Introduction to Scala

The Heir of Java?

Eivind Barstad Waaler

BEKK/UiO

October 1, 2009

Outline

What is Scala?

- Multi-paradigm:
  - 100% Object-oriented
  - Functional programming
- Typing: static, strong, inferred/implicits
- Java bytecode → JVM
  - .NET (CLR) Also available
- 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 mul = x * y
}
```

Scala ⇑ — Java ⇓

```
class IntMath() {
  private int x, y;
  public IntMath(int x, int y) {
    this.x = x; this.y = y;
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public int mul() { return x * y; }
}
```

Methods and operators

- . and () can be omitted – operator notation
- Methods can have any name
- Operators are simply methods → full operator overloading
- Java has no operator overloading

```
2 + 5 // Operator notation - same as Java
2.+(5) // Regular method notation
2 max 5 // Operator notation
2.max(5) // Regular notation
```

Object Orientation

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

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
}
```

Singleton Objects

- No static members in Scala → Singleton Objects
- Keyword `object` instead of `class`

```
class IntPair(val x: Int, val y: Int) {
  def add = x + y
  def mul = x * y
}

object IntPair {
  def apply(x: Int, y: Int) = new IntPair(x, y)
}

val pair = IntPair(3, 4)
pair.mul // Returns 3 * 4
```

Traits

- Encapsulates method and field definitions, much like classes
- A class can mix in any number of traits → 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")
```

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
- Special syntax support with => operator

```
val f1 = (i: Int) => i % 2 == 0 // Special operator
val f2 = (s: String) => s.toUpperCase + s.length

val arr = Array(1, 2, 3, 4)
arr.filter(f1) // Returns Array(2, 4)

val name = f2("eivind") // Returns "EIVIND6"
```

Closures/anonymous functions

- Scala – Anonymous functions
- Like anonymous classes in Java (and Scala)
- 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"
```

Scala/Java – Avoid Duplicate Code

```
val arr = Array(1, 2, 3, 4)
val bin = arr.map(_ % 2) // Returns Array(1, 0, 1, 0)
val even = arr.filter(_ % 2 == 0) // Returns Array(2, 4)
```

Scala $\Uparrow$ — Java $\Downarrow$

```
int[] arr = {1, 2, 3, 4};
int[] bin = new int[arr.length];
int[] even;
for(int i = 0; i < arr.length; i++) {
  bin[i] = arr[i] % 2; // Add to bin
}
for(int i = 0; i < arr.length; i++) {
  if(arr[i] % 2 == 0) { ... } // Add to even
}
```

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"
```

For expressions

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

```
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
```

Conclusion

- Object orientation and functional programming combined
- Static and compiled – benefits, not obstacles
- 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/