Extracted from:

Programming Scala

Tackle Multi-Core Complexity on the JVM

The Pragmatic Programmers

Programming Scala

Tackle Multi-Core Complexity on the Java Virtual Machine

Venkat Subramaniam

Edited by Daniel H Steinberg

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-31-X

ISBN-13: 978-1-934356-31-9

Printed on acid-free paper.

P1.0 printing, June 2009

Version: 2009-7-7

Chapter 3

Getting Up to Speed in Scala

Scala lets you build on your Java skills. In this chapter, we'll start on familiar ground—with Java code—and then move toward Scala. Scala is similar to Java in several ways and yet different in so many other ways. Scala favors pure object orientation, but it maps types to Java types where possible. Scala supports Java-like imperative coding style and at the same time supports a functional style. Crank up your favorite editor; we are ready to start on a tour through Scala.

3.1 Scala as Concise Java

Scala has very high code density—you type less to achieve more. Let's start with an example of Java code:

Download ScalaForTheJavaEyes/Greetings.java

```
//Java code
public class Greetings {
  public static void main(String[] args) {
    for(int i = 1; i < 4; i++) {
      System.out.print(i + ",");
    }

    System.out.println("Scala Rocks!!!");
  }
}
```

Here's the output:

```
1,2,3,Scala Rocks!!!
```

Scala makes quite a few things in the previous code optional. First, it does not care whether we use semicolons. Second, there is no real

val vs. var

You can define a variable using either a val or a var. The variables defined using val are immutable and can't be changed after initialization. Those defined using var, however, are mutable and can be changed any number of times.

The immutability applies to the variable and not the instance to which the variable refers. For example, if we write val buffer = new StringBuffer(), we can't change what buffer refers to. However, we can modify the instance of StringBuffer using methods like append().

On the other hand, if we define an instance of String using val str = "hello", we can't modify the instance as well because String itself is immutable. You can make an instance of a class immutable by defining all of its fields using val and providing only the methods that let you read, and not modify, the state of the instance.

In Scala, you should prefer using val over var as much as possible since that promotes immutability and functional style.

benefit for the code to live within the class Greetings in a simple example like this, so we can get rid of that. Third, there's no need to specify the type of the variable i. Scala is smart enough to *infer* that i is an integer. Finally, Scala lets us use println without typing System.out.println. Here is the previous code simplified to Scala:

Download ScalaForTheJavaEyes/Greetings.scala

```
for (i <- 1 to 3) {
  print(i + ",")
}

println("Scala Rocks!!!")
```

To run the previous Scala script, type scala Greetings.scala, or run it from within your IDE.

You should see this output:

```
1,2,3,Scala Rocks!!!
```

The Scala loop structure is pretty lightweight. You simply mention that the values of the index i goes from 1 to 3. The left of the arrow (<-) defines a val, not a var (see the sidebar on the current page), and its right side

is a generator expression. On each iteration, a new val is created and initialized with a consecutive element from the generated values.

The range that was generated in the previous code included both the lower bound (1) and the upper bound (3). You can exclude the upper bound from the range via the until() method instead of the to() method:

Download ScalaForTheJavaEyes/GreetingsExclusiveUpper.scala

```
for (i <- 1 until 3) {
  print(i + ",")
}

println("Scala Rocks!!!")
```

You'll see this output:

```
1,2,Scala Rocks!!!
```

Yes, you heard right. I did refer to to() as a *method*. to() and until() are actually methods on RichInt,[1] the type to which Int, which is the inferred type of variable i, is implicitly converted to. They return an instance of Range. So, calling 1 to 3 is equivalent to 1.to(3), but the former is more elegant. We'll discuss more about this charming feature in the sidebar on the next page.

In the previous example, it appears that we've reassigned i as we iterated through the loop. However, i is not a var; it is a val. Each time through the loop we're creating a different val named i. Note that we can't inadvertently change the value of i within the loop because i is immutable. Quietly, we've already taken a step toward functional style here.

We can also perform the loop in a more functional style using foreach():

Download ScalaForTheJavaEyes/GreetingsForEach.scala

```
(1 to 3).foreach(i => print(i + ","))

println("Scala Rocks!!!")
```

Here's the output:

```
1,2,3,Scala Rocks!!!
```

1. We'll discuss rich wrappers in Section 3.2, *Scala Classes for Java Primitives*, on the following page.

The Dot and Parentheses Are Optional

Scala allows you to drop both the dot and the parentheses if a method takes either zero or one parameter. If a method takes more than one parameter, you must use the parentheses, but the dot is still optional. You already saw benefits of this: a + b is really a.+(b), and 1 to 3 is really 1.to(3).

You can take advantage of this lightweight syntax to create code that reads naturally. For example, assume we have a turn() method defined on a class Car:

```
def turn(direction: String) //...
```

We can call the previous method in a lightweight syntax as follows:

```
car turn "right"
```

Enjoy the optional dot and parentheses to reduce code clutter.

The previous example is concise, and there are no assignments. We used the foreach() method of the Range class. This method accepts a function value as a parameter. So, within the parentheses, we're providing a body of code that takes one argument, named in this example as i. The => separates the parameter list on the left from the implementation on the right.

3.2 Scala Classes for Java Primitives

Java presents a split view of the world—there are objects, and then there are primitives such as int, double, and so on. Scala treats everything as objects.

Java treats primitives differently from objects. Since Java 5, autoboxing allows you to send primitives to methods that expect objects. However, Java doesn't let you call a method on a primitive like this: 2.toString().

On the other hand, Scala treats everything as objects. This means you can call methods on literals, just like you can call methods on objects. In the following code, we create an instance of Scala's Int and send it to the ensureCapacity() method of java.util.ArrayList, which expects a Java primitive int.

Download ScalaForTheJavaEyes/ScalaInt.scala

```
class ScalaInt {
  def playWithInt() {
    val capacity : Int = 10
    val list = new java.util.ArrayList[String]
    list.ensureCapacity(capacity)
  }
}
```

In the previous code,[2] Scala quietly treated Scala.Int as the primitive Java int. The result is no performance loss at runtime for type conversions.

There is similar magic that allows you to call methods like to() on Int, as in 1.to(3) or 1 to 3. When Scala determines that Int can't handle your request, Scala quietly applies the intWrapper() method to convert[3] the Int to scala.runtime.RichInt and then invokes the to() method on it.

Classes like RichInt, RichDouble, RichBoolean, and so on, are called *rich wrapper* classes. They provide convenience methods that can be used for classes in Scala that represent the Java primitive types and String.

3.3 Tuples and Multiple Assignments

Suppose we have a function that returns multiple values. For example, let's return a person's first name, last name, and email address. One way to write it in Java is to return an instance of a PersonInfo class that holds the appropriate fields for data we'd like to return. Alternately, we can return a String[] or ArrayList containing these values and iterate over the result to fetch the values. There is a simpler way to do this in Scala. Scala supports tuples and multiple assignments.

A *tuple* is an immutable object sequence created as comma-separated values. For example, the following represents a tuple with three objects: ("Venkat", "Subramaniam", "venkats@agiledeveloper.com").

2. We could have defined val capacity = 10 and let Scala infer the type, but we specified it explicitly to illustrate the compatibility with Java int.
3. We will discuss implicit type conversions in Section 7.5, *Implicit Type Conversions*, on page 101.

We can assign the elements of a tuple into multiple vars or vals in parallel, as shown in this example:

Download ScalaForTheJavaEyes/MultipleAssignment.scala

```
def getPersonInfo(primaryKey : Int) = {
  // Assume primaryKey is used to fetch a person's info...
  // Here response is hard-coded
  ("Venkat", "Subramaniam", "venkats@agiledeveloper.com")
}

val (firstName, lastName, emailAddress) = getPersonInfo(1)

println("First Name is " + firstName)
println("Last Name is " + lastName)
println("Email Address is " + emailAddress)
```

Here's the output from executing this code:

```
First Name is Venkat
Last Name is Subramaniam
Email Address is venkats@agiledeveloper.com
```

What if you try to assign the result of the method to fewer variables or to more variables? Scala will keep an eye out for you and report an error if that happens. This error reporting is at compile time, assuming you're compiling your Scala code and not running it as a script. For example, in the following example, we're assigning the result of the method call to fewer variables than in the tuple:

Download ScalaForTheJavaEyes/MultipleAssignment2.scala

```
def getPersonInfo(primaryKey : Int) = {
  ("Venkat", "Subramaniam", "venkats@agiledeveloper.com")
}

val (firstName, lastName) = getPersonInfo(1)
```

Scala will report this error:

```
(fragment of MultipleAssignment2.scala):5: error:
  constructor cannot be instantiated to expected type;
 found   : (T1, T2)
 required: (java.lang.String, java.lang.String, java.lang.String)
val (firstName, lastName) = getPersonInfo(1)
     ^
...
```

Instead of assigning the values, you can also access individual elements of a tuple. For example, if we execute val info = getPersonInfo(1), then we can access the first element using the syntax info._1, the second element using info._2, and so on.

Tuples are useful not only for multiple assignments. They're useful to pass a list of data values as messages between actors in concurrent programming (and their immutable nature comes in handy here). Their concise syntax helps keep the code on the message sender side very concise. On the receiving side, you can use pattern matching to concisely receive and process the message, as you'll see in Section 9.3, *Matching Tuples and Lists*, on page 120.

3.4 Strings and Multiline Raw Strings

String in Scala is nothing but java.lang.String. You can use String just like the ways you do in Java. However, Scala does provide a few additional conveniences when working with String.

Scala can automatically convert a String to scala.runtime.RichString—this allows you to seamlessly apply some convenience methods like capitalize(), lines(), and reverse.[4]

If you need to create a string that runs multiple lines, it is really simple in Scala. Simply place the multiple lines of strings within three double quotes ("""..."""). That's Scala's support for here documents, or *heredocs*. Here, we create a string that runs three lines long:

Download ScalaForTheJavaEyes/MultiLine.scala

```
val str = """In his famous inaugural speech, John F. Kennedy said
        "And so, my fellow Americans: ask not what your country can do
        for you-ask what you can do for your country." He then proceeded
        to speak to the citizens of the World..."""
println(str)
```

The output is as follows:

```
In his famous inaugural speech, John F. Kennedy said
        "And so, my fellow Americans: ask not what your country can do
        for you-ask what you can do for your country." He then proceeded
        to speak to the citizens of the World...
```

Scala lets you embed double quotes within your strings. Scala took the content within triple double quotes as is, so this is called a *raw string* in Scala. In fact, Scala took the string too literally; we wouldn't want

4. This seamless conversion, however, sometimes may catch you by surprise. For example, "mom".reverse == "mom" evaluates false, since we end up comparing an instance of RichString with an instance of String. "mom".reverse.toString == "mom", however, results in the desired result of true.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Programming Scala's Home Page
http://pragprog.com/titles/vsscala
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/vsscala.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com