



Just Enough... Scala for Spark

Databricks Training Team
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

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.

sbt and 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!

Variables

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 (pretty much) exclusively use `val`. If you find yourself wanting to use a `var`, there is almost certainly a better way to structure your code.

Style

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

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
```

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”)

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 parens () 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
```

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
```

Control

Scala has control many familiar control structures.

if-else

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

for-each loop

Basic loop similar to Java for-each

```
val xs = Vector(1,2,3,4,5)
for(x <- xs)
  println(x)
// prints numbers 1 through 5
```


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;           //filter out even xs
  y <- Set(1,2,3);         //inner loop over a list
  if x + y == 6            //filter out entries that
                          //
don't sum to 6
) println(s"x=$x, y=$y")
// 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)
```

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 (often called methods) are defined using the `def` keyword.

- Parameter types must be specified
- Return types are optional: they can be inferred at compile-time (unless the function is recursive)
- Function body should be separated from the signature by an equals sign (unless the return type is `Unit`)
- Braces are not needed around a function body of a single expression
- Parens are not needed in the function signature if there are no params
 - Empty parens means they are optional on the call
 - Function defined without parens, means they are not allowed, so the call looks like a variable access
- The `return` keyword is not needed

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"
}
```

```
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
```


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: Functions

- Create a function which takes an Int and returns the square of that Int
- Create a Vector with 5 Int values in it
- On each item in the Vector, call your squaring function and collect the results in a new Vector
 - Hint: Look up the `:+` operator in the Vector docs
 - Extra credit: Can you do this with recursion instead of a var?
- Now call `.map()` on the original Vector and pass your function as a param
 - Try writing your square function inline in the call to map, as a *lambda expression*

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.

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 preceded 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  
}
```

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 { /* ... */ }
```


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
```

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)
```

Tuples

Scala has Tuple types for 1 through 22 elements.

- In a tuple, each element has its own type,
- and each element can be accessed using the `._n` syntax, where `n` is a 1-based index.

```
scala> val a = (1, "second", 3.4)
a: (Int, String, Double) = (1,second,3.4)
```

```
scala> a._2
res0: String = second
```

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"
```

Lab: Word Count

- Copy some text from the web and assign it to a variable (actually a “val”) in Scala
- Using this text to test out your code as you go, **write a function `wordCount(s:String)` that counts the occurrences of each word and returns a `Map[String, Int]` containing the counts**
- There are *lots* of ways to do this! Try the simplest one you can, first.
 - If you’re used to a language like Java, the simplest approach might be to use a for loop and go from there
- Then, if you have time, see if you can make the code more functional and more compact

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
```

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
```

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
```

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.
- Remember enriched classes.
- 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**

Let's write some code!

