Scala

Object-oriented and functional programming combined

Eivind Barstad Waaler

BEKK/UiO

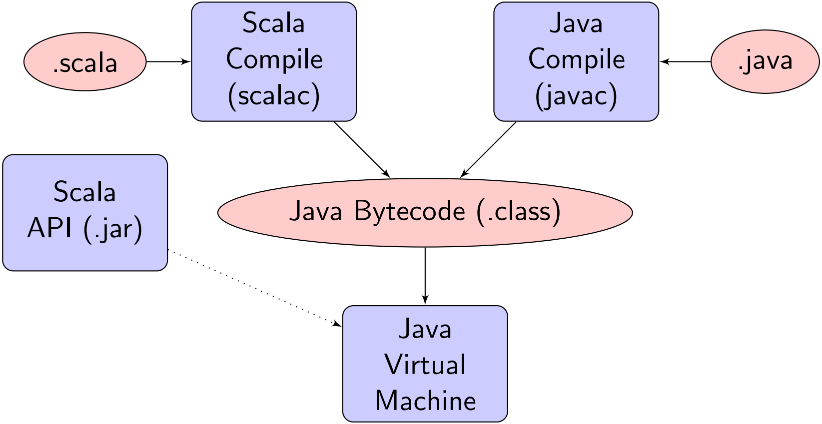
INF3110 – November 16, 2009

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 2.7.7 → 2.8

Scala & Java


.scala
Scala
Compile
(scalac)
Java
Compile
(javac)
.java
Scala
API (.jar)
Java Bytecode (.class)
Java
Virtual
Machine

Scala/Java Example

```scala
class IntMath(val x: Int, val y: Int) {
  def mul = x * y
}
```

Scala ⇑ — Java ⇓

```java
class IntMath() {
  int x, y;

  public IntMath(int x, int y) {
    this.x = x; this.y = y;
  }

  public int mul() { return x * y; }
}
```

Scala/ML Example

```scala
def fac(n: Int): Int = n match {
  case 0 => 1
  case n => n * fac(n - 1)
}

// Alternatively
def fac(n: Int): Int = if(n == 0) 1 else n * fac(n - 1)
```

Scala $\Uparrow$ — ML $\Downarrow$

```sml
fun fac 0 = 1
  | fac n = n * fac(n - 1)

(* Alternatively *)
fun fac n = if n = 0 then 1 else n * fac (n-1)
```

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

Functions as Parameters

```
// Bind function to value 'isEven'
val isEven = (i: Int) => i % 2 == 0

isEven(4) // true
isEven(5) // false

val arr = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// override def filter(p : (A) => Boolean) : Array[A]
val even = arr.filter(isEven) // Array(2, 4, 6, 8, 10)
```

Functions as Return Values

```scala
// Function creates new threshold function
val thFunc = (th: Int) => (x: Int) => x < th

// Function for threshold 10
val th10 = thFunc(10)

th10(20) // 20 < 10 = false
th10(5) // 5 < 10 = true
```

Anonymous Functions

- Like anonymous classes in Java (and Scala)
- The _ (underscore) can be used to anonymize arguments

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

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

// Shorter version using _ and type inference
arr.filter(_ % 2 == 0)
```

Partially Applied Functions & Currying

```scala
val plusFunc = (x: Int, y: Int) => x + y
plusFunc(2, 3) // 2 + 3 = 5

// Partially applied function
val plus2 = plusFunc(2, _: Int)
plus2(3) // 2 + 3 = 5

// Currying
def plusCur(x: Int)(y: Int) = x + y
plusCur(2)(3) // 2 + 3 = 5
val plus5 = plusCur(5) _
plus5(5) // 5 + 5 = 10
```

Pattern Matching

```scala
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"
```

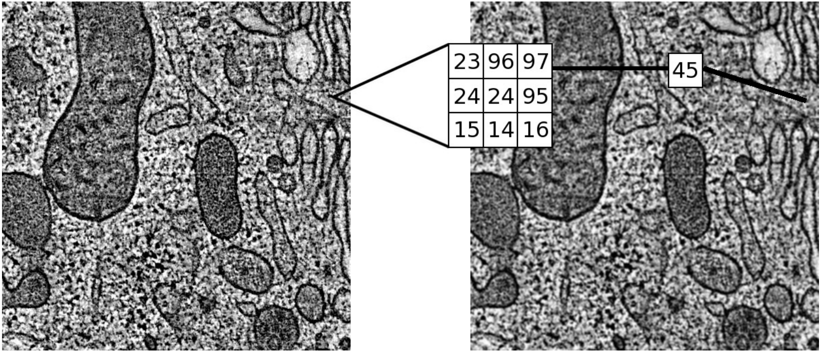
Example – Image Processing

- Stack many basic operations on single image
- Computationally heavy – parallel computing
- Example: average blur filter

```
val img = loadImage("/cell.jpg")
val se = StrEl(Square, 3)
val avg_img = img.avg(se) // <= How to implement avg?
```

Example – Average Blur Filter

- Structuring element (here 3x3)
- New point is average of area covered by se



Example – Objects

- Matrix – Underlying data structure
- Image – Image operations
- GrayScaleImage

```
class Matrix[T] { ... }

class Image {
  type DataType // abstract data type
  ...
}

class GrayScaleImage extends Image {
  type DataType = Matrix[Int]
  ...
}
```

Example – Function as Parameter

- Average – sum elements and divide by size
- For each pixel – run average function

```
class Matrix[T] {
  // 'op' is a function transforming sequence to single
  def seOp(se: StrEl[Int], op: (Seq[T]) => T): Matrix[T]
}

class GrayScaleImage {
  def avg(se: StrEl[Int]) = {
    data.seOp( // Call 'seOp' defined above
      se,
      (seq) => seq.reduceLeft(_ + _) / seq.size
    )
  }
}
```

Example – Parallel Computing

- Future – call function without waiting for answer
- Split processing into multiple futures
- Merge results

```
// Run two futures in parallel
val add1 = future { (1 to 10).reduceLeft(_ + _) } // 55
val add2 = future { (11 to 20).reduceLeft(_ + _) } // 155

// Wait for result
awaitAll(100, add1, add2)

// Merge
val result = add1() + add2() // 210
// Same as: (1 to 20).reduceLeft(_ + _)
```

Example – Divide Work by Currying

- Curry the opSe function (seen previously)
- Divide image into multiple windows

```
// Curried function
def seOp(se: ..., op: (Seq[T]) => T)(win: Window): ...

def avg(se: StrEl[Int]) = {
  val op = data.seOp( // Call 'seOp' defined above
    se,
    (seq) => seq.reduceLeft(_ + _) / seq.size
  ) _ // Leave window out => function

  parallel(data, op) // Next foil
}
```

Example – Continued...

- One future for every function/window combination

```
def parallel(data: ..., op: ...) = {
  val windows = data.split // Split image into windows

  val futures = for(win <- windows) yield future {
    op(win) // Run the curried function on the data
  }

  // Wait for all threads to complete
  val results = awaitAll(5000, futures: _*)

  // Merge results - assumes + method on data
  results.reduceLeft(_ + _)
}
```

Conclusion

- Object orientation and functional programming combined
- Static language with inference – appears dynamic?
- Seamless integration with Java – run in existing environment
- Powerful language $\longrightarrow$ high complexity?

More Info:

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