  
}
```


Características de PG

Varianza (3/4)



```
sealed trait Lista[+A] {  
  def contains(a1: A): Boolean = // ...  
}
```

“covariant type A occurs in contravariant position in type A of value a1”

Características de PG

Varianza (4/4)



```
sealed trait Lista[+A] {  
  def contains[A1 >: A](a1: A1): Boolean = // ...  
}
```



Características de PG

Immutable List API (1/2)



```
def reduce[A1 >: A](op: (A1, A1) => A1): A1
```

Reduces the elements of this traversable or iterator using the specified associative binary operator.

Características de PG

Immutable List API (2/2)

The screenshot shows the Scala Standard Library API documentation for the `scala.collection.immutable.List` class. The left sidebar displays a tree of Scala standard library packages, with `scala.collection.immutable` selected. The main content area is titled `List` and includes the following information:

- Related Docs:** [object List](#) | [package immutable](#)
- sealed abstract class `List[+A]`** extends [AbstractSeq\[A\]](#) with [LinearSeq\[A\]](#) with [Product](#) with [GenericTraversableTemplate\[A, List\]](#) with [LinearSeqOptimized\[A, List\[A\]\]](#) with [java.io.Serializable](#)
- Description:** A class for immutable linked lists representing ordered collections of elements of type. This class comes with two implementing case classes `scala.Nil` and `scala.List` that implement the abstract members `isEmpty`, `head` and `tail`. This class is optimal for last-in-first-out (LIFO), stack-like access patterns. If you need another access pattern, for example, random access or FIFO, consider using a collection more suited to this than `List`.
- Annotations:** `@SerialVersionUID()`
- Source:** [List.scala](#)
- Example:**

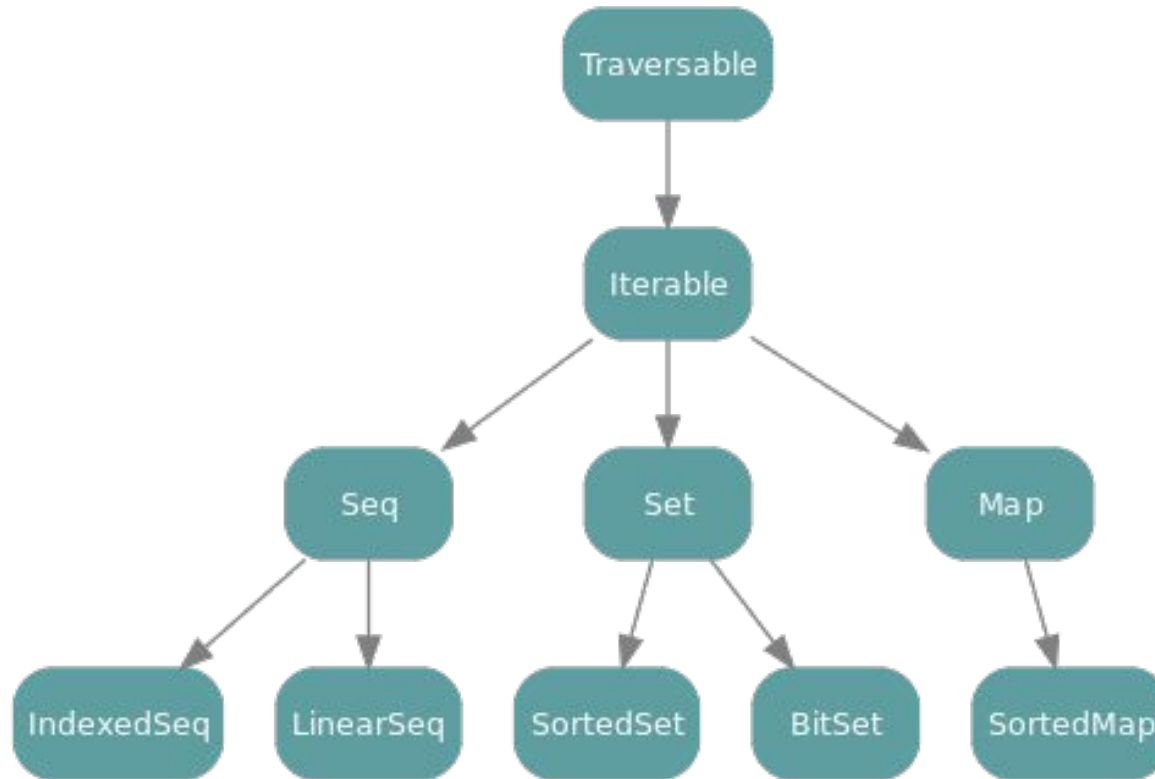
```
// Make a list via the companion object factory
val days = List("Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday")
// Make a list element-by-element
val when = "AM" :: "PM" :: List()
// Pattern match
days match {
  case firstDay :: otherDays =>
    println("The first day of the week is: " + firstDay)
  case List() =>
    println("There don't seem to be any week days.")
}
```
- Performance:**
 - Time:** `List` has $O(1)$ prepend and head/tail access. Most other operations are $O(n)$ on the number of elements in the list. This includes the index-based lookup of elements, `length`, `append` and `reverse`.
 - Space:** `List` implements **structural sharing** of the tail list. This means that many operations are either zero- or constant-memory cost.
- Example of structural sharing:**

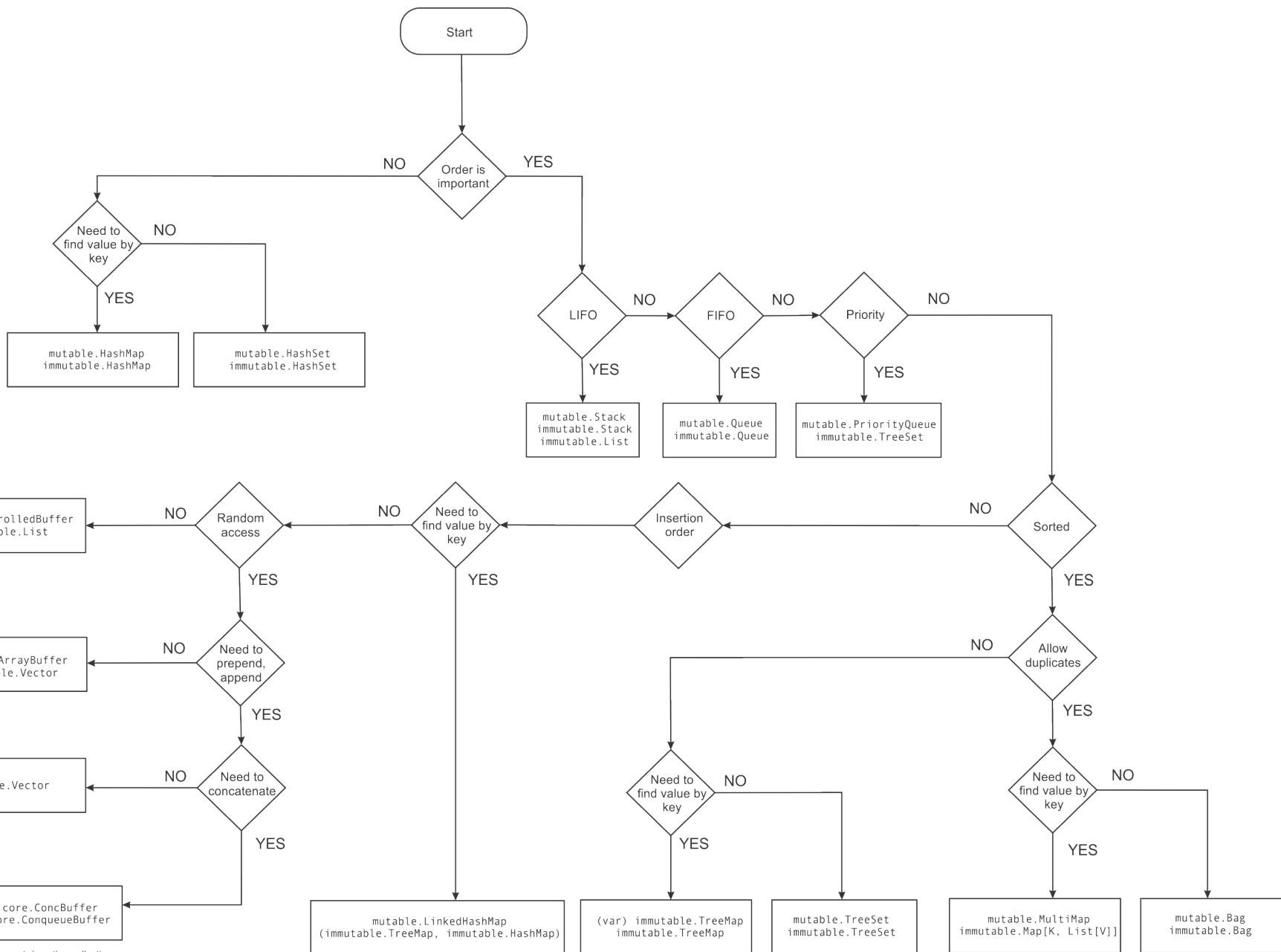
```
val mainList = List(3, 2, 1)
val with4 = 4 :: mainList // re-uses mainList, costs one :: instance
val with42 = 42 :: mainList // also re-uses mainList, cost one :: instance
val shorter = mainList.tail // costs nothing as it uses the same 2::1::Nil instances as mainList
```
- Version:** 2.8
- Since:** 1.0
- Note:** The functional list is characterized by persistence and structural sharing, thus offering considerable performance and space consumption benefits in some scenarios if used correctly. However, note that objects having multiple references into the same functional list (that is, objects that rely on structural sharing), will be serialized and deserialized with multiple lists, one for each reference to it. I.e. structural sharing is lost after serialization/deserialization.

At the bottom, a link points to the [Collection Library overview](#) section on Lists for more information.

Características de PG

API de Colecciones





<https://github.com/storm-enroute/reactive-collections>

<http://stackoverflow.com/questions/9313866/>

<https://github.com/nicolasstucki/multisets>



Características de PG

Ejercicios

1. Dada una lista de `Bicicleta`s predeterminada (l1 en `Ejercicios.scala`). Filtra todas aquellas bicicletas que se encuentren en una marcha impar.
2. Reutilizando la salida del ejercicio anterior, busca la bicicleta que circule a mayor velocidad. Pista: utiliza el método `reduce` para llevar a cabo la implementación.
3. Utilizando la lista inicial, suma las cadencias de todas las bicicletas. Pista: utiliza el método `foldLeft` para llevar a cabo la implementación.
4. ¿A qué estructura de control, muy frecuente en el paradigma imperativo, te recuerda `foldLeft`?

Características de PG

Takeaways y cómo seguir

- Takeaways
 - La genericidad en Scala es muy potente, sólo hemos visto pinceladas de su uso
 - Es muy recomendable identificar comportamientos recurrentes en nuestro código para generar abstracciones de alto nivel, procurando no reinventar la rueda
 - Scala promueve un estilo de interfaces con multitud de métodos de tamaño muy reducido
- ¿Por dónde seguir?
 - ¿[Qué colección](#) debería elegir?
 - Estudiar otras colecciones: Option, Map, Set, etc.
 - Estudiar aspectos más avanzados sobre tipos en Scala: Type Alias, Type Constructors, Type Bounds, etc.
 - Librerías de programación genérica (type-level): shapeless



Curso de Introducción a Scala

6. Conclusiones

Conclusiones

- *Scala as a better Java*
- Pero la verdadera ganancia de utilizar Scala **no reside en el paradigma OO**
- Scala contiene multitud de features, lo que lo convierte en un lenguaje complejo, pero **muy flexible** (*escala* con las necesidades de los usuarios)
- La **genericidad** y las técnicas propias del **paradigma funcional** dotan al programador de superpoderes
- El elevado coste asociado al aprendizaje de este lenguaje **merece la pena**



Habla Computing
info@hablapps.com
@hablapps

Scala Programming @ Madrid
@madridscala

Jesús López González
jesus.lopez@hablapps.com
@jeslg