

# Big Data Systems Engineering with Scala

Robin Hillyard, M.A (Oxon), Ph.D (Cantab)

# About me

- B.A/M.A Engineering Science (Oxford) (1st Class Hons) 1973
- Ph.D. Computer Science (Cambridge) 1978
- Worked in:
  - Computer-aided design (“Solid Modeling”, “Surface Modeling”) (Pascal, Algol68)
  - Artificial Intelligence/Machine Learning/NLP (Lisp, Java, etc.)
  - Object-relational database design (C)
  - Document Management (C, C++, OPL, Perl, Java)
  - Financial (Java)
  - eCommerce (Java, ColdFusion, Javascript, Groovy)
  - Healthcare
    - Privacy, security, anonymization (Java)
    - Reactive programming (Java, Scala)
    - Big-data analysis with Hadoop/Spark/GraphX/ElasticSearch (Pig, Java, Scala)

# About me (contd.)

- Current job: CTO at Dataflowage, Inc.
  - Big Data consulting
  - Specializing in Spark
- This is my third time teaching this class (which I created)
- Blog: <http://scalaprof.blogspot.com>
- LinkedIn: <https://www.linkedin.com/in/robinhillyard>
- Twitter: @Phasm1d
- Slack team\*: <https://csye7200.slack.com>
- Google Plus: <https://plus.google.com/u/0/collection/kqtKXB>

\* best way to contact me

# About me (contd.)

- 1968: wrote my first program (in Fortran)
  - solve  $\text{sech}(x)=x$
  - it worked first time.
- 1969: wrote my first plotter driver as well as first use of a “personal computer”
- 1972: wrote my first debugger (for Assembly language).
- 1983: wrote my first object-relational database.
- 1984: wrote my first unit-test runner.
- 1994: wrote my first Java program.
- 2012: wrote my first Scala program.

# About you...

- Backgrounds?
- Programming classes?
- Programming jobs?
- O-O?
- F-P?
- Java? Java8?
- Big Data?
- Functional Programming?

# Big Data Systems

## Engineering with Scala

- *Why Big Data?*
  - Until fairly recently, *most* business-oriented computer software was developed for one of these purposes:
    - personal applications (document preparation, spreadsheets, presentations, email, etc.)
    - database applications to support internal business needs
    - interactive systems for business (eCommerce)
    - analysis of finite datasets
  - But now, the internet can provide essentially infinite datasets with a huge potential for data-mining, inference, etc. Collectively, these vast sources are known as “Big Data.”



# Big Data Systems Engineering with Scala

- *Why Systems Engineering?*
  - This class aims to provide a *practical* approach to dealing with Big Data:
    - performant
    - testable
    - versatile
    - elegant
  - Will our solutions always be the shortest? The fastest possible? The most mathematically sound?
    - no—but they will be tested and effective

# Big Data Systems Engineering with Scala

- *Why with Scala?*
  - The Big Data world is increasingly turning to Scala as the language of choice:
    - Functional Programming ->
      - performance, provability, testability, parallelizable
      - Scala is used by Spark under the hood



# Academic Honesty

- Sources of information:
  - My lecture notes, code samples, blog (OK except for mid-term exam)
  - Recommended text, your notes (OK even for exams)
  - The internet (OK for clarification, background, etc. but **not** OK for copying code samples\*)
  - plagiarism will never be tolerated

\* with the exception of the term project (where all properly attributed code is allowable)

# Is this class for YOU?

- Why you *shouldn't* attend this class:
  - This is *not* an easy class;
  - You cannot just coast through without doing the work;
  - You will be learning some significant new concepts;
  - Scala is not Java (or Python);
  - Unless you have some familiarity with Haskell, Clojure, or other functional language you will probably struggle at first.

# Is this class for YOU? (2)

- Why you *should* attend this class:
  - You will be challenged;
  - You will learn a lot about *good programming techniques*, most of which are applicable to *any* language;
  - You will be more employable—even if you can't find a Scala job right away;
  - You will have a lot of fun:
    - programming in a functional way is very satisfying;
    - especially when it comes to the term project

# What is Scala?

- Scala
  - stands for *Scalable Language*
  - is a “blend of object-oriented and functional programming concepts in a statically typed language”
  - is a general-purpose language
  - is based on the Java Virtual Machine and its “bytecode,” just the same as Java, Groovy, Clojure, JRuby, Jython, etc.
  - is therefore bi-directionally compatible with Java (and other JVM languages). Although Scala has its own definitions for things such as *List*, you can, if you like, use *java.util.List*

# Why not Python?

- Python is a very popular language which is used by programmers in many disciplines, including Data Science.
- Python is interpreted, whereas Scala is compiled:
  - What difference does this make in practice? Compiled is better if you are running in a production environment; interpreter is better if you are in research/data science mode.
- Scala is strictly typed whereas Python is dynamically typed:
  - In practice, this means that you can probably write Python more quickly (less time spent trying to keep the compiler/interpreter happy) but typing helps ensure your code does what you expect and won't crash!

References: Quora; 6 points;



# Quick Example - especially for those migrating from Java

```
package edu.neu.coe.scala
case class Mailer(server: String) {
  import java.net._
  val s = new Socket(InetAddress.getByName(server), 587)
  val out = new java.io.PrintStream(s.getOutputStream)
  def doMail(message: String, filename: String) = {
    val src = scala.io.Source.fromFile(filename)
    for (entry <- src.getLines.map(_.split(","))) out.println(s"To: ${entry(0)}\nDear $
{entry(1)},\n$message")
    src.close
    out.flush()
  }
  def close() = {
    out.close()
    s.close()
  }
}
object EmailApp {
  def main(args: Array[String]): Unit = {
    val mailer = new Mailer("smtp.google.com")
    mailer.doMail(args(0), "mailinglist.csv")
    mailer.close
  }
}
```

Notice that we can place imports wherever needed.

Notice how we are including some Java types here, although that is unusual.

Notice that there are no semi-colons

Notice that Scala uses [] for types, not <>

Here we create a main program which is just like one in Java, except here it is not marked "static". There is a better way, however.



# Another example

```
object Ratios {
```

```
  /**
```

```
    * Method to calculate the Sharpe ratio, given some history and a constant risk-free rate
```

```
    * @param investmentHistory the history of prices of the investment, one entry per period, starting with the most recent price
```

```
    * @param periodsPerYear the number of periods per year
```

```
    * @param riskFreeRate the risk-free rate, each period (annualized)
```

```
    * @return the Sharpe ratio
```

```
  */
```

```
  def sharpeRatio(investmentHistory: Seq[Double], periodsPerYear: Int, riskFreeRate: Double): Double =  
    sharpeRatio(investmentHistory, periodsPerYear, Stream.continually(riskFreeRate))
```

```
  /**
```

```
    * Method to calculate the Sharpe ratio, given some history and a set of benchmark returns
```

```
    * @param investmentHistory the history of prices of the investment, one entry per period, starting with the most recent price
```

```
    * @param periodsPerYear the number of periods per year
```

```
    * @param riskFreeReturns the actual (annualized) returns on the risk-free benchmark
```

```
    * @return the Sharpe ratio
```

```
  */
```

```
  def sharpeRatio(investmentHistory: Seq[Double], periodsPerYear: Int, riskFreeReturns: Stream[Double]): Double = {
```

```
    // calculate the net gains of the investment
```

```
    val xs = for (x <- investmentHistory.sliding(2)) yield x.head - x.last
```

```
    // calculate the net returns on the investment
```

```
    val ys = (xs zip investmentHistory.iterator) map { case (x, p) => x / p }
```

```
    // calculate the annualized returns
```

```
    val rs = for (y <- ys) yield math.pow(1 + y, periodsPerYear) - 1
```

```
    // calculate the Sharpe ratio
```

```
    sharpeRatio(rs.toSeq, riskFreeReturns)
```

```
  }
```

```
  /**
```

```
    * Method to calculate the Sharpe ratio, given actual and benchmark returns
```

```
    * @param xs the actual historical returns (expressed as a fraction, plus or minus) of the given investment
```

```
    * @param rs the actual historical returns (expressed as a fraction, plus or minus) of the benchmark
```

```
    * @return the Sharpe ratio
```

```
  */
```

```
  def sharpeRatio(xs: Seq[Double], rs: Stream[Double]): Double = {
```

```
    val count = xs.size
```

```
    // calculate the excess returns
```

```
    val zs = (xs zip rs) map { case (r, f) => r - f }
```

```
    // calculate the Sharpe ratio
```

```
    zs.sum / count / math.sqrt((xs map (r => r * r)).sum / count)
```

```
  }
```

```
}
```

Notice that wherever we define a variable or parameter, it's **always** of form *name : Type*

Look at these *for* loops. Each of them returns a value via the *yield* keyword

Notice that “variables” must be preceded by “val” or “var” but don’t necessarily need a type annotation.

Notice that we don’t need a *return* keyword.

# Another Example: Ingest

```
package edu.neu.coe.scala.ingest
```

```
import scala.io.Source
```


```
class Ingest[T : Ingestible] extends (Source => Iterator[T]) {  
  def apply(source: Source): Iterator[T] = source.getLines.toSeq.map(e =>  
    implicitly[Ingestible[T]].fromStrings(e.split(",").toSeq)).iterator  
}
```

```
trait Ingestible[X] {  
  def fromStrings(ws: Seq[String]): X  
}
```

```
case class Movie(properties: Seq[String])
```

```
object Ingest extends App {  
  trait IngestibleMovie extends Ingestible[Movie] {  
    def fromStrings(ws: Seq[String]): Movie = Movie.apply(ws)  
  }  
  implicit object IngestibleMovie extends IngestibleMovie  
  
  override def main(args: Array[String]): Unit = {  
    val ingester = new Ingest[Movie]()  
    if (args.length > 0) {  
      val source = Source.fromFile(args.head)  
      for (m <- ingester(source)) println(m.properties.mkString(", "))  
      source.close()  
    }  
  }  
}
```

This example of Scala uses some of the advanced features of the language, including *type classes* and *implicits* to allow us a truly generic solution to ingestion. Don't expect to understand it just yet.



# What Scala is not?

- Scala *is not*
  - a “scripting language,” although it can be and is used in many places scripts might be found, such as in the *build.sbt* file (used to build Scala applications)
  - a domain-specific-language (DSL) although it can be used to build DSLs
  - too esoteric for people to learn!

# Functional vs. Procedural

	Functional	Procedural
Paradigm	Declarative (you describe/declare problem)	Imperative (you explicitly specify the steps to be taken to solve problem)
Functions	First-class objects, composable	“callbacks”, anonymous functions, etc.
Style	Recursive, mapping, decomposition	Iterative, loops, if...
Examples	Lisp, Haskell, Scala...	Almost all other mainstream languages, including Java* *Java8 has many FP concepts

# Why Scala?

- Scala:
  - is fun (!) and very elegant;
  - is mathematically grounded;
  - is powerful (many built-in features);
  - has syntactic sugar: sometimes you can express something in different ways—however, usually, one form is “syntactic sugar” for the other form and the two expressions are entirely equivalent;
  - is predictable: once your program is compiling without warnings, it *usually* does what you expect;
  - is ideal for reactive programming (concurrency);
  - is ideal for parallel programming (map/reduce, etc.);
  - used by Twitter, LinkedIn, Foursquare, Netflix, Tumblr, etc.



# Is Scala really an important language today?

- Scala is presently in 32nd position in the TIOBE index (<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>)
  - Up one place since September 2015
- Ratios of Java : Scala
  - Google search
    - 34 in general (“xxx programming”)
    - 6.5 in context of “Big Data”
    - 18.5 in context or “reactive”
  - “stackoverflow.com”
    - 22 (but Scala is growing)



# How do we use Scala?

- Very briefly, there are at least *five* ways:
  - Create a module (file) called *xxx.scala* and compile it, build it, run it (using shell, *sbt*, or IDE)
  - The “REPL” (*read-execute-print-loop*) (using shell commands `scala` or `sbt console`)—or IDE—**the REPL is your friend**
  - Create a “worksheet” called *xxx.sc* and save it (IDE only)
  - Use a *REPL-with-Notebook* such as Zeppelin
  - Dynamically inject code into your system using Twitter’s “eval” utility.
- Two popular IDEs: *Scala-IDE* (based on *Eclipse*) and *IntelliJ-IDEA* (what I recommend).

# Object-oriented programming

- The central concept of O-O is that methods (operations, functions, procedures, etc.) are invoked **on objects** (as opposed to local/global data structures in memory)
- A class is defined as the set of methods that can be invoked on its instances and the set of properties that these instances exhibit
- Objects are instances of a *class* and contain both methods and data (properties)
- Classes form an inheritance hierarchy such that sub-classes inherit methods and data from their super-classes (or may override)
- Examples: Scala, Java, C++, C#, LISP, Objective-C, Python, PHP5, Ruby, Smalltalk

# What is Functional Programming?

- Functional Programming:
  - functions are first-class objects and can be *composed*
  - avoids side-effects—variables and mutable collections are the exception rather than the rule
    - this, in turn, leads to *referentially transparency* (provable via substitution—Turing complete—“ $\lambda$ -calculus”)
    - and to predictability (and, thus, testability)
    - and so is inherently scalable and parallelizable
  - lazy evaluation is a major, significant aspect of functional programming
  - uses recursion when that is the appropriate mathematical definition rather than iteration

# Typed functional programming

- Typing adds a new dimension to FP
  - ML, Miranda, Haskell, Scala, Clojure, etc. are all typed
  - Variables have types; types have kinds; kinds have...?
- Typing eliminates\* “casting” errors at run-time

\* well, Scala does allow a certain amount of circumventing strict typing so it's still possible (but rare) to run into class cast errors.

# How does O-O mix with FP?

- In O-O/FP language (like Scala):
  - classes have fields (just like Java, say) but these fields can contain *either* values *or* functions:
    - when they are *function* fields, we call them *methods* (similar to Java).
  - a *function* has 0..N parameter sets—each enclosed in “()”;
    - a *method* has, additionally, “this” as a *parameter set*; and, by convention, “this” is invisibly *prefixed* to the method name (other parameter sets *follow* the method name);
  - example method *apply* on an indexed sequence (with one parameter set) can be invoked thus (all equivalent):
    - `this apply x`
    - `this.apply(x)`
    - `this(x)`
    - `(x)`



# What's imperative programming?

- Imperative programming comes from an understanding of the hardware architecture of a computer, whether a Turing machine, a Von-Neumann machine, or whatever:
  - The notion is that you have a block of addressable memory and at least one register (the accumulator).
  - The program, as well as the data, is stored in this memory
    - another register (the program counter—PC) points to the current instruction;
    - the system steps through the program by changing the value of the PC.
  - To *use* a value in your program you must:
    - find its address;
    - issue a fetch (or load) statement to get the value into the accumulator;
    - perform some arithmetic (or store) operation on it.
  - To *store* a value from the accumulator into memory you must:
    - determine an appropriate address.
    - issue a store statement to that address.
- All programs written in imperative style *ultimately* perform these operations.



# Other differences

- Pointers:
  - In procedural programming, pointers are used everywhere (although good O-O style masks use of some pointers by *this*, etc.)
    - Unfortunately pointers which are mutable can be null—result: *NullPointerException* (NPE).
  - Scala allows *null* for compatibility with Java, but you should never use it! (There are better ways)
- Types:
  - With non-strict typing (e.g. as in Java), generic types are used
    - Unfortunately, sometimes these are wrong—result: *ClassCastException*.
  - Scala has *strict* typing

# Scala vs. Java 8

- There are three concepts that are common to both of these languages:
  - Lamdas (anonymous functions)
  - Methods in traits/interfaces
  - Stream operations on collections
  - SAM (single-abstract-method) types (not in Scala until 2.12)
- Otherwise, Scala and Java 8 are completely different: Java 8 lacks all of the other functional programming “goodies” that we will learn about.

# Declarative Programming

- If you go back in history to the dawn of programming you will find that (almost) all programs were written in the imperative style\*:

```
package edu.neu.coe.scala;
public class NewtonApproximation {
    public static void main(String[] args) {
        // Newton's Approximation to solve cos(x) = x
        double x = 1.0;
        int tries = 200;
        for (; tries > 0; tries--) {
            final double y = Math.cos(x)-x;
            if (Math.abs(y)<1E-7) {
                System.out.println("the solution to cos(x)=x is: "+x);
                System.exit(0);
            }
            x = x + y/(Math.sin(x)+1);
        }
    }
}
```

\* you could also call this the “Von Neumann” style (search for Backus Turing Award)

Does this language look familiar? It should—it’s Java.

# Observations on Newton-Raphson Approximation

- This is written in Java but the style is similar to the original Fortran (Backus et al)—the loop would have been a GOTO originally and tries would have been called “I” or “N”
- If you’re unsure about the method, see:
  - [http://en.wikipedia.org/wiki/Newton's method](http://en.wikipedia.org/wiki/Newton's_method)
- Everything is static (there is no O-O here, as you’d expect)
- There’s not just one but *two* variables (*x* and *tries*)
- The program tells the system *exactly* how to run the calculation
  - there’s a loop until some terminating condition is met
  - meanwhile, the best estimate (*x*) is explicitly updated

# So, what's wrong with the imperative/declarative style?

- In the beginning a program was loaded into memory (from magnetic/paper tape or punched cards) and ran until it was finished (or “abend”ed).
- As computers became more shareable, used disks, had other devices attached, the idea of interrupts was born:
  - An interrupt put the current program on hold and started running a different program for a while—when finished it would go back to the original program
  - The less tolerant of delay was the device, the higher the priority of the interrupt (a card or paper tape reader was intolerant, a disk a little more tolerant)
- This worked fine until networks and shared databases came along (early 70s)—even then it worked OK until the sheer volume and frequency of network interrupts—and the number of database users got too high
- This problem was “solved” by inventing “threads” (sub-processes)—and using a programming language that explicitly allows threads to be programmed at a low level, e.g. Java



# The straw that broke the camel's back

- Have you ever tried to develop and, more importantly, *test* a threaded application?
- I have—it can be maddeningly frustrating.
- Typical “solutions”:
  - synchronize mutable state—but be careful not to synchronize too much at once lest you run into race conditions and deadlocks
  - custom Executor services and a greater number of processors (make the problem “go away”—for a while)
  - defensive deep copying (Aaargh!)



# Part two

# Important stuff

- The next few slides are important!
- But don't worry, we'll be covering this stuff in *much greater detail* in the upcoming weeks.

# Types, values, etc.

- The strength of Scala rests to a large extent on its strict type safety. How do types, values and other properties relate to each other?
- Variables\* all have five important aspects:
  - **type**: lets you and the compiler know what properties the variable supports—methods, range of legal values, etc.
  - **value**: the (current) value of the variable;
  - **name**: the identifier of the variable (i.e. how it gets referenced);
  - **scope**: lets you and the compiler know where the variable can be referenced;
  - **evaluation mechanism**: lets you and the compiler know how/when the variable “receives” its value

*\*By “variable”, a Scala programmer doesn’t mean something that can necessarily change its value during a run. A variable is used in the sense of algebra—it’s something that stands for some sort of quantity.*

# A quick explanation: types

- Different styles of type:
  - **functions**: these can transform variable(s) of one (or more) types into a variable of another type.  
e.g. `(x: Int) => x.toString`
  - **scalars**: ordinary value types such as *Int*, *Double*, *String*, *LocalDate*, etc.  
e.g. `3`
  - **containers**: wrappers around groups of variables which may contain zero thru N members:
    - **longitudinal\*** (collections): e.g. *Iterator*, *List*, *Array*, etc.  
e.g. `List("a", "b", "c")`
    - **transverse\***: e.g. *Option*, *Tuple*, *Try*, *Future*, *Either*, etc.  
e.g. `Tuple("a", 1, 3.1415927)`
    - **hybrid\***: e.g. *Map*, *Seq[Tuple2[String, Int]]* etc.  
e.g. `Map("a" -> 1, "b" -> 2)`

*\*These terms are not in common use: I use them to help differentiate different types of container.*

# A quick explanation: scope

- Scope in Scala is similar (but not the same) as in Java.
- Example of legal code:

```
val x = 3  
  
def y = {  
  val x = 5  
  x + 8  
}
```

The value of *y* is 13 (not 8).



# A quick explanation: evaluation mechanism

- You can reference a variable in several ways, each with a different evaluation mechanism:
  - **direct reference (call by value)**: a variable has a value and that value is effectively substituted for the variable wherever that variable is referenced.
    - The same thing can be true of a parameter to a function
  - **pointers (call by name)**: although we don't normally talk about pointers in Scala, it's easy to think of a pointer to some piece of memory. The pointer itself might not change but there's nothing to stop the memory location changing its value — that would be a mutable variable.
    - The same thing can be true of a parameter to a function
  - **lazy**: if a variable is lazily-evaluated, its actual value is not calculated until it is needed. Lazy variables are a lot like class (or object) methods. Function parameters which are call-by-name are essentially lazy, but they get re-evaluated if referenced more than once inside the function.

# Exercise 1 - REPL

A. `def f(x: Int) = x*x`  
B. `f(9)`  
C. `def f(x: Int) = {println(x); x*x}`  
D. `f(9)`  
E. `val y = f(9)`  
F. `lazy val z = f(9)`  
G. `z + 19`  
H. `f{println("hello"); 9}`  
I. `def f(x: => Int) = x*x`  
J. `f{println("hello"); 9}`  
K. `def f(x: () => Int) = x()*x()`  
L. `f{() => println("hello"); 9}`

## Questions:

- What can you tell me about the REPL given A and C?
- What's the difference between E and F?
- What's going on with F and G?
- What's happening in I and J?
- What's the difference in K and L?

# Constructors & Extractors

- You're familiar with the idea of *constructors* such as:
  - `List(1,2,3)`
  - `Complex(1.0,-1,0)`
- But, surely, if you can construct objects, you must be able to "deconstruct" (or extract) them:

```
case class Complex(real: Double, imag: Double)
val z = Complex(1.0,-1.0)
z match {
  case Complex(r,i) => println(s"$r+i$i")
}
```

```
def show(l: List[Int]): String =
  l match {
    case Nil => ""
    case h::t => s"$h,"+show(t)
  }
```

# Exercise 2 - REPL

```
A. case class Complex(real: Double, imag: Double)
B. val z = Complex(1,0)
C. z match {case Complex(r,i) => println(s"$r i$i"); case _ => println(s"exception:
  $z1")}
D. val l = List(1,2,3)
E. l match { case h :: t => println(s"head: $h; tail: $t"); case Nil =>
  println(s"empty") }
```

## Question:

- What's happening in A
- What about C?
- What about D/E?

# Assignment 1 (part 1)

- Purpose:
  - The purpose of this assignment is mainly to ensure that you each have a development environment that will **compile**, **test** and **run** Scala programs.
- How to start:
  - Go to: <https://github.com/rchillyard/Scalaprof/tree/master/HelloWorld>
  - DO NOT clone the entire repo (at least not yet)
  - Copy the following resources (retaining the structure of the project as that is important):
    - build.sbt
    - src/main/scala/CSYE7200/HelloWorld.scala
    - src/test/scala/CSYE7200/HelloWorldSpec.scala



# Assignment 1 (part 2)

- build.sbt:

```
name := "HelloWorld"
```

```
version := "1.0"
```

```
scalaVersion := "2.11.8"
```

```
libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.6" % "test"
```



Keep a blank line between each of these.

- HelloWorld.scala

```
package CSYE7200
```

```
object HelloWorld extends App {  
  def greeting = "Hello World!"  
  println(greeting)  
}
```



Note that you need *def* here, not *val* (so that you can check it from Spec).

- HelloWorldSpec.scala

```
package CSYE7200
```

```
import org.scalatest.{FlatSpec, Matchers}
```

```
class HelloWorldSpec extends FlatSpec with Matchers {  
  behavior of "HelloWorld"  
  it should "get the correct greeting" in {  
    HelloWorld.greeting shouldBe "Hello World!"  
  }  
}
```

# Assignment 1 (part 3)

- using build.sbt



There are lots of ways to start a fresh Scala project. IntelliJ, Eclipse will do it. Activator (play) will do it. Or you can just roll your own. But using this will at least get the basic structure right.

- You can create a project using IntelliJ or Eclipse and everything will be taken care of for you.
- Or you can just type into the shell:
  - *sbt test run*
  - which will compile, test and run in your shell.
  - Or, to use your code in the REPL just by typing:
  - *sbt console*

# Assignment 1 (part 4)

- Go to kaggle.com
  - If you haven't already done so, register, log in and agree to their data usage terms (go to datasets).
  - Download this file:
    - <https://www.kaggle.com/deepmatrix/imdb-5000-movie-dataset> (you will need your cell phone to accept and enter a code)
  - Move it to some place that's easy to remember on your machine, for example: ~/kaggle/imdb-5000-movie-dataset.csv

# Assignment 1 (part 5)

- Next, go to the class repository (see [Blackboard/.../Course Material/Resources](#))
  - Clone the repo with git, github, SourceTree, etc.
  - Download this file:
    - `src/main/scala/edu/neu/coe/scala/ingest/Ingest.scala`
  - Run the program (providing the argument “`~/kaggle/imdb-5000-movie-dataset.csv`” - or wherever you placed the file you downloaded from Kaggle)
  - You should have no problems with this so far — you should see a list of 5000 movies, each with its properties

# Assignment 1 (part 6)

- Now, edit the *Ingest.scala* file so that you only get to see the movies which originate in New Zealand
- You should replace the line which prints the properties with these two lines:

```
val kiwiMovies = for (m <- ingester(source); if (m.properties(20)=="New  
Zealand")) yield m  
println(kiwiMovies.size)
```

- Let me know how many you get in your submission.
- **Due Tuesday night at 11:59pm**