# Scala - Functional & OO Programming Combined

## The Heir of Java?

Eivind Barstad Waaler

UiO

September 29, 2009

# Outline

# What is Scala?

- Multi-paradigm:
  - 100% Object-oriented
  - Functional programming
- Typing: static, strong, inferred/implicits
- Java bytecode → JVM
- Brief history:
  - 1995 – Pizza → GJ → javac & Java generics
  - 2001 – Scala design started by Martin Odersky (EPFL)
  - 2003 – Scala version 1.0
  - 2006 – Scala version 2.0
  - Today – Scala version 2.7.6 → 2.8

# A first example

```
class IntMath(val x: Int, val y: Int) {

  def sum(): Int = {
    x + y;
  }

  def mul = x * y
}

val test = new IntMath(3, 4)

println(test.x) // output: 3
println(test.sum) // output: 7
println(test.mul) // output: 12
```

# Method syntax

- `.` and `()` can be omitted – operator notation
- Methods can have any name
- Operators are simply methods $\rightarrow$ full operator overloading

```scala
2 + 5 // Operator + is just a method on the Int class
2.+(5) // same as above
2 max 5 // The max method in operator notation

class MyNumber(val num: Int) {
  def +(other: MyNumber) = new MyNumber(other.num + num)
}

val x = new MyNumber(5)
val y = new MyNumber(6)
val z = x + y // z is new MyNumber(11)
```

# Type inference/implicits

- If type is obvious – no need to write it
- Implicits can be defined
- Implicits heavily used in std lib (RichInt etc.)

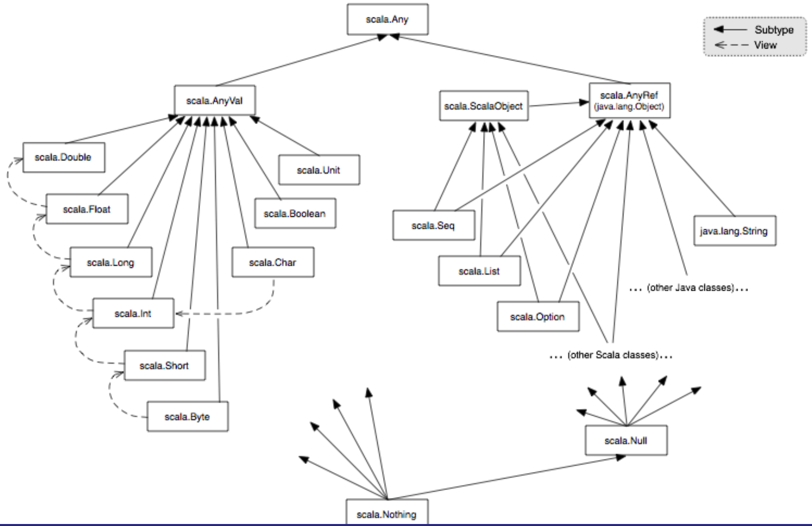
```
// Type inference
val a = 42 // Type = Int
val b = "hello world!" // Type = String
def add(x: Int, y: Int) = x + y // Return-type = Int
val sum = add(3, 5) // Type of sum = Int
def add(x: Int, y: Double) = x + y // Return-type = Double

// Implicits
class MyInt(val i: Int) { def doubleIt = i * 2 }
implicit def fromInt(i: Int) = new MyInt(i)
5.doubleIt // doubleIt called on Int
```

# Object Orientation

- Pure OO – everything is an object
- Classes – blueprints for objects (like Java)
- Singleton objects
- Traits – AOP like possibilities

# Scala Class Hierarchy



# Classes

- Much like Java
- Contains fields and methods – can override each other
  - override keyword mandatory
- Can take parameters directly – constructor

```
class A(val num: Int)
class B(num: Int, val str: String) extends A(num) {
  def calc() = num * num // calc is a method
}
class C(num: Int, str: String) extends B(num, str) {
  override val calc = 65 // override method with val
}

val a = new A(35) // Type A
val b = new B(32, "Eivind") // Type B - calc method
val b2: B = new C(32, "Eivind") // Also type B - calc val
```

# Singleton Objects

- No static members in Scala → Singleton Objects
- Keyword `object` instead of `class`
- Companion class/object – same name
  - Factory methods
  - Other unique/static behaviour

```
// Array class definition
final class Array[A](_length: Int) extends Array0[A]

// Array object definition with method
object Array {
  def apply(xs: Int*): Array[Int] = { ... }
}

// Create array with four integers
val arr = Array(1, 2, 3, 4)
```

# The "magic" apply()-method

- Scala provides special syntax for calling the `apply()` method
- In classes used to look up elements (for instance in Collections)
- In objects used to create class instances
- Functions in Scala are actually just `apply()` methods
- Make your classes/objects appear as built-in syntax

```
// Array object has apply method for creating arrays
def apply(xs: Int*): Array[Int] = { ... }
// Array class has apply method to access the array
def apply(i: Int): A

// Scala special syntax
val arr = Array(4, 3, 2, 1) // Array.apply(4, 3, 2, 1)
val three = arr(1) // arr.apply(1)
```

# Traits

- Encapsulates method and field definitions, much like classes
- A class can mix in any number of traits $\rightarrow$ multiple inheritance
- Widen thin interfaces to rich ones
- Define stackable modifications

```
trait Hello {
  def hello { println("Hello!") }
}

trait Goodbye {
  def bye { println("Bye bye!") }
}

// Object with both hello() and bye() methods..
object A extends Hello with Goodbye
```

# Traits – Widen thin interface to rich

- Define one or a few abstract methods
- Define concrete methods implemented in terms of the abstract

```
trait Ordered[A] {
  abstract def compare(that: A): Int
  def <(that: A): Boolean = this.compare(that) < 0
  def <=(that: A): Boolean = this.compare(that) <= 0
  ...
}

class Name(val name: String) extends Ordered[Name] {
  def compare(that: Name) = this.name.compare(that.name)
}

if(name1 <= name2) { ... // val name1 = new Name("Ola")
```

# Traits – Define stackable modifications

- Modify methods of a class
- Stack several modifications with each other

```
abstract class IntQueue {
  def get: Int
  def put(x: Int)
} // + concrete impl IntQueueImpl

trait PutPrint extends IntQueue {
  abstract override def put(x: Int) {
    println("Put: " + x)
    super.put(x)
  }
}

val printQueue = new IntQueueImpl with PutPrint
```

# Generics

- Classes and traits can be generified
- Generics/type parameterization impl with erasure (like Java)
- Type parameters are required (not like Java)
- Variance – nonvariant, covariant and contravariant
- Upper and lower bounds

```
trait Set[T] { // Nonvariant
  def contains(elem: T): Boolean
  ...
trait Set[+T] // Covariant
trait Set[-T] // Contravariant

trait OrderedSet[T <: Ordered[T]] // Upper bound
trait Array[+T] {
  def indexOf[S >: T](elem: S): S // Lower bound
```

# Abstract types

- Types as abstract members
- Much like type parameterization, different usage
- Generic types – reusable containers, collections ++
- Abstract types – premade subclasses, hides implementation

```
abstract class ValSet {
  type DType <: AnyVal // Abstract type
  def put(elem: DType) = ... // Define methods
}

class IntSet extends ValSet {
  type DType = Int // Concrete override of type
}
// Anonymous instance
val dSet = new ValSet { type DType = Double }
```

# Functional programming

- Scala goal: Mix OO and FP
- Some FP characteristics:
  - Higher-order functions
  - Function closure support
  - Recursion as flow control
  - Pure functions – no side-effects
  - Pattern matching
  - Type inference/implicits
- Good fit for concurrent or distributed programming

# Mutable/immutable

- Immutable data structures important in FP
- Pure function – same result with same arguments
- Scala uses keywords `var` and `val`
- Immutable definitions (`val`) encouraged

```
val num = 45 // Immutable - allways 45
num = 60 // Error: reassignment to val

var num2 = 45 // Mutable - can be changed
num2 = 60 // Ok
```

# Scala functions

- Higher-order functions – args and results
- Objects that implement `scala.FunctionN` traits
- Special syntax support with => operator

```
// Three equivalent definitions to find even numbers
val f1 = new Function[Int, Boolean] { // Full version
  def apply(i: Int) = i % 2 == 0
}
val f2 = (i: Int) => i % 2 == 0 // Special operator
def f3(i: Int) = i % 2 == 0 // Regular function definition

// Usage (f1, f2 and f3 equivalent)
f1(64) // Converts to f1.apply(64)
val arr = Array(1, 2, 3, 4)
arr.filter(f1) // Returns Array(2, 4)
```

# Closures/anonymous functions

- Scala – Anonymous functions
- Like anonymous classes in Java (and Scala)
- Nothing special with closures – just passing function object
- The _ (underscore) can be used to anonymize arguments

```
val arr = Array(1, 2, 3, 4)

arr.filter((i: Int) => i % 2 == 0) // Returns Array(2, 4)
arr.filter(_ % 2 == 0) // Shorter version using _

arr.map(_ % 2) // Returns Array(1, 0, 1, 0)

arr.foreach(print _) // Prints "1234"
```

# Partially applied functions and currying

- Partially applied functions – leave args out
- Currying – multiple argument lists

```
// Partially applied function
def sum(i: Int, j: Int) = i + j
val fivePlus = sum(5, _: Int) // New func with 1 arg
fivePlus(6) // Result 11

val myprint = print _ // Example from previous slide
myprint("hello world!")

// Currying example
def curriedSum(i: Int)(j: Int) = i + j
curriedSum(2)(3) // Result 5
val fivePlus = curriedSum(5)_ // New func with 1 arg
fivePlus(6) // Result 11
```

# Pattern matching

- Like `switch` statements on steroids
- Kinds of patterns:
  - Wildcard patterns – the _ char again
  - Constant patterns – numbers, strings ++
  - Variable patterns – names
  - Constructor patterns – `case` classes
  - Sequence patterns – all sequence classes (List, Array ++)
  - Tuple patterns – all tuples
  - Typed patterns – like function arguments
- Variable binding – using the `@` sign
- Pattern guards – adding an `if` clause
- Pattern overlaps – case order is important!

# Pattern matching – Basic example

```
def desc(x: Any) = x match {
  case 5 => "five" // Constant pattern
  case i: Int => "int: " + i.toString // Typed patterns
  case s: String => "str: " + s
  case (a, b) => "tuple: " + a + b // Tuple pattern
  case _ => "unknown" // Wildcard/default pattern
}

desc(8) // "int: 8"
desc("Scala") // "str: Scala"
desc(5) // "five"
desc(("Eivind", 32)) // "tuple: Eivind32"
desc(4.0) // "unknown"
```

# Pattern matching – List example

```
def desc(x: Any) = x match {
  case List(_: String, _: String) => "List of two strings"
  case List(_, _) => "List of two elems"
  case head :: _ => "List starting with " + head
  case _ => "Whatever"
}

desc(List(1, 2)) // List of two elems
desc(List(1, 2, 3)) // List starting with 1
desc(List("hello", "world")) // List of two strings

// Note! Two equivalent defs - "Error: unreachable code"
case head :: _ => "List starts with " + head
case List(head, _*) => "List starts with " + head
```

# For expressions

- Generators, definitions and filters
- Can yield value – `Range` class
- A rewrite of methods `map`, `flatMap` and `filter` – monads

```
for (l <- "letters") println(l) // Split with line
for (num <- 1 until 10) print(num) // Prints "123456789"

for {
  p <- persons // Generator
  n = p.name // Definition
  if(n startsWith "E") // Filter
} yield n

// Two generators - with ( and ; to compact
for (n <- 1 to 3; l <- "xyz") yield n.toString + l
```

# Actors api

- Simplify concurrent programming
- Hides threads – message based
- Immutable objects and functional style recommended!

```
import scala.actors.Actor._
val helloActor = actor {
  while (true) {
    receive {
      case msg => println("hello message: " + msg)
    }
  }
}
helloActor ! "Hello World!!"
```

# Parallel Computing with Futures

```
def parallel[T](obj: Splittable[T], op: (T) =>
    Splittable[T]): Splittable[T] = {
  obj.split match {
    case Array(region) => op(region)
    case regions: Array[T] => {
      val futures = for(region <- regions) yield future {
        op(region)
      }
      val results = awaitAll(5000, futures: _*)
      val parts = for(result <- results) yield
          result.get.asInstanceOf[Splittable[T]]
      parts.reduceLeft(_ merge _)
    }
  }
}
```

# Continuations

- Coming in Scala 2.8
- Typical tasks:
  - Asynchronous I/O with Java NIO
  - Executors and thread pools
  - Cross-request control flow in web applications
- New library functions, not keywords – `shift` and `reset`
- Complicated – different mindset

```
reset {
  shift { k: (Int => Int) =>
    k(k(k(7)))
  } + 1
} * 2 // result: 20
```

# Conclusion

- Object orientation and functional programming combined
- Static and compiled
- Rich syntax – Java-code / 3?
- Seamless integration with Java – run in existing environment
- Big momentum – the Heir of Java?

**More Info:**

http://www.scala-lang.org/