Scala for Java Programmers

- Scala
  - Statically-typed, object-oriented / functional programming hybrid
  - Developed by Martin Odersky at École Polytechnique Fédérale de Lausanne
  - Inspired by criticism of Java; recent research in programming languages
  - Runs on the JVM - compiled to Java bytecode
  - Cross-compatible with Java
- Who uses Scala?
  - Twitter, Apple, Foursquare, Novell, Siemens, Xerox, LinkedIn...
  - 15th in GitHub repos and 16th in StackOverflow questions (2013)
- Why?
  - Scalability and performance
  - Expressiveness - less verbose than Java
  - Type safety

Scala Syntax

```scala
var capital = Map("US" -> "Washington", "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

Listing 1: A simple Scala program

```java
Map<String,String> capital = new HashMap<String,String>();
capital.put("US","Washington");
capital.put("France", "Paris");
capital.put("Japan", "Tokyo");
System.out.println(capital.get("France"));
```

Listing 2: Meanwhile, in Java-land...

Pattern matching

```
1   def name: String = path.split('/').lastOption match {
2     case s: Some[String] => s.get.split('.').head
3     case None => "/"
4   }
```

Listing 3: Pattern matching on an option (from MeteorCode Pathway)

```
1 // Define a set of case classes for representing binary trees.
2 sealed abstract class Tree
3 case class Node(elem: Int, left: Tree, right: Tree) extends Tree
4 case object Leaf extends Tree
5 // Return the in-order traversal sequence of a given tree.
6 def inOrder(t: Tree): List[Int] = t match {
7   case Node(e, l, r) => inOrder(l) ::: List(e) ::: inOrder(r)
8   case Leaf          => List()
9 }
```

Listing 4: Case classes and pattern matching (from scala-lang.org)

Functional programming

```
1 def canadianify(s: String) = s + ", eh?"
2 def safeStringOp(s: String, f: String => String) = {
3   if (s != null) f(s) else s
4 }
5 safeStringOp(null, canadianify) // returns "null"
6 safeStringOp("Functional programming", canadianify) // returns "
      Functional programming, eh?"
```

Listing 5: Higher-order functions

```
1 val nums = List(1, 2, 3, 4)
2 nums.map(n => n + 1) // returns List(2,3,4,5)
3 nums.reduceLeft(_ + _) // returns 10
4 nums.filter(_ % 2 == 0) // returns List(2,4)
5
6 def minimum(a: Int, b: Int) = if (a < b) { a } else b
7 nums.reduceLeft(minimum(_,_)) // returns "1"
```

Listing 6: `map()`, `reduce()`, and `filter()`

Multiple Inheritance with Traits

```
1  abstract class Bird
2
3  trait Flying {
4      def flyMessage: String
5      def fly() = println(flyMessage)
6  }
7  trait Swimming {
8    def swim() = println("I'm swimming")
9  }
10 class Penguin extends Bird with Swimming
11
12 class Pigeon extends Bird with Swimming with Flying {
13     val flyMessage = "I'm a good flyer"
14 }
15
16 class Hawk extends Bird with Flying {
17     val flyMessage = "I'm an excellent flyer"
18 }
```

Listing 7: Multiple inheritance with traits.