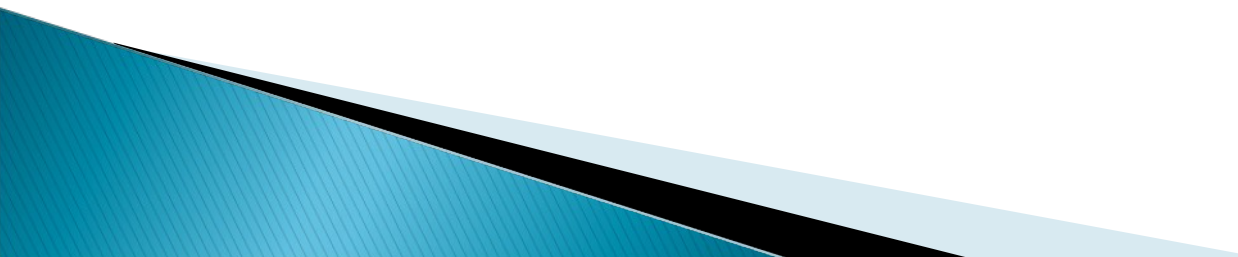





Agenda

- Introducción a Scala.
 - Features de Scala.
 - Closures.
 - Actores.
 - Inferencia.
 - Mucho más...
 - Funciona? Un aplicación de ejemplo.
 - Jugamos con el REPL: Read-Eval-Print-Loop.
 - Conclusión.
- 

Scala in a nutshell

- Corre sobre la JVM.
 - Interoperabilidad con Java en ambos sentidos. No hay que empezar desde cero.
 - Se pueden utilizar frameworks ya existentes implementados en Java.
 - Híbrido entre funcional y objetos.
 - Features interesantes, reduce el verbosity de Java, mejores defaults que Java (por ej: visibilidad).
 - Modelo de concurrencia no presente en Java => Actores.
 - "Intérprete" a lo Python o Ruby.
 - Mucho mas... :)
- 

Empecemos

- Tipado estático como Java. No así groovy, clojure, etc.
- Pero implementa inferencia de tipos:
 - Java: `final List<String> list = new ArrayList<String>();`
 - Scala: `val list = List[String]()`
- `val` es referencia inmutable (final de Java) y `var` es referencia mutable.
 - `var count = 1`
 - `count += 1 // ok!`
 - `val anotherCount = 1`
 - `anotherCount += 1 // fail!`

Sintaxis básica

- Punto y coma para finalizar una sentencia es opcional.
- Paréntesis y punto son opcionales en métodos de un solo parámetro.
- Tuplas y asignación multiple.
- Strings: como en java y multiline.
- return opcional (no recomendado).
- Las mayoría de las sentencias generan un valor de retorno:
 - `val myValue = if (1==2) 0 else 1`
- Clases public por default.

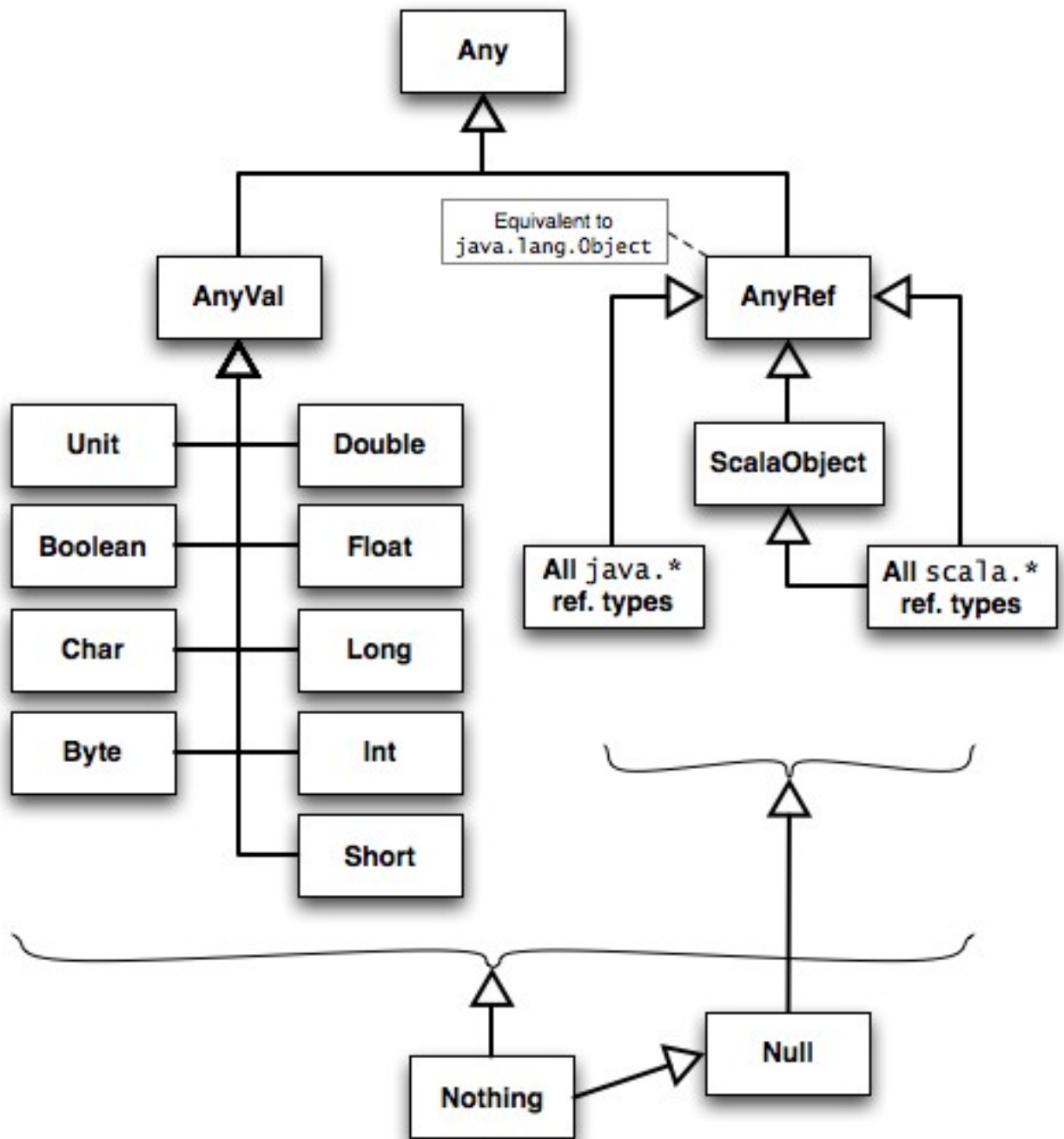
Colecciones

- Versiones mutables e inmutables. Default => inmutables
- Ejemplos:
 - `List(1,2,3,4,5,6).foreach { e => println(e) }`
 - `List(1,2,3,4,5,6).map { e => e * e }`
 - `List(5,10,15,20,30).partition { e => e < 15 }`
- Soporte para closures: funciones anónimas definidas inline.
 - `val f = (e: Int) => { e * e }`
- Tuplas.
 - `val tuple = ("my name", 3)`

Inferencia de Tipos

- Scala posee inferencia de tipos, reduce la cantidad de código a escribir.
- ```
class Math {
 def divideBy(a: Int, b: Int) =
 if (b == 0) throw new IllegalArgumentException
 else a / b
}
```
- El tipo de retorno es inferido a Int.
- Cómo funciona la inferencia? Es necesario entender la jerarquía de clases primero.

# Jerarquía de clases en Scala



All the types are in the `scala` package unless otherwise indicated.



# Inferencia de Tipos (II)

- ```
class Math {  
    def divideBy(a: Int, b: Int) =  
        if (b == 0) throw new IllegalArgumentException  
        else a / b  
}
```
- La rama del if devuelve Nothing, la rama del else devuelve Int. El tipo en común entre los dos es Int, entonces divideBy devuelve Int.

Classes

- ```
class Complex(val real: Int, val imaginary: Int) {
 def +(o: Complex) =
 new Complex(real + o.real, imaginary + o.imaginary)

 override def toString() =
 real + (if (imaginary < 0) "" else "+") + imaginary + "i"
}
```
- Tiene "sobrecarga de operadores":  

```
val a = new Complex(3,4)
val b = new Complex(4,5)
a + b // igual a.+(b)
```
- ```
res: Complex = 7+9i
```

Pattern matching

- Permite matchear objetos y ejecutar código dependiendo de los tipos.
- Útil al codear actores.
- Permite agregar guardas para determinar si se matchea una alternativa o no:
- ```
def activity(day: String) = {
 day match {
 case "lunes" => "codear"
 case "martes" => "refactorizar"
 }
}
```
- ```
activity("lunes")  
res: java.lang.String = codear
```

Pattern matching (II)

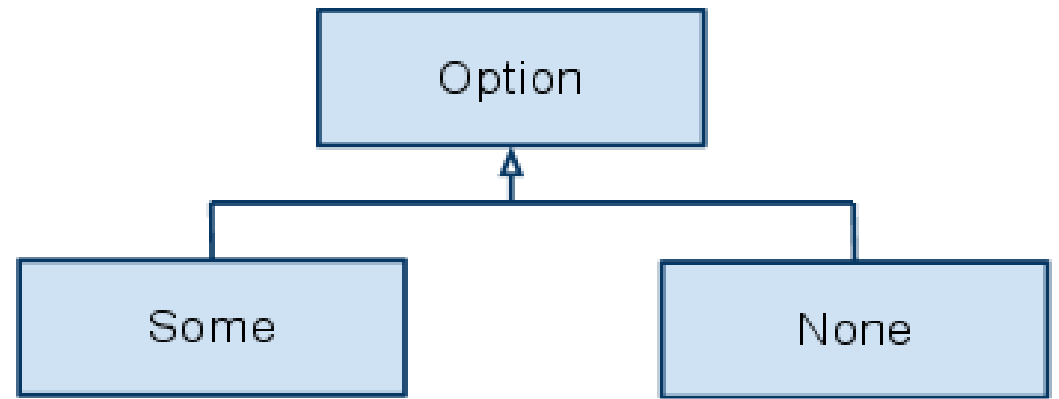
- ```
def activity(day: String, hour: Int) = {
 day match {
 case "lunes" if hour >= 9 && hour < 18 => "codear"
 case "lunes" => "descansar"
 }
}
```
- ```
activity("lunes", 9)  
res: java.lang.String = codear
```
- ```
activity("lunes", 19)
res: java.lang.String = descansar
```

# Funciones parcialmente aplicadas

- Las funciones pueden ser llamadas con una cantidad de parámetros menor a la requerida.
- Esto devuelve un function value con los parametros especificados bindeados.
- Luego se puede realizar la llamada en otro momento especificando los parámetros restantes.
- ```
import java.util.Date  
def log(now: Date)(msg: String) =  
    println("[%s] %s".format(now, msg))
```

```
val logNow = log(new Date) _  
logNow("hello")  
logNow("world")
```

Options




- Utilizado en circunstancias en las cuales una referencia devuelta puede ser Null.
- ```
class Document(val title:String)

def getTitle(doc: Document) =
 if (doc.title == null) None else Some(doc.title)

val doc = new Document(null)
val title = getTitle(doc)
```
- ```
val t = title.getOrElse("Sin Titulo").toUpperCase
```
- `title.isDefined` // indica si el valor no es null
- `title.isEmpty` // indica si el valor es null

Actores

- Un modelo de concurrencia que abstrae al dev de locks, mutexs, etc y permite hacer aplicaciones de alto desempeño.
 - Los 'actores' son entidades activas que se comunican con otros actores mediante el paso de mensajes.
 - Los mensajes generalmente son inmutables => reducen los problemas de sincronización.
 - Los actores pueden ser scheduleados usando diferentes estrategias:
 - Un actor en cada thread.
 - Actores compartiendo un pool de threads.
 - Implementación custom.
- 

Actores (II)

- El paso de mensajes es en gral asincrónico (aunque pueden ser sincrónicos).
- ```
import scala.actors.Actor._
val printer = actor {
 loop {
 receive {
 case v: Int => println("received " + v)
 }
 }
}
```
- `printer ! 4`



# Actores y Futures

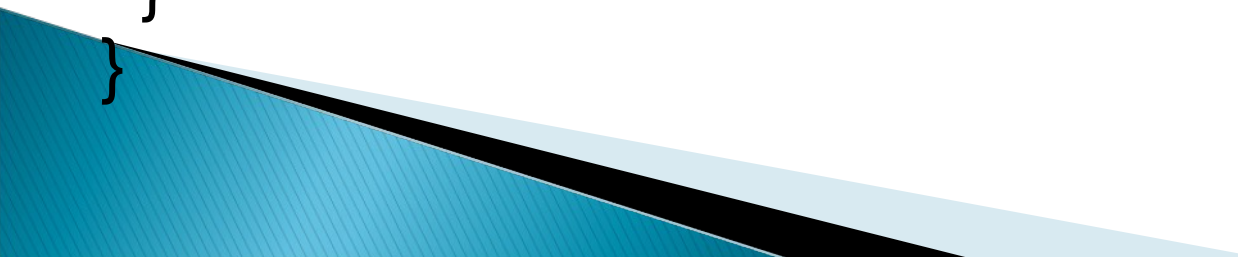
- case class Sum(val left: Int, val right: Int)  
case class Product(val left: Int, val right: Int)
- ```
val mathematician = actor {  
  loop {  
    receive {  
      case Sum(left, right) => reply(left+right)  
      case Product(left, right) => reply(left*right)  
    }  
  }  
}
```
- mathematician ! Sum(3,4)
 - mathematician !! Product(3,4)

Traits

- Atacan aspectos que atraviesan transversalmente la jerarquía de clases, ej Logging, seguridad, transacciones, etc.
- Como interfaces... pero pueden tener implementación.
- Única restricción: constructor sin parámetros.

```
trait Loggable {  
  def log(level: Int, message: String) =  
    println("[level %d] %s".format(level, message))  
}
```

```
class MyActor extends Actor with Loggable {  
  def act = {  
    log(level=5, "starting actor...")  
  }  
}
```



Traits (II)

- Podemos cambiar cómo se implementa el logger:
- ```
trait BetterLoggable extends Loggable {
 override def log(level: Int, message: String) =
 println("a better log message: " + message)
}
```

```
val actor = new MyActor with BetterLoggable
actor start
```

```
// wow! :)
```



# Traits (III) – Patrón Decorator

```
class Window{
 def draw = "ventana"
}
trait WBorder extends Window{
 override def draw = super.draw + " con borde"
}
trait WTitleBar extends Window{
 override def draw = super.draw + " con barra de titulo"
}
trait WMenuBar extends Window{
 override def draw = super.draw + " con barra de menu"
}
```

```
val w1 = new Window with WBorder with WMenuBar
w1 draw // ventana con borde con barra de menu
val w2 = new Window with WTitleBar with WMenuBar with WBorder
w2 draw // ventana con barra de titulo con barra de menu con borde
```

# Implicits

- Patrón: "Pimp my library": no podemos cambiar código de bibliotecas de sw existentes.

- Ejemplo:

```
// <código cerrado>
```

```
trait Callback {
 def onClick: Unit
}
```

```
class Button(val callback: Callback) {
 def click = callback.onClick
}
```

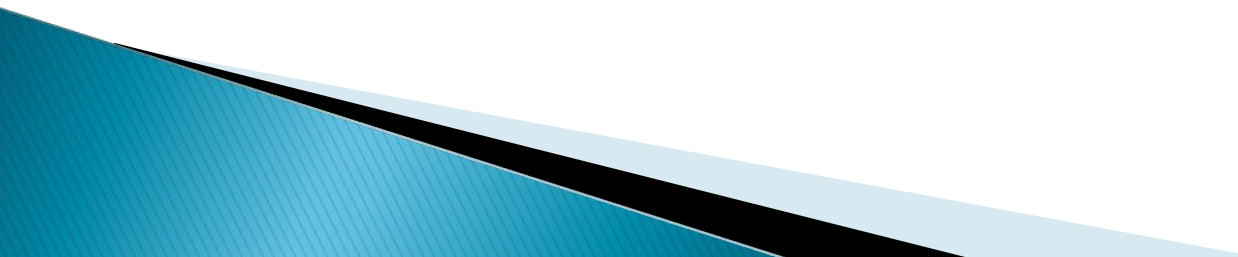
```
// </codigo cerrado>
```

- Ya somos expertos en funcional, no queremos clases anónimas por todos lados... qué hacemos?

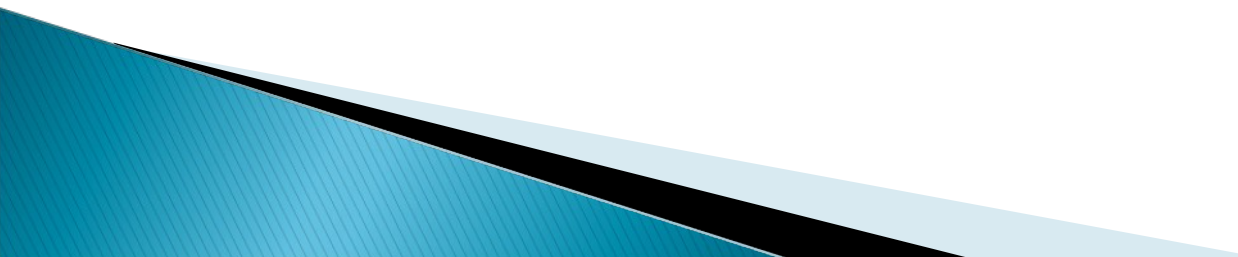
# Implicits (II)

- `// definimos una clase que recibe un closure por parámetro`
- `class ClosureCallback(val f: () => Unit) extends Callback {  
 override def onClick = f()  
}`
- `// una función mas, marcada para aplicación implícita`
- `implicit def convertClosureToClosureCallback  
 (f: () => Unit) = new ClosureCallback(f)`
- `// probemos :)`
- `new Button( () => println("hola") )`

# Parallel Editor

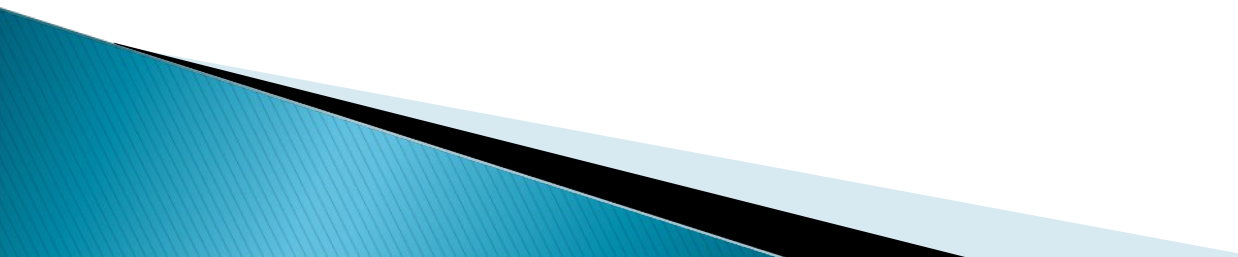
- Demostración de que funciona.
  - Trabajo Profesional.
  - Uso intensivo de actores.
  - Integración con Java.
- 

# Para cerrar

- No todo es color de rosa:
    - Inconpatibilidad binaria entre releases del lenguaje.
    - Compilación mas lenta que en Java.
    - Herramientas no tan desarrolladas (ide, building).
  - Lo bueno:
    - Código de alta densidad.
    - No hay que empezar de cero. Aprovecha libs de java.
    - Conceptos y features interesantes.
    - Modelo de concurrencia "nuevo" (ahora en Java a través de Akka).
    - Mix de objetos y funcional.
- 



# Referencias

- Scala: <http://www.scala-lang.org>
  - Scala-Ide (Eclipse): <http://www.scala-ide.org>
  - Programming Scala (Tackling Multi-Core complexity on the JVM) - Venkat Subramaniam
  - Programming Scala (O'Reilly)
  - Actors in Scala: [www.artima.com/shop/actors\\_in\\_scala](http://www.artima.com/shop/actors_in_scala)
  - Parallel Editor: [github.com/maurocianscio/parallel-editor](https://github.com/maurocianscio/parallel-editor)
- 



**Preguntas?**

# Gracias!

Mauro Ciano [@maurociao](#)  
Leandro Gilioli [@legilioli](#)