Just Enough... Scala for Spark

2015

Acknowledgments

This presentation is adapted from Dan Garrette's *Scala Basics* web page.

- Dan is presently a CS postdoc at Univ. of Washington, and created this material to assist students in his courses at UT Austin
- The original page is hosted at
  - http://www.dhgarrette.com/nlpclass/scala/basics.html

- Additional material thanks to Brian Clapper's Scala Bootcamp

Java Language

- Embody object-oriented principles
- Closer to C++ than to Smalltalk
- Easier to use, harder to mess up than C++
  - Automatic memory management
- From a programming language point of view it was a “MVP” (minimal viable product)
- However, that made it easy to learn
  - Though missing many sophisticated language features
- Java Community Process (open but slow)

JVM (Java Virtual Machine)

Goals?

- Like Smalltalk, run in a known environment
- Partially isolated from underlying OS/Hardware
    - “Write once run anywhere”
- Hopefully perform better than Smalltalk
- Automatic memory management via garbage collection

Success?

- Mostly! Great performance, portability!

Scala: a JVM language

Scala was designed from the beginning as a JVM language. Although it has many features which work differently than Java, and many other features which Java lacks entirely,

- it compiles to JVM bytecode,
- deploys as .class or .jar files,
- runs on any standard JVM,
- and interoperates with any existing Java classes.

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._
object FrenchDate {
  def main(args: Array[String]) = {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

Scala, scalac

To compile a program in Scala:
put your source code in a file
(Scala does not have the package/folder restrictions that Java does)
run the scalac compiler executable that is part of the Scala distro

```
scalac HelloWorld.scala
```

To run your program, use the scala command:

```
scala –classpath . HelloWorld
```

What about that interop with Java? You can package your Scala projects so that they include all dependencies, and can run using the traditional `java` command as well.

The Scala REPL

Scala projects are typically built with Maven, or a tool called sbt, which allows configuration of build tasks themselves in Scala.

However, we don’t need to build or even compile Scala code to try it out!

Scala includes a REPL (“read-eval-print-loop”) where we can experiment and test out code. Just run the scala binary:

```
Welcome to Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_05).
Type in expressions to have them evaluated.
Type :help for more information.

scala> val s = "Hello World!"
s: String = Hello World!

scala> println(s.toLowerCase)
hello world!
```

Follow along with this presentation by trying out code in the Scala REPL!

SBT

SBT is the Scala Build Tool (www.scala-sbt.org).

The "S" used to stand for "Simple", but it's not all that simple. (It's gotten a lot simpler, though…)

The easiest way to create an SBT project is with the Lightbend Activator:
https://www.lightbend.com/activator/download

If you're using a Mac with Homebrew, use:

```
brew install typesafe-activator
```

Scala and IDEs

If you're doing development in an IDE, there are two main choices:

- The Scala IDE (http://scala-ide.org/), an Eclipse plugin
- IntelliJ IDEA, with the Scala plugin (version 14 or better)

There's also a Netbeans Scala plugin, but Netbeans is far less popular, so you'll have more trouble getting help.

Databricks

- Databricks is a tool and environment for
  - Easily launching, managing, and stopping Spark clusters
  - Collaboratively creating Spark code projects and queries
  - Scheduling repeated jobs
  - Publishing visualizations, dashboards, and web services

- Today, we will focus on using Databricks as an easy, interactive, cloud-based way to write Scala code.

Programming Fundamentals

To do procedural programming, we need

- Variables – a place to store data
  - “Represent” memory, but in modern architectures they are fairly far from physical memory
  - And techniques that bring them closer to physical memory often do so in surprising (though clever) ways
- Syntax (the rules about parentheses, braces, etc.)
- Flow control constructs
  - Looping (repeating sections of code)
  - Branching (conditionally jumping to other code sections)

Grab a Cheat Sheet!

- **Class handout – nice to have in print! – is by Alvin Alexander (author of O'Reilly Scala Cookbook)**
  - **http://alvinalexander.com/downloads/scala/Scala-Cheat-Sheet-devdaily.pdf**
    - **(also in PDF in box.com folder)**

- I'm good with all that, now I want style recommendations
  - http://docs.scala-lang.org/cheatsheets/

- Ok, just get me some examples and the hard stuff
  - https://github.com/lampepfl/progfun-wiki/blob/gh-pages/CheatSheet.md

Variables (and values)

There are two keywords for declaring “variables”: `val` and `var`.

- Identifiers declared with `val` cannot be reassigned;
  - this is like a `final` variable in Java
- Identifiers declared with `var` may be reassigned.

```
val a = 1
var b = 2

b = 3  // fine
a = 4  // error: reassignment to val
```

You should generally use val. If you find yourself wanting to use a var, there may be a better way to structure your code.

Style and syntax basics

- Variables and methods start with lowercase letters
- Constants start with capitals
- Class names start with a capital letter
- Everything else uses camelCase

- Blocks are delimited by { } (and are expressions)
- Multiple expressions on one line are separated with a ;
- Line ends don't require ; and whitespace is not semantically significant

Types

- Scala has a rich and complex type system
  - Here, we're interested in the most common/useful types, not an exhaustive analysis
- Value types: Double, Float, Int, Long, Short, Byte, Char, Boolean, Unit
- AnyRef (java.lang.Object)
  - Seq, List, other Scala class/trait types
  - Java object types (classes, interfaces)
- AnyVal, AnyRef derive from Any ("unified type system")

Specifying Types

Scala has powerful type inference capabilities:

- In many cases, types do not need to be specified
- However, types may be specified at any time

```
val a = 4                   // a: Int = 4
val b: Int = 4              // b: Int = 4
var c: Int = _              // c: Int = 0
```

This can make complex code more readable, or protect against errors. Types can be specified on any subexpression, not just on variable assignments.

```
val c = (a: Double) + 5    // c: Double = 9.0
```

All types are determined statically during compilation.

Common types include Int, Long, Double, Boolean, String, Char, Unit (“void”)

databricks

First data structure: Tuple

- Fixed length and element types
- Immutable
- Access elements via ._1, ._2, etc.

```
val customer = ("Bob", "bob@foomail.com", 201)
println(customer._2)

bob@foomail.com

customer: (String, String, Int) = (Bob,bob@foomail.com,201)
```

Control

Scala has many familiar control structures.

if-else

```
val x = 4
if(x > 2)
  println("greater than 2")
else if(x < 4)
  println("less than 2")
else
  println("equal to 2")
// prints "greater than 2"
```

Lab 1: Customer

- Create 3 customer “records” using Tuples
- Each customer record will contain the customer’s
  - First name
  - Last name
  - Account balance
- Calculate the mean of the 3 account values
- Use the following data:
  - Smith, John, 150
  - Jackson, Anna, 250
  - Hernandez, Tim, 350

Loops

One looping construct is “while”

```
var x = 0
while (x < 3) {
  println ("x is " + x)
  x = x + 1
}

Output:
x is 0
x is 1
x is 2
x: Int = 3
```

databricks

Lab 2: FizzBuzz

Made famous by Joel Spolsky, the "Fizz-Buzz test" is an interview question designed to help filter out the 99.5% of programming job candidates who can't seem to program their way out of a wet paper bag.

The text of the programming assignment is as follows:

Write a program that prints the numbers from 1 to 100. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”.

Congratulations, you can now code FizzBuzz in Scala!

for-each loop

Basic loop similar to Python for-in or Java for-each

```
for (x <- 1 to 5) // 1 to 5 is a "Range"
  println(x)

val xs = Vector(1,2,3,4,5)
for(x <- xs)
  println(x)

// both print numbers 1 through 5
```

Everything is an expression

In Scala, many things are expressions that are not in other languages.

Blocks are expressions that are evaluated and resolve to the value of the final expression in the block:

```
val x = {
  val intermediate1 = 2 + 3
  val intermediate2 = 4 + 5
  // will be "returned" from the block:
  intermediate1 * intermediate2
}
// x: Int = 45
```

Functions

Functions are defined using `def`

- Parameter types must be specified
- Return types are optional
  - can be inferred at compile-time (unless the function is recursive)
- Function body should be separated from signature by =
- Braces are not needed if the function body is just a single expression
- Parentheses are not needed in the function signature if there are no params
  - Empty parentheses means they are optional on the call
  - Function defined without parentheses, means they are not allowed, so the call looks like a variable access
- The return keyword is not needed (and is rarely used)

Function Examples

```
def mult(i: Int, j: Int): Int = i * j  // return type specified
def add(i: Int, j: Int) = i + j        // no braces needed
def mystring() = "something"           // parentheses option in caller
def mystring2 = "something else"       // no parentheses allowed in call
def doubleSum(i: Int, j: Int) = {      // braces for multiple statements
  val sum = i + j
  sum * 2                              // "return value"
}
def optSum(i:Int, j:Int = 100) = i + j // j defaults to 100 if omitted

mult(2,3)            // res55: Int = 5
add(2,3)             // res48: Int = 5
mystring()           // res50: String = something
mystring             // res51: String = something
mystring2            // res52: String = something else
doubleSum(2,3)       // res53: Int = 10
optSum(5)                                   // res54: Int = 105
```

Function objects and lambdas

Scala also supports function objects, which have types, and can be assigned:

```
scala> val add = (a:Int, b:Int) => a+b
add: (Int, Int) => Int = <function2>

scala> add(4,5)
res1: Int = 9
```

… and lambdas, or function expressions, which can be used inline without an assignment:

```
scala> List(3,4,5).map(n => n*n) // type is inferred
res3: List[Int] = List(9, 16, 25)
```

Lab 3: Functions

- Starting with your "customer" code from Lab 1
  - Write a function that takes a customer (tuple) and returns the customer's last name

  - Write a function that takes 2 customers and returns a tuple containing each of their last names as well as the mean of the account values

  - Write a function that takes 2 customers. *Another function defined on 2 Ints returning a Double* – and applies that function to the 2 customers' account values, returning the result
    - Try it out to calculate sum, mean, product of the values

Scopes and Closures

Scala supports nested scopes (e.g., functions defined inside of other functions) as well as closures.

This means that any time we create a function which references free variables defined in an outer scope, that function holds a reference to those outer scope variables, and those variables maintain their definition-site meanings.

So when we store a function or pass a function as a parameter, we’re also storing/passing any variables from the outer scope(s) that we might be using in our function.

Object-Oriented Programming

- Combine state and behavior into an “object” which only exposes behavior (“encapsulation”)
  - Send messages (“method calls”) to objects, get output
- Create new, extended objects from existing ones (“inheritance”)
- Sending a message defined for a base type will automatically trigger relevant extended behavior in a derived type (“polymorphism”)

OO Example, Pros/Cons

```
trait Shape {
  def area:Double
}

class Square(side:Double) extends Shape {
  def area:Double = { side * side }
}

class Circle(rad:Double) extends Shape {
  def area:Double = { Math.PI * rad * rad }
}

class Printer(shape:Shape) {
  def printArea() = println("Area is " + shape.area)
}

val a:Shape = new Square(2)
val p:Printer = new Printer(a)
p.printArea()
```

Classes

- Classes can be declared using the class keyword.
- Methods are declared with the def keyword.
- Methods and fields are public by default, but can be specified as protected or private.
- Constructor arguments are, by default, private, but can be preceeded by val to be made public.

```
class A(i: Int, val j: Int) {
  val iPlus5 = i + 5
  private[this] val jPlus5 = j + 5 //instance privacy

  def addTo(k: Int) = new A(i + k, j + k)
  def sum = i + j
}
```

Lab 4: Customer class

- Start by copy/pasting your customers and customer functions from labs 1 and 3 (or from the solutions)

- Refactor the code so it does the same thing but, instead of using a tuple to represent Customer, define and use a Scala class

- Add a method which returns the customer’s first and last names combined as a String

- Add a toString method which returns that same combined name

Inheritance and Traits

Inheritance: Classes are extended using the extends keyword

```
class B(i: Int, k: Int) extends A(i, 4)
```

Traits are like interfaces, but they are allowed to have members declared ("mix-in" members).

```
trait C { def doCThing = "C thing" }

trait D { def doDThing = "D thing" }

class E extends C with D { /* ... */ }
```

Scala's Built-in Collections

The most common collections are Vector, List (similar to Vector), Map, and Set.

Vector[T] is a sequence of items of type T. Elements can be accessed by 0-based index, using parentheses () as the subscript operator:

```
scala> val a = Vector(1,2,3)
a: Vector[Int] = Vector(1, 2, 3)

scala> a(0)
res0: Int = 1
```

A Map[K,V] is an associative array or dictionary type mapping elements of type K to elements of type V. Values can be accessed through their keys.

```
scala> val a = Map(1 -> "one", 2 -> "two", 3 -> "three")
a: Map[Int,String] = Map(1 -> one, 2 -> two, 3 -> three)

scala> a(1)
res2: String = one
```

Type Inference and Empty Collections

- What if Scala doesn’t have enough information to infer a type for a container?

```
val list = List()
list: List[Nothing] = List()
```

We get a fairly useless list! We can fix this two ways…

- `val list:List[String] = List()`

or

- `val list = List.empty[String]`

Collections API

- Key OO methods:

apply(n)	retrieve nth (0-based) element, can omit apply
:+ x	new collection with x appended
x +: coll	new collection with x prepended
contains	test whether collection contains passed element
indexOf	retrieve index of passed element (or -1)
length	get size of collection
slice(from,to)	subsequence up to (not incl) “to”
sorted	sorted copy of collection

- Most of the API is suited to functional programming; we’ll come back to that later!

ScalaDoc Tips

Because of implicit classes, “magic” apply, and other patterns, you may see a function call but have a hard time locating that function in the docs! Here are some tips:

- Note the O and C symbols next to each class name. They permit you to navigate to the documentation for a class (C) or its companion object (O). Implicit functions will often be defined in the companion object.
- Look into RichInt, RichDouble, etc., if you want to know how to work with numeric types. Similarly, for strings, look at StringOps.
- The mathematical functions are in the package scala.math, not in a class.
- Sometimes, you’ll see functions with funny names. For example, in BigInt, there’s a unary_- method. This is how to you define the prefix negation operator -x.
- Methods can take functions as parameters. For instance, the count method in StringOps requires a function that returns true or false for a Char, specifying which characters should be counted: `def count(p: (Char) => Boolean): Int`

Lab 6: Customer Data

- Create a class called CustomerData which will serve as a façade for managing customers
- Internally, choose a Scala container (e.g., List) to hold the actual Customer instances
- Expose the following methods
  - length
  - get (takes an integer, returns the customer at that index)
  - add method (which takes a customer and stores it)
  - find method (which takes a name like “John Smith” and returns a collection of all stored customers with that name)
  - remove method (takes a customer object and removes it from the internal collection). Hint: lookup the ++ method.
  - foreach with the same semantics as in Lab 5

Imports

Classes, objects, and static methods can all be imported.

Underscore can be used as a wildcard to import everything from a particular context.

```
import scala.collection.immutable.BitSet
import scala.math.log
import scala.math._
```

Immutability

Default collections are immutable: if you use a “write” operation on them, they return a new collection.

```
scala> val x = Vector(1,2)
x: scala.collection.immutable.Vector[Int] = Vector(1, 2)

scala> val y = x :+ 3
y: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> x eq y
res22: Boolean = false
```

However, mutable collections are also available:

```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer

scala> val a = ArrayBuffer(1,2)
a: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2)

scala> a += 3
res46: a.type = ArrayBuffer(1, 2, 3) // a is a val; object is mutated in place
```

Functional Basics

- Leverage higher-order functions, referential transparency, immutable data structures, recursion, scope chain for state, etc.
- Ok, what does that mean in practice? …

Procedural iteration

```
// procedural
var i = 0 //mutable state
val data = List("foo", "bar", "baz")

//leak implementation and manually test
state
while (i<data.size) {
  println(data(i)) // redundant ref
  i = i + 1 // manage state
}
```

databricks

Challenges

- Manual state management and conditional logic
  - that is not directly relevant to the task
  - can easily lead to errors
- Mutable state makes thread safety much harder
- Exposing unnecessary internals of data structures
  - Violates encapsulation
  - Makes extension (e.g., parallelization) harder
- Reduces clarity by disguising transformations, burying semantics

Functional Iteration

```
List("foo","bar","baz").foreach(println)
```

- Benefits
  - Omits all state management and control logic
  - Leaves data structure internals hidden
  - Implementation, location, etc. independent
  - Surfaces key semantics (“list – foreach – println”)

Functional Iteration

- Suppose we have a collection of numbers
  - We want to pick out the numbers under 20
  - Calculate their squares
  - Sum the even ones
- Functional style:

```
val data = List(3,4,8,12,15,19)
data.filter(n => n<20)
    .map(n => n*n)
    .filter(n => n%2 == 0)
    .reduce((a,b) => a+b)
```

Same operation using Scala shortcuts:

```
data.filter(_<20).map(n => n*n).filter(_%2 == 0).reduce(_ + _)
```

Lab 7: Functional Iteration

Start with your solution to Lab 6…

- Refactor find, remove, and foreach to use functional methods on the underlying data collection
  - You should be able to remove all of the “for loops”

- Add a total method which returns the total of the account values of the customers

Notice that the “job” of the customer data façade has mostly gone away … we could just use a collection of Customer

Extra credit! Add a method which calculates geometricMean of customer account values, using a functional approach

- Hints: Use map, Math.pow ... Try multiplying using the built in .product method first, then see if you can write it as a .reduce for practice ☺

Collections, Option, and functional iteration

Instead of null (Java), None (Python), etc., Scala uses a parametric type called Option to handle the situation where

- a strongly typed object handle is needed
- but a value may or may not be present

An option …

- May represent a value: Some (value)
- Or no value: None

```
val x = Some("foo"); val y = None
println("x is " + x.getOrElse("no val in x!"))
println("y is " + y.getOrElse("no val in y!"))
```

It is often useful to treat Option as a container:

```
List(Some(3), None, None, Some(20)).flatten
res187: List[Int] = List(3, 20)
```

Iterators

An Iterator[T] is a lazy sequence

- It only evaluates its elements once they are accessed
- Iterators can only be traversed one time

Accidentally traversing the same iterator more than once is a common source of bugs. If you want to be able to access the elements more than once, you can always call .toVector to load the entire thing into memory.

```
val a = Iterator(1,2,3)
val b = a.map(x => x + 1) // stage an operation, but don't traverse yet
val c = b.sum             // c: Int = 9
val d = b.mkString(" ")   // d: String = ""

val e = Iterator(1,2,3)
val f = e.map(x => x + 1) // stage an operation, but don't traverse yet
val g = f.toVector        // g: Vector[Int] = Vector(2, 3, 4)
val i = g.mkString(" ")   // i: String = "2 3 4"
```

Case Classes

Case classes are syntactic sugar for classes with a few methods pre-specified for convenience. They are designed to support structural pattern-matching.

These include toString, equals, hashCode, and static methods apply (so that the new keyword is not needed for construction) and unapply (for pattern matching).

Case class constructor args are public by default. Case classes are not allowed to be extended. Otherwise, they are just like normal classes.

```
case class G(i: Int, j: Int) {
  def sum = i + j
}

val g = G(4, 5)    // g: G = G(4,5)
g.sum              // res19: Int = 9
g == G(4,5)        // res21: Boolean = true
```

Pattern Matching

Allows for succinct code and can be used in a variety of situations.

- Many built-in types have pattern-matching behavior defined
- A main use of pattern matching is in match expressions
- Scala also supports conditional, wildcard, and recursive matching

```
val a = Vector(1,2,3)

val sum = a match {
  case Vector(x,y) => x + y
  case Vector(x,y,z) => x + y + z
}
// sum: Int = 6
```

Pattern Matching

We can match values, types, structure, or combinations of those:

```
val x:Any = … // try (3,4), (30, 4), (30, “four”), (3,4,5)

val out = x match {
  case (3, _) => "I got a 3"
  case (_, s:String) => "ends with " + s
  case (_,_) => "Tuple2!"
  case _ => "default"
}
```

Pattern matching can be combined with recursion to simplify many algorithms.

Lab 8: Case Classes and Option

Starting with your solution to lab 7, make the following changes:

- Change customer to be a case class
- Change CustomerData so that internally it contains a list of Option[Customer] and
  - add takes an Option
  - get(n) returns an Option
  - find, remove, and foreach have the same signatures and semantics as they did before
    - Hint: this will require slight changes in the implementation

More complex for-each

The for-each loop can be used in more complex ways, allowing succinct syntax for looping over multiple collections and filtering:

```
for(
   x <- Vector(1,2,3,4,5);  //outer loop over a vector
   if x % 2 == 1;           //keep odd values xs
   y <- Set(1,2,3);         //inner loop over a list
   if x + y == 6            //keep values that sum to 6
  ) println(s"x=$x, y=$y")  //print with interpolation

// prints:
//   x=3, y=3
//   x=5, y=1
```

For-comprehensions

Using `yield` allows the for-each expression to evaluate to a value, and not just produce side effects:

```
val data = for(
               x <- Vector(1,2,3,4,5);
              if x % 2 == 1;
              y <- Set(1,2,3);
              if x + y == 6
            ) yield x*y

Data:scala.collection.immutable.Vector[Int] = Vector(9, 5)
```

databricks

Implicit Classes

Scala allows you to “add” behavior to existing classes in a principled way using implicit classes.

An implicit class takes exactly one constructor argument that is the type to be extended and defines behavior that should be allowed for that type.

```
implicit class EnhancedVector(xs: Vector[Int]) {
  def sumOfSquares = xs.map(x => x * x).sum
}

Vector(1,2,3).sumOfSquares     // res0: Int = 14
```

When are "." and "()" Optional?

Scala makes no distinction between methods and "operators." You can actually drop the . and () from any 1-argument method:

```
case class A(i: Int) {
  def addTo(a: A) = A(i + a.i)
}
scala> A(5) addTo A(2)
res0: A = A(7)
```

This can sometimes make things more readable:

```
scala> 1 to 5
res1: scala.collection.immutable.Range.Inclusive =
Range(1, 2, 3, 4, 5)
```

One last bit of magic

`apply`

The apply method of a class or object is used to overload the parentheses syntax, allowing you to specify the behavior of what looks like function application.

```
class A(i: Int){
  def apply(j: Int) = i + j
}

val something = new A(3)
something(4)                        // res0: Int = 7
```

databricks

Let's write some code!

databricks™