Lenguaje Scala

Introducción I

- Scala fue creado por Martin Odersky y colaboradores
- En 2001 se toma la decisión de crear un Java mejor
- En 2003 aparece la primera versión (experimental) de Scala
- En 2005 ve la luz Scala 2.0, escrito en Scala
- En 2011 se libera Scala 2.9 y se funda la compañía Lighbend
- En 2017 usamos Scala 2.12.X

Introducción II

- Scala es un lenguaje cuyos programas se ejecutan en la JVM
- Tiene un compilador (scalac) y un intérprete (scala)
  - Ver en el repo el ejemplo apoyo/apoyo0002
- Sistema de tipos fuerte, como Java, pero con inferencia automática de tipos
- En Scala todo es un objeto (3 + 2 equivale a 3.+(2))
- Está totalmente integrado con Java
- Es conciso, orientado a objetos y funcional
- Ver en el repo el ejemplo apoyo/apoyo0001

Primeros pasos: El intérprete

- También llamado REPL (**R**ead **E**val **P**rint **L**oop)
- Se le invoca desde la línea de comandos tecleando scala
- También desde tecleando sbt console, o activator console
- Compila y evalúa código Scala inmediatamente
- Útil para experimentar y probar rápidamente
- Ver en el repo el ejemplo apoyo/apoyo0003/repl.txt y comandos.txt

Primeros pasos: Valores inmutables y mutables

- Un valor inmutable se define mediante la palabra clave val
  - val texto = “uno”
  - texto = “dos” → provoca un error de compilación
- Un programa Scala intentará usar vals siempre que sea posible
- Un valor mutable se define mediante la palabra clave var
  - var texto = “uno”
  - texto = “dos” → legal

Primeros pasos: Inferencia de tipos

- Scala es capaz de detectar (no siempre) los tipos de los objetos que manejamos
  - val texto = “uno” → el intérprete responde con texto: String = uno
  - val texto:String = “uno” → la misma respuesta
  - val texto:String = 7 → provoca un error de compilación. Un Int no es una String
- Hay que guardar un equilibrio entre aprovechar la inferencia de tipos y la claridad y mantenibilidad del código que escribimos

Primeros pasos: Expresiones

- En Scala, la mayoría de sus constructos son expresiones y no sentencias
- Un if-else en Java, por ejemplo, evalúa su cuerpo pero no devuelve nada como tal. Es una sentencia: String x; if(a==b){ x = “uno” } else { x = “dos” }
- Un if-else en Scala sí. Es una expresión: if(a==b){ “uno” } else { “dos” }
- El valor del if-else sería “uno” ó “dos”
- Lo mismo pasaría con un bloque de código o un try-catch

Primeros pasos: Bloques de código

- Los bloques son expresiones, devuelven un resultado: la última línea del bloque

- scala> val test = {
- | val uno = 1
- | val dos = 2
- | uno + dos
- | }
- test: Int = 3

Primeros pasos: Concisión I

```
val test  =
  if ("Scala" startsWith "S") {
    val scala = "Scala"
    val es = "es"
    val conciso = "conciso"
    scala + " " + es + " " + conciso
  } else
    "Algo raro ha sucedido"
```

Primeros pasos: Concisión II

- No hace falta usar {} para expresiones de una sola línea
- Los tipos pueden omitirse
- Los “.” y los () también: “Scala” startsWidth “S”, no “Scala”.startWidth(“S”)
- Los “;” de fin de sentencia no son obligatorios
- La sentencia return tampoco

Orientación a objetos

- Scala soporta todas las nociones fundamentales de orientación a objetos
- Ver en el repo el ejemplo apoyo/apoyo0004
- Clases: por defecto, todo es público. Existen los modificadores *private* y *protected*
- Atributos para almacenar el estado de un objeto
- Métodos y modificadores de acceso
- Traits (características). Similares a las interfaces de Java
- Polimorfismo: redefinición de métodos y de “tipo”
- Herencia simple mediante la palabra clave *extends*
- Composición de clases mediante la adición de uno o varios *traits*, similar al *implements* de Java
- Objetos como tales, llamados *singletones*

Orientación a objetos: Paquetes

- Scala maneja los paquetes, en principio, como Java
- Sin embargo, hay novedades:
  - import uno.dos._
  - import uno.dos.{A,B}
  - import java.sql.{Date => SqlDate}
- En Scala, un import puede colocarse en muchos más lugares que en Java
  - def x() = {
  - import uno.dos.ClaseTres
  - val m = new ClaseTres
  - m.metodo()
  - }

Orientación a objetos: Pre y postcondiciones

- Scala tiene un objeto especial llamado Predef
- Entre otras cosas, podemos definir precondiciones (require) y postcondiciones (ensuring)
- Por ejemplo:

```
def addNaturals(nats: List[Int]): Int = {
  require(nats forall (_ >= 0), "List contains negative numbers")
  nats.foldLeft(0)(_ + _)
} ensuring(_ >= 0)
```

- Java llamaría a esto aserciones

Orientación a objetos: case clases

- Una *case class* se define de esta forma:
- case class Persona(nombre: String, edad: Int)
- Los parámetros son vals. Scala crea los métodos hashCode, equals, copy y toString
- Se crea automáticamente un companion object
- Para instanciar: val p = Persona(“abc”,20)
- Detrás del escenario, Scala ejecuta: Persona.apply(“abc”,20), en donde apply es un método del companion object
- Decimos que el companion es una factoría de objetos
- Las case clases se usan a menudo en reconocimiento de patrones
- Estas clases pesan más y no pueden ser ancestros de otra case class

Ejecutar una aplicación

- Desde sbt
  - sbt run
- Desde el activator de LIghtbend
  - activator run
- Desde el intérprete de Scala:
  - C:\Users\usuario\git\ftts_14_11_2016\apoyo\apoyo0004>scala -cp target\scala-2.12\classes Principal

Testing I

- En el mundo del desarrollo es Scala se piensa que es esencial crear baterías de pruebas
- Bien a medida que el software se escribe, bien empleando las técnicas de TDD y/o BDD
- Existen muchas herramientas con esta orientación capaces de interactuar con Scala:
  - ScalaTest, Specs2, ScalaCheck, JUnit, TestNG
  - Mocking frameworks: Mockito, ScalaMock
- Usaremos ScalaTest
- Ver en el repo apoyo/apoyo0005

Testing II

- El código de los tests es legible y mantenible
- Las descripciones de los actores y sus comportamientos están integrados en el test
- Los *Matchers* nos proporcionan mensajes de error claros
- Scalatest se integra bien con ScalaCheck y Mockito

Testing III

- Algunos ejemplos del uso de matchers
- value shouldEqual expected

  emptyCollection shouldBe 'empty

  emptyCollection should be('empty)

  collection should not be 'empty

  collection should have size 20

  collection should contain(value)

  boolean shouldBe true

  an[IllegalArgumentException] should be thrownBy expr

Testing IV

- Algunos ejemplos del uso de aserciones
- assert(value === expected)

  assert(emptyCollection.empty)

  assert(collection.empty === false)

  assert(collection.size === 20)

  assert(collection.contains(2) === true)

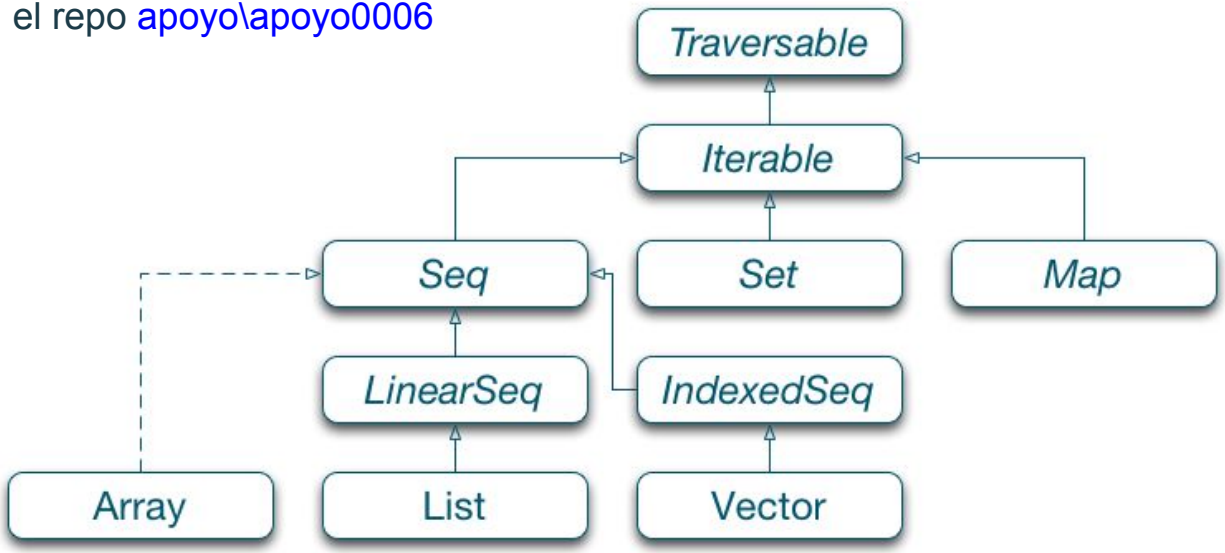
  intercept[IndexOutOfBoundsException] {

    collection.foo(badArgument)

  }

Colecciones y Programación funcional: Jerarquía

Ver en el repo apoyo\apoyo0006



Colecciones: Instanciación

- Cada clase de tipo colección tiene un *companion object*
- Esto permite declaraciones como:
  - Set(1,2,3) en lugar instanciar un nuevo conjunto usando el operador new
- Las colecciones tienen *type parameters*, por ejemplo:
  - Map[A,B]
- El compilador intentará inferir los tipos, pero pueden indicarse:
  - Set[Int](1,2,3)
- Las colecciones tienen algunos métodos típicos (ver en el repo apoyo\apoyo0006)

Colecciones: Mutables e inmutables

Las colecciones de Scala se organizan de la siguiente manera:

- Paquete *scala.collection*, la base común
- Paquete *scala.collection.immutable*
- Paquete *scala.collection.mutable*

Scala usa por defecto las colecciones inmutables

- Ver los alias empleados en *Predef*
- *Seq* se basa en *scala.collection.Seq* para permitirnos manejar los arrays eficientemente

Colecciones: Inmutables

- Una colección inmutable no se puede modificar en el estilo de Java (*in place*)
- Cada vez que *modificamos* una colección inmutable obtenemoms una nueva instancia
- Scala dice que sus colecciones inmutables son persistentes (comparten estructura) siempre que sea posible:
  - val a = List(1,2,3)
  - val b = a.tail
  - “a” apunta al comienzo de la lista, y “b” al segundo elemento. No se crea una nueva instancia
- A la hora de emplear colecciones es esencial conocer sus niveles de eficiencia en según qué operaciones

Programación funcional (fp) I: Premisa

- Premisa: construímos programas empleando exclusivamente funciones puras, esto es, carentes de efectos colaterales (*side effects*)
- Algunos ejemplos de efectos colaterales
  - Modificar una estructura de datos *in place*
  - Modificar un atributo en un objeto (los *setters* de Java) o una variable
  - Lanzar una excepción o detener un programa con un error sin tratar
  - Leer o escribir en cualquier tipo de dispositivo
- La fp define cómo creamos programas, no lo que estos pueden hacer
- Es posible escribir programas en un estilo funcional puro (Haskell)
- Scala no llega tan lejos (¡tiene variables!) pero integra la OO con la fp perfectamente

Programación funcional (fp) II: Funciones puras

- Una función pura, en Scala A => B, representa una computación que asocia un valor a de tipo A a exactamente un valor b de tipo B
- De tal modo que b se determina exclusivamente en función de a
- Así que dado el mismo valor de a, se obtiene el mismo valor de b
- En Scala, por ejemplo:
  - val funcionPura: Int => String = _.toString
  - La llamada a funcionPura(8) siempre devuelve “8” y, además,
  - realiza una computación sin *side effects*
- Una función pura no tiene efectos observables en la ejecución de un programa más que computar un resultado dados sus inputs

Programación funcional (fp) III: Transparencia referencial

- Podemos formalizar lo antedicho recurriendo a la noción de Transparencia referencial
- Aunque esta propiedad es, en rigor, característica de las expresiones, puede aplicarse a las funciones puras
- Por ejemplo: 5 + 6 es una expresión cuyo resultado proviene de aplicar la función pura “+”
- De hecho, podemos sustituir en todo el programa la expresión por su resultado, con la absoluta certeza de que son completamente equivalentes

Programación funcional (fp) IV: Funciones de alto nivel

Transparencia referencial y pureza

- Una expresión e es referencialmente transparente si para todos los programas p todas las ocurrencias de e en p pueden ser reemplazadas por el resultado de evaluar e, sin afectar el comportamiento observable de p
- Una función f es pura si la la expresión f (x) es referencialmente transparente para todo x, siendo x referencialmente transparente
- Scala permite usar funciones puras y funciones de alto nivel (funciones que retornan funciones y admiten como parámetros a otras funciones)

Bucles for

- Un bucle for en Scala es una sentencia, no una expresión, así que su cuerpo devuelve *Unit* (similar al *void* de Java). Ver en el repo apoyo\apoyo0007
- Está pensado para ejecutar efectos colaterales (imprimir en consola, cosas así)
- La sintaxis es for (secuencia) bloque
  - for (n <- 1 to 5) println(n)
- La secuencia puede contener generadores (generators), filtros (filters) y definiciones (definitions)

Expresiones for

- Scala transforma un bloque for en una expresión for mediante un mínimo cambio de sintaxis:
- for (secuencia) yield expresión
  - for (n <- 1 to 5) yield n
- La secuencia puede contener generadores (generators), filtros (filters) y definiciones (definitions)
- La expresión crea un elemento del resultado

Expresiones for: generators

- Una expresión generadora tiene la forma: elemento <- colección y dirige la iteración
- colección representa la estructura sobre la que vamos a iterar
- elemento representa una variable local ligado al elemento actual de la iteración
- El primer generador define el tipo del resultado
- Puede haber múltiples generadores

Expresiones for: filters

- Un filtro decide qué elementos proporcionados por una iteración van a tratarse: if expresión
- expresión debe ser booleana
- Sintácticamente, un filtro puede acompañar a un generador en la misma línea
    - for (secuencia if expresion) yield resultado

Expresiones for: definitions

- Las definiciones son simplemente variables locales
- Pueden ir acompañadas de un filtro
- Sintácticamente, por ejemplo:
  - for {

    elem <- colección

    definicion = expresión if expresión cumple condición

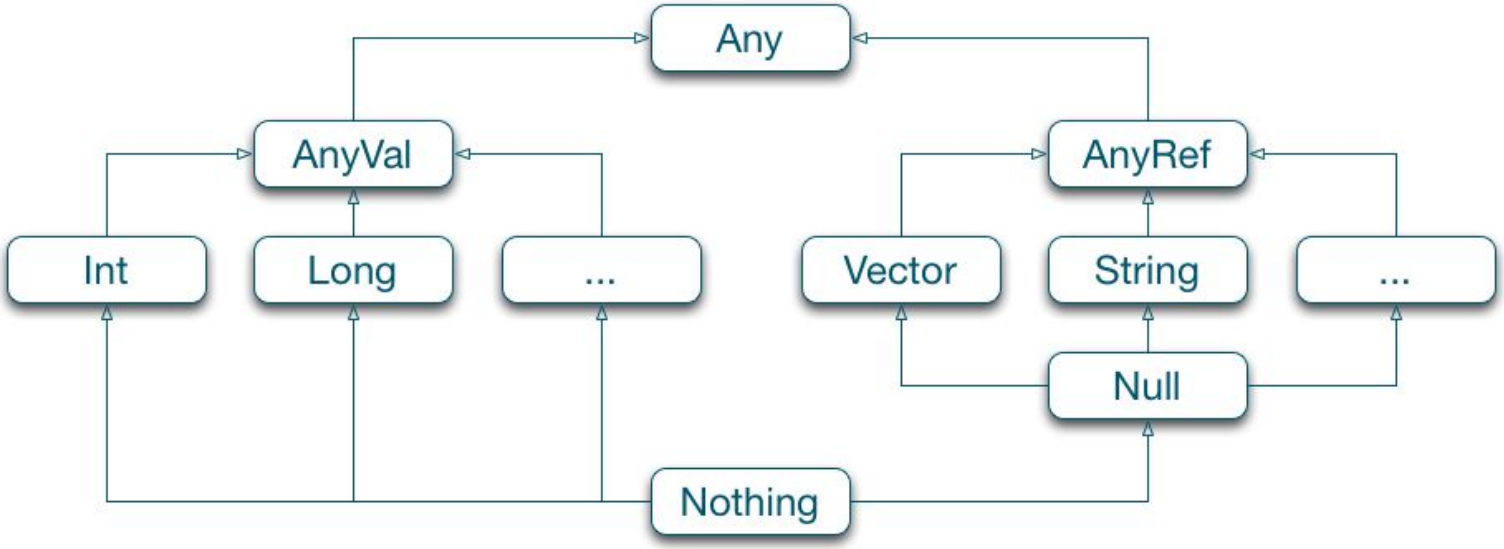
    } yield resultado involucrando la definición
- Los bucles y expresiones for son transformados automáticamente por el compilador

Herencia y características

Ver en el repo apoyo\apoyo0008

- Scala soporta herencia simple: un único ancestro
- La clase raíz de las que todas descienden automáticamente se llama Any
- Un descendiente hereda todos los miembros que no sean privados
- El lenguaje ofrece también polimorfismo de la forma habitual:
  - Los miembros que no sean finales se pueden sobreescribir
  - El principio de sustitución de Liskov se mantiene
- También tenemos el equivalente de los genéricos de Java

Herencia y características: Jerarquía de tipos



Herencia y características: traits

- La idea de un trait (característica) es la misma que la de una *interface* Java
- Simular la herencia múltiple sin caer en sus inconvenientes
- En Scala, un trait puede tener sólo un ancestro
- En un trait se puede definir métodos con código, además de otros artefactos
- Una clase puede definirse en términos de múltiples traits
- Un trait tiene como ancestro a la clase AnyRef
- Ejecutar el archivo mixin.scala desde dentro del intérprete (comando :load) y observar los resultados

Reconocimiento de patrones

- El *pattern matching* es una de las características más potentes en Scala
- Sintaxis mínima
  - expresión match {

    case patrón1 => resultado1 //Match pattern 1

    case patrón2 => resultado2 //Match pattern 2

    }
- Los diferentes casos (normalmente) retornan un valor. Son expresiones
- Si todos los casos posibles no están previstos, se produce un MatchError
- Ver en el repo apoyo\apoyo0009

Valores opcionales

- Se trata de evitar los *null*s y las *NPE* de Java
  - Cosas como mapa.get(clave); if(clave == null)...
- Scala ofrece una solución funcional, la Clase Option (un ADT)
- Option tiene dos posibles valores: Some (una *case class*) y None (un *singleton*)
- Ver en el repo apoyo\apoyo0010

Gestión de errores

- La forma de capturar excepciones en Java es mediante bloques try-catch
- En Scala, también podemos hacer lo mismo
- Sin embargo, hay una manera de gestionar errores más “funcional”, empleando la clase *Try*, preferiblemente, o la clase Either
- La clase Try no captura todas las excepciones. Solo las no fatales
- Ver en el repo apoyo\apoyo0011