A Brief, But Dense, Intro to Scala

Derek Chen-Becker <dchenbecker@gmail.com>

October 10, 2009

Abstract

This article attempts to provide an introduction to all of the really interesting features of Scala by way of code examples. Familiarity with Java or a Java-like language is recommended and assumed, although not absolutely necessary. The topics covered here range from basic to quite advanced, so don't worry if it doesn't all make sense on the first read-through. Feedback on pretty much anything but my choice of font is welcome at the email address above. This work is licensed under the Creative Commons Attribution-Share Alike 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/us/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

1 A Little History of Scala

Scala has its origins in a series of projects undertaken by Martin Odersky at EPFL Switzerland, notably Pizza and GJ[1], which eventually led to Martin writing the reference javac compiler in Java 1.5 with generics. Scala was originally released in 2003, hit version 2.0 in 2006, and is currently at 2.7.5[2]. Scala compiles down to 100% JVM bytecode and generally performs the same as comparable Java code. What else would you expect from the guy that wrote the javac compiler? Scala has a vibrant and rapidly growing community, with several mailing lists, a few wikis and a lot of traffic. Scala is already used in production by some very big names, notably Twitter[3], Siemens[4], and Xerox[5]. If you want more information on Scala or the Scala community, take a look at Section 10 on page 12.

2 What's Different About Scala?

Scala is a hybrid Object-Oriented/Functional language. This means that it combines some of the concepts of OO languages (Java,C#,C++) with Functional languages (Haskell, Lisp) to try to get the best of both approaches. Because of this, Scala has a unique combination of features:

- Scala is a pure Object-Oriented language. At the language level there are no primitive types, only Objects (although at the bytecode level Scala does have to do some special handling of JVM primitive types). Further, Scala separates the concepts of per-instance and static methods and fields into distinct `class` and `object` types.
- Scala has first-class function Objects. This means that Functions can be assigned to variables and passed as arguments to other functions. This makes defining and composing small pieces of logic very easy compared to having to define and implement interfaces.
- Scala supports closures. Closures are functions that capture variables or values from the scope of their definition. We use closures heavily in Lift for callback handlers since they provide a clean, simple way of passing state around between requests.

[1]There's an excellent interview with Martin on the origins of Scala at http://www.artima.com/scalazine/articles/origins_of_scala.html
[2]This is the most recent *stable* version
[3]http://www.artima.com/scalazine/articles/twitter_on_scala.html
[4]http://incubator.apache.org/esme/
[5]http://www.scala-lang.org/node/2766

- Scala uses traits to define code contracts. Traits are similar to interfaces, but can carry concrete implementations, define abstract variables and values, and specify inheritance hierarchies. Classes can inherit from multiple traits, which provides a very nice technique for composing functionality in your code.
- Scala has a base set of data structures such as Lists, Sets and Maps with a comprehensive suite of higher-level functions that operate on the collection contents. This means that you can write transformational code in a much more concise and maintainable manner.
- Scala has strong support for immutable data structures in its libraries. Immutable data structures are a key ingredient for doing highly-concurrent programming, since when the data can't be changed, you don't need to lock access to it.
- Scala has support for XML literals in the language. Given the prevalence of XML today for processing and data exchange, this makes generating and consuming XML quite easy within your applications.

In addition to these end-user features, Scala also has some very nice facilities to help library writers better communicate and enforce the intent of their code:

- Scala has a very comprehensive type system that allows you to define complex requirements and relationships in your code
- Scala has syntactic sugar for function invocation that makes it simple to write your own control structures.

3 Start With the Basics

Now it's time to start diving into some code examples to really show you what Scala code looks like and how it works.

3.1 Packages and Imports

Like Java, Scala supports packages and imports as a way to organize code. There are a few big differences in how things work in Scala:

```
/* Here we can define a package at the top of the source file, just like we would
 * in Java. */
package org.developerday.boulder.scala

// This is equivalent. Packages are really nested namespaces, like C#
package org.developerday {
  package boulder {
    package scala {
      // ...
    }
  }
}

// Because of this nesting, this does not do what most people would expect:
import scala.List
// Error: List is not a member of scala (org.developerday.boulder.scala)

// This works. Note that import statements in Scala can be in any scope
import _root_.scala.List

/* Scala supports the wildcard import just like Java, but the underscore is
 * used for the wildcard instead of the asterisk */
import _root_.scala.collection.jcl._

/* Imports in Scala are also a lot more flexible. Here we can import some
```

```
 * specific classes from a package */
import _root_.scala.collection.{Map,BitSet}

// You can also rename imports to avoid name conflicts
import _root_.scala.collection.mutable.{Map => MutableMap}

/* Renaming can also be used with wildcard imports. Here we avoid conflicting
 * with Scala's Array type */
import java.sql.{Array => JArray, _}
```

3.2 Basic Code Definitions

Now that we have namespaces covered, let's look at what goes into actually defining a class in Scala. Some important keywords:

val Defines an immutable value, similar to a final variable in Java

var Defines a mutable value. This is just like a variable in Java

def Defines a method

```
/* Here we define a class with a three arg primary constructor. In
 * the constructor we can directly define a read-only public val as well
 * as a protected read/write var. Note that the default access for classes,
 * fields and methods in Scala is "public" */
class Person (val name : String, protected var _size : Int, assoc : List[Person]) {
   // Code in the body is part of the primary constructor
   println(toString)

   /* A secondary constructor. Note that we have to chain to the primary so
    * that the fields get set. */
   def this(size : Int, assoc : List[Person]) = this("Anonymous", size, assoc)

   /* Some really private counters. private[this] means only this *instance*
    * can access these fields. Other instances of the same class cannot. */
   private[this] var sizeReadCounter : Int = 0
   private[this] var sizeWriteCounter : Int = 0

   // lazy values are computed on first access and cached
   lazy val baconPaths = compute6Degrees(assoc)

   /* Accessors can be defined to perform special handling on reads and writes
    * of properties. Accessors cannott conflict with existing field names. */
   def size = synchronized { sizeReadCounter += 1; _size }
   def size_= (a : Int) = synchronized { sizeWriteCounter += 1; _size = a }

   /* API-level accessors. The protected and private keywords in Scala are more
    * flexible about their scoping */
   private[boulder] def sizeReadCount = synchronized {sizeReadCounter}
   private[boulder] def sizeWriteCount = synchronized {sizeWriteCounter}
   protected[scala] def associates = assoc

   /* Scala supports operator overloading. Really, this is just another method.
    * Note that Scala will infer the return type of this method as Boolean. */
   def < (other : Person) = _size < other._size

   // Scala methods can also have explicit return types
   def compute6Degrees(n : List[Person]) : List[List[Person]] = Nil // for now
```

```
  // The override keyword is required to mark overridden methods
  override def toString = "%s wears size %d".format(name, _size)
}

/* A companion object has the same name as a class and represents the static
 * methods and fields on that class. Companion objects have access to all
 * private members (except private[this]) */
object Person {
  /* "Factory method" allows special syntax. The apply method allows the class
   * to be used as if it were a method (see Fred and Barney below). Unapply is
   * used for pattern matching and will be covered later on. */
  def apply(name : String, size : Int, assoc: List[Person]) =
    new Person(name,size,assoc)
  def apply(name : String, size : Int) = new Person(name,size, Nil)
  def unapply(p : Person) = Some((p.name, p._size, p.associates))

  val Fred : Person = Person("Fred", 16, List(Barney))
  val Barney : Person = Person("Barney", 12, List(Fred))
}
```

3.3 Case Classes

Case classes are a special type designed to simplify creating classes meant to simply hold data. Among other things:

- Case classes can inherit from other classes (inheriting from other case classes is being deprecated)
- Constructor parameters are automatically vals
- A companion object is automatically generated with apply/unapply methods
- Behind the scenes you automatically get overrides for equals, hashCode and a reasonable toString on the class itself

```
package org.developerday.boulder.scala

/* Define a hierarchy. "sealed" means that this class may only be subclassed
 * within this source file */
sealed abstract class Vehicle(val description : String, val topSpeed : Long)

case class Bike(desc : String, speed : Long, gears : Int)
  extends Vehicle(desc,speed)

// Constructor params may also be made into vars
case class Boat(desc : String, speed : Long, var fuel : Double)
  extends Vehicle(desc,speed)

// Case classes can have methods just like normal classes
case class Airplane(desc : String, speed : Long, var fuel : Double)
    extends Vehicle(desc,speed) {
  def takeOff {...}
}

val myBike = Bike("Pedals and brakes", 15, 1)

/* The unapply method lets you easily extract data from a case class. Here we
 * can bind the description and speed values of the bike to vals named "desc"
 * and "speed", respectively. */
val Bike(desc,speed,_) = myBike
```

3.4 Traits and Flexible Composition

```
package org.developerday.boulder.scala

// At their most basic, traits are 1-to-1 with Java interfaces
trait Computable {
 def compute (n : Int) : Double
}

// But traits also allow specification of members and method impls
trait Named {
 val version = "1.0.0"

 // abstract val: mixing into a class without a "name" val is an error!
 val name : String

 // We can provide a base implementation here
 def printName { println(name) }
}

// Traits can restrict what they can be mixed into
trait CreepyPerson extends Named with Computable {
 /* The self type redefines what "this" means. Below we access "_size" from
  * person */
 self : Person =>

 // Override the print method to make it creepier
 override def printName {
  println("Creepy! " + name + ", size = " + this._size)
 }
}

/* Error! NotAPerson needs Named, Person and Computable per the self type and
   the interfaces that CreepyPerson extends. */
class NotAPerson extends CreepyPerson
```

Using Traits is a relatively straightforward process with the `extends` (if the class has no defined superclass) or `with` keywords:

```
// Define some simple traits
trait Phoneable { def call(num : String) = {} }
trait Messageable { def message (msg : String) = {} }
trait Emailable extends Messageable {
 def email(msg : String) = {}
 override def message(msg : String) = email(msg)
}
trait SMSable extends Messageable {
 def sms(msg : String) = {}
 override def message (msg : String) = sms(msg)
}

/* Traits are mixed in with either "extends" or "with", depending on whether
 * you're defining a subclass or not. */
class ContactPerson(name : String, age : Int, contacts : List[ContactPerson])
 extends Person(name, age, contacts) with Emailable with Phoneable { }

// Disambiguation of inheritance is done via "linearization"
class iPhone extends SMSable with Emailable {
 message("Starting up!") // calls email(...)
}
```

Linearization is the process of locating the most “recent” definition of a given method or field from the inheritance tree of a class. The general rule of thumb is that it’s performed from right to left, although the real algorithm is more complex. If you want specific details, see Section 5.1.2 of the Scala Language Specification.

4 Control Structures

Scala has fewer control structures than Java. It has the standard if, do and while, but the for loop is really very different (and much more powerful). There are no continue or break keywords.

```
// All statements have a return type. If == ternary operator
def max (a : Int, b : Int) = if (a > b) a else b

// Simple looping
def whileLoop = { var count = 0; while (count < 10) { count += 1 } }
def doLoop = { var count = 0; do { count += 1 } while (count < 10)}

/* Scala doesn't support break or continue in loops. Instead, we define
 * recursive functions. For example, in Java we might have the following
 * (totally contrived) loop:
 *
 * for (int i = 0; i < max; i++ ) {
 *  if (someFunc(i) < 42.0) break;
 *  // The rest of the code
 * }
 *
 * Here's how we do that in Scala: */
def someFunc(j : Int) = j + 4
def loop (i : Int, maxVal : Int) : Unit = someFunc(i) match {
  case x if x < 42.0 => return
  case x if i < maxVal => { /* The rest of the code */; loop(i + 1, maxVal) }
  case _ => return
}
loop(0, 25)

// Define some data for the next example
val prefixes = List("/tmp", "/work", System.getProperty("user.home"))
val suffixes = List("txt", "xml", "html")

import java.io.File
/* The for comprehension is a powerful construct for operating on collections.
 * Behind the scenes, for comprehensions are transformed into some combination
 * of map, flatMap, filter and foreach. Here we're defining a function that
 * will locate a file based just on its base name without a prefix. The
 * function will search a pre-defined set of directories and suffixes. */
def locateFiles(name : String) : List[String] =
  for (pf <- prefixes;
       sf <- suffixes;
       /* Two things to mention here: first, we wrap the filename in a Some so
        * that it can be used as a generator (generators must provide a filter
        * method). Second, we use a guard directly here on the value to filter
        * out any filenames that don't exist. */
       filename <- Some("%s/%s.%s".format(pf, name, sf))
         if (new File(filename)).exists) yield filename

// After scalac -Xprint:namer (roughly) transformation, we get the real code
// that's generated.
def locateFilesAlt(name : String) : List[String] =
  prefixes.flatMap { prefix =>
    val filenames = suffixes.map { "%s/%s.%s".format(prefix, name, _) }
    filenames.filter { new File(_).exists }
  }

/* Scala also makes it easy to define your own control operations because you
 * can have multiple argument lists for a method (this is called currying),
 * and function arguments aren't required to be surrounded by parentheses.
 * Here, we're defining our own "using" statement that will automatically
 * handle closing resources when finished. We're also using structural
 * typing here so that the resource can be any object that has a close() method
 * on it. Structural typing can be a powerful tool, but behind the scenes it has
 * to use reflection, so be judicious in its use. */
def using[A <: { def close() : Unit}, B](resource : A )(f : A => B) : B = {
  try { f(resource) } finally { resource.close }
}

import java.io.{BufferedReader,FileReader}
val firstLine = using(new BufferedReader(new FileReader("/etc/hosts"))) {
  reader => reader.readLine
}
```

5 Functions in Scala

Scala has function objects. Behind the scenes these are actually very similar to anonymous classes. In fact, there are a a whole set of Function1, Function2, etc. traits that represent functions of a given number of args that are used to implement function objects. Functions can be assigned to vars or vals, returned from methods, and passed as arguments to methods.

```
/* Functions are defined within curly braces with the following syntax
 * (basically)
 *
 * (arg : type) => { function body }
 *
 * When the compiler can infer the argument type, you can omit it:
 *
 * arg => { function body }
 *
 * Placeholders (below) can further simplify function definition.
 */

val printLog =
  { msg : String =>
    println(System.currentTimeMillis + ":" + msg)
  }

// Define some data to operate on
case class User(name : String, age : Int)
val people = List(User("Fred", 35), User("Barney", 32))

// Underscores are expanded to placeholder arguments:
people.sort(_.age < _.age).take(5)

// This is the expanded version of the function above:
people.sort({(a : User,b : User) => a.age < b.age}).take(5)

/* Functions can be passed as an argument. In this case, the signature for
 * foreach is foreach(String => Unit), meaning that you pass a function that
 * takes a String and returns a void (Unit in Scala)
 */
people.map(_.name).foreach(printLog)
```

5.1 Closures in Scala

In addition to normal functions as we've shown here, Scala supports closures. A closure is a function that captures (or closes over) the variables that are defined in its scope, which means that the function can still access those variables after the scope has exited. A more complete explanation can be found at http://en.wikipedia.org/wiki/Closure_%28computer_science%29. Here is a simple example of using closures to maintain independent variables for function objects. What's really happening here is that when Scala detects that you're defining a function that uses variables outside of its scope, it will create a common object instance to hold those shared variables.

```
package org.developerday.boulder.scala

object ClosureDemo {
  // The setup function will return two closures that share a variable.
  def setup (greeting : String) = {
    var msg = greeting
    val setGreet = { in : String => msg = in }
    val greet = { name : String => println("%s, %s!".format(msg, name)) }
    (setGreet,greet)
  }

  def main (args : Array[String]) = args match {
    // Use pattern matching to extract arguments
    case Array(firstGreet,secondGreet,name) => {
      val (setFunc,greetFunc) = setup(firstGreet)

      greetFunc(name)

      // Here we're changing the shared variable that we defined in setup.
      setFunc(secondGreet)
      greetFunc(name)
    }
    case _ => println("Usage: ClosureDemo <first> <second> <name>")
  }
}
```

Running this application results in the output:

```
$ scala org.developerday.boulder.scala.ClosureDemo Hi Hello Derek
Hi, Derek!
Hello, Derek!
```

6 Data Structures in Scala

Scala has support for several data structures with higher-level functionality that simplify the code you write. The four primary structures are Lists, Sets, Maps and Tuples. Lists and Tuples are always immutable[6], while Sets and Maps have both mutable and immutable variants. Let's first take a look at Lists:

```
// Lists can be constructed via a factory method
val intList = List(1,2,3,4,5) // type inferred as List[Int]
```

[6]Technically, ListBuffer is a mutable structure that can be used to construct Lists

```
// or via the "cons" operator
val stringList = "one" :: "two" :: "three" :: Nil // inferred as List[String]

// Lists can be concatenated
val bothLists = intList ::: stringList // Becomes a List[Any]

/* Lists have many higher-order methods. Many take functions as predicates or
 * transformations. In the following cases the "_" wildcard is acting as a
 * placeholder. */
bothLists.filter(_.isInstanceOf[Int]) // List(1,2,3,4,5)
bothLists.reverse.head       // "three"
intList.map(_ + 6)           // List(7,8,9,10,11)
intList.exists(_ % 2 == 0)   // true
```

Next, we have Sets and Maps, in both mutable and immutable versions.

```
// Sets are immutable by default, but have mutators that return new instances
val takenNumbers = Set(42,31,51)
val moreTaken = takenNumbers + 75

// Apply on Sets tests for existence.
takenNumbers(42) // true
takenNumbers(41) // false

/* If you use a var then you get a nicer syntax for immutable collections that
 * support puts and gets. You have to remember, though, that each time you use
 * += or -=, you're really creating a new instance. */
var varNumbers = takenNumbers
varNumbers += 8; varNumbers -= 31

// You can explicitly make Sets mutable if desired
val mutableNumbers = _root_.scala.collection.mutable.HashSet(1,5,11)

// This is modifying the instance in place
mutableNumbers += 12

// Similar semantics apply for Maps. (X -> Y) actually makes a tuple (X,Y)
val ages = Map("Fred" -> 33, ("Ted", 45))

// Apply on maps does key-based retrieval
ages("Fred") // 33

val newAges = ages + ("Ed" -> 55)

var varAges = ages; varAges += "Ed" -> 55

val mutableAges =
  _root_.scala.collection.mutable.HashMap(ages.elements.toList : _*)
mutableAges += "Ed" -> 55
```

Finally, Tuples are somewhat like Lists, although Lists must have a homogeneous type and Tuples can be heterogeneous. Tuples are very useful for return multiple values from a method or for quickly creating an instance to hold data within your code.

```
// Tuples are immutable, heterogeneous, sequenced sets of data
val personInfo = ("Fred", 35)

// Tuple contents are accessed with ordinals
val firstName = personInfo._1; val firstAge = personInfo._2
```

```
// There is some very nice syntactic sugar for tuples (similar to unapply)
val (name,age) = personInfo // name = "Fred", age = 35
```

7 Pattern Matching

Scala has some very cool support for pattern matching in the language. I like to think of it as the "super-switch-statement", and it can be a very powerful way to decompose your logic based on data. Note that the patterns are evaluated in order, so generally you should define them as most specific to least specific.

```
def matchIt (in : Any) = in match {
  // Constant values can be combined in matches
  case 1 | 14 | "test" => { }

  /* This is a simple match based strictly on type. Note that Java's generics
   * erasure means that you can't fully match on generic classes. For example,
   * "case x : List[Any]" and "case x : List[Int]" are really the same thing
   * after erasure. */
  case x : Int => { }

  // You can also specify guard conditions on matches that allow you to
  // refine your matches.
  case x : Double if x > 42.0 => { }

  /* If you have an object with an unapply method defined on it, the object
   * can be used as part of a pattern. This can allow you to do some pretty
   * interesting things with patterns, which we'll show in a moment. */
  case Person(name, _, _) => { }

  /* Case classes automatically define unapply methods so that you can match
   * on constructor arguments. The "b @" syntax binds the whole Bike object
   * to a variable named "b". */
  case b @ Bike(_, _, gears) if gears > 3 => { /* use b here */ }

  // Tuples have an unapply, so you can bind to the contents
  case (x, y) => { }

  // Lists can be deconstructed via the "cons" operator. Here we're matching a
  // List of exactly two elements and binding them to x and y.
  case x :: y :: Nil => { }

  // In this case, x is the head of the List and y is the Tail. Note that
  // this won't match and empty List (Nil)
  case x :: y => { }

  // We can pattern match on XML literals
  case <head>{ stuff }</head> => { }

  // Underscore is the wildcard match (i.e. default)
  case _ => { }
}

/* I use regular expressions a lot, so this is one of my favorite examples.
 * Here, we'll take a look at how powerful pattern matching is combined with
 * custom extractors (the unapply method) */

/* Here we convert a string into a scala.util.matching.Regex (this class
 * roughly corresponds to java.util.regex.Pattern). The "r" method is actually
 * part of scala.runtime.RichString, and is reached via an implicit conversion.
 * The triple quotes make un unescaped string so that the regex is cleaner to
 * read. */
val colonSplit = """(\w+)\:(\w+)""".r

/* Here we'll create a custom unapply method to automatically extract string
 * values as Ints if possible. In unapply, returning Some[A] means success
 * and None means no match. */
object IntVal {
  def unapply(in : String) = try { Some(in.toInt) } catch { case _ => None }
}

def parts (in : String) = in match {
  /* The Regex instance has an unapplySeq method that will attempt to extract
   * the regular expression matching groups and assign them to vars!
   * You can also use nested matching statements and guards like you would
    * in other match statements. */
  case colonSplit(IntVal(number), value) if number == 144 => {
   println("Twelve squared = " + value)
  }
  case colonSplit(IntVal(number), value) => {
   println("Number key %d = %s".format(number.toInt, value))
  }
  case colonSplit(key, value) => { println("%s=%s".format(key, value))}
  case _ => { println("No match for " + in)}
}
```

8 XML Support

```
package org.developerday.boulder.scala

object XmlSupport {
  // XML literals are directly supported in Scala
  val simpleFragment =
    <entries>
      <entry><name>Fred</name></entry>
      <entry><name>Barney</name></entry>
    </entries>

  // String data can be embedded into XML literals via braces. XML entities
  // will be escaped, so this becomes <name>Fred&amp;Barney</name>
  val embedFragment = <name>{ "Fred&Barney" }</name>

  import _root_.scala.xml.Unparsed
  // Using Unparsed will avoid escaping: <name id="girls">Betty&Wilma</name>
  val noEscapes = <name id={"girls"}>{ Unparsed("Betty&Wilma") }</name>

  val clickable = <span onclick={"alert(\"Ouch!\");"}>Hey!</span>

  // You can use the XPath-like operators to extract matching children
  def getNamesSimple (in : NodeSeq) : List[String] =
    (in \\ "name").toList.map(_.text)
}
```

9 Working with Types

Scala has a very powerful type system. The Scala compiler also has a very powerful type inferencer and allows you to define your own implicit type conversion functions. Together, these mechanisms can allow you to write concise code but keep it type-safe.

```
/* Scala supports generics for classes, just like Java >= 1.5. You can specify
 * upper and lower bounds on types, as well as some more complex requirements
 * that we're not going to show here. In this example we're defining a type A
 * that needs to be a subclass of AnyRef, and a type T that must be a superclass
 * of List[A]. */
class VariousTypes[A <: AnyRef, T >: List[A]] {
  def listify (a : A, b : A) : T = List(a,b)
  // Methods can be generic as well
  def exists [C](check : A, within : Seq[C]) : Boolean =
    within.exists(_ == check)
  // Or if you don't care you can use a wildcard type. Seq requires a type, so
  // we use the wildcard here.
  def existsAlt (check : A, within : Seq[_]) : Boolean =
    within.exists(_ == check)
}

object VariousTypes {
  val listifier = new VariousTypes[String, Seq[AnyRef]]()

  /* Scala also supports implicit conversions between types. Be careful, since
   * these introduce magic into your code. When used sparingly and in a limited
   * scope, they can significantly improve the readability of your code. In any
   * case, make sure that implicits are well documented! */
  implicit def bool2String(b : Boolean) = b.toString

  /* Here the implicit gets invoked to convert true to "true". The basic rule is
   * that if an argument to a method doesn't match the proper type, Scala will
   * search through the currently scoped implicits to see if one will work to
   * convert the argument to one that works for the method.
   */
  listifier.print(listifier.exists(true, m), System.out)
}
```

10 Scala Resources

If you want to learn more about Scala:

- http://www.scala-lang.org/
- http://scala.sygneca.com/
- Books:
  - *Programming in Scala* by Odersky, Venners & Spoon
  - *Beginning Scala* by David Pollak
  - *Programming Scala* by Wampler & Payne (online)
  - *Programming Scala* by Venkat Subramaniam
  - *Steps in Scala* by Loverdos & Syropolous
  - *The Definitive Guide to Lift* by Chen-Becker, Danciu and Weir
- More articles and documents can be found on the scala-lang site!