Extracted from:

Pragmatic Scala

Create Expressive, Concise, and Scalable Applications

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Scala 2.11

Pragmatic Scala

Create Expressive, Concise, and Scalable Applications

Venkat Subramaniam

edited by Jacquelyn Carter

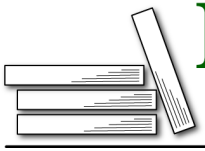
Pragmatic Scala

Create Expressive, Concise, and Scalable Applications

Venkat Subramaniam

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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 *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-054-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2015

CHAPTER 1

Exploring Scala

Scala is a great language for writing highly expressive and concise code without sacrificing the power of static typing.

You can use Scala to build anything from small utility programs to entire enterprise applications. You can program in the familiar object-oriented style, and transition, when you like, to the functional style of programming. Scala does not force developers down a single path; you can start on familiar grounds and, as you get comfortable with the language, make use of features that can help you become more productive and your programs more efficient.

Let's quickly explore some of the features of Scala, and then take a look at a practical example in Scala.

Scala Features

Scala, short for Scalable Language, is a hybrid functional programming language. It was created by Martin Odersky and was first released in 2003 (for more information, see "A Brief History of Scala" in Appendix 2, *Web Resources*, on page ?). Here are some of the key features of Scala:

- It supports both an imperative style and a functional style.
- It is purely object-oriented.
- It enforces sensible static typing and type inference.
- It is concise and expressive.
- It intermixes well with Java.
- It is built on a small kernel.
- It is highly scalable, and it takes less code to create high-performing applications.
- It has a powerful, easy-to-use concurrency model.

You'll learn more about each of these features throughout this book.

More with Less

One of the first differences you'll find as you ease into Scala is that you can do more with a lot less code in Scala than in Java. The conciseness and expressiveness of Scala will shine through every line of code you write. You'll start applying Scala's key features and soon they'll make your routine programing quite productive—Scala simplifies everyday programming.

To get a taste of Scala's power and its benefits, let's look at a quick example where we make use of many of the features. Even though the syntax may appear unfamiliar at this time, key in the code and play with it as you read along. The more you work with the code, the quicker it becomes familiar.

If you've not installed Scala yet, please see Appendix 1, *Installing Scala*, on page ? for the steps. Now to the first code example:

Introduction/TopStock.scala

```
Line 1 val symbols = List("AMD", "AAPL", "AMZN", "IBM", "ORCL", "MSFT")
   2 val year = 2014
   3
   4 val (topStock, topPrice) =
   5   symbols.map { ticker => (ticker, getYearEndClosingPrice(ticker, year)) }
   6          .maxBy { stockPrice => stockPrice._2 }
   7
   8 printf(s"Top stock of $year is $topStock closing at price $$$topPrice")
```

If this is your first look at Scala, don't be distracted by the syntax. Focus on the big picture for now.

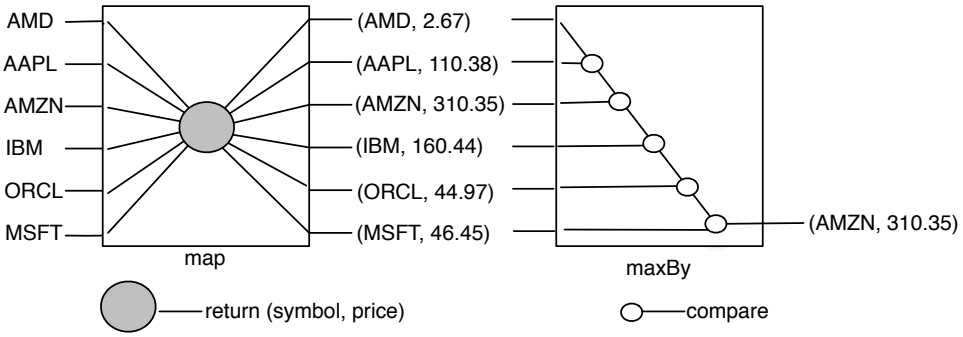
Given a list of symbols, the code computes the highest priced stock among them. Let's tear apart the code to understand it.

Let's look at the main parts of the code first. On line 1, `symbols` refers to an immutable list of stock ticker symbols and, on line 2, `year` is an immutable value. On lines 5 and 6, we use two powerful, specialized iterators—the `map()` function and `maxBy()` function. In Java we're used to the term *method*, to refer to a member of a class. The word *function* is often used to refer to a procedure that's not a member of a class. However, in Scala we use the words *method* and *function* interchangeably.

Each of the two iterators take on two separate responsibilities. First, using the `map()` function, we iterate over the ticker symbols to create another list with pairs or tuples of tickers and their closing price for the year 2014. The resulting list of tuples is of the form `List((symbol1, price1), (symbol2, price2), ...)`.

The second iterator works on the result of the first iterator. `maxBy()` is a specialized iterator on the list that picks the first highest value. Since the list is

a collection of tuples (pairs) of values, we need to tell maxBy() how to compare the values. In the code block attached to maxBy() we indicate that given a tuple, we're interested in its second property (represented by _2): the price value. That was very concise code, but quite a bit is going on there. Let's visualize the actions in the following figure:



As we see in the figure, map() applies the given function or operation—fetching price—for each symbol to create the resulting list of symbols and their prices. maxBy() then works on this subsequent list to create a single result of the symbol with the highest price.

The previous code is missing the getYearEndClosingPrice() function; let's take a look at that next:

Introduction/TopStock.scala

```
def getYearEndClosingPrice(symbol : String, year : Int) = {
  val url = s"http://ichart.finance.yahoo.com/table.csv?s=" +
    s"$symbol&a=11&b=01&c=$year&d=11&e=31&f=$year&g=m"

  val data = io.Source.fromURL(url).mkString
  val price = data.split("\n")(1).split(",")(4).toDouble
  price
}
```

Even though the syntax may not yet be familiar, this code should be easy to read. In this short and sweet function, we send a request to the Yahoo Finance web service and receive the stock data in CSV format. We then parse the data, to extract and return the year-end closing price. Don't worry about the format of the data received right now; that's not important for what we're focusing on here. In Chapter 15, *Creating an Application with Scala*, on page ?, we'll revisit this example and provide all the details about talking to the Yahoo service.

To run the previous example, save the two pieces of code in a file named TopStock.scala and type the command

```
scala TopStock.scala
```

You'll see a result like this:

```
Top stock of 2014 is AMZN closing at price $310.35
```

Spend a few minutes tracing through the code to make sure you understand how this is working. While you're at it, see how the method computed the highest price without ever explicitly changing any variable or object. The entire code is totally dealing with only immutable state; no variable or object was tortured, err…changed, after it was created. As a result, you wouldn't have to be concerned about any synchronization and data contention if you were to run it in parallel.

We've fetched data from the web, done some comparisons, and yielded the desired result—nontrivial work, but it took only a few lines of code. This Scala code will stay concise and expressive even if we ask some more of it. Let's take a look.

In the example, we're fetching data for each symbol from Yahoo, which involves multiple calls over the network. Assume the network delay is `d` seconds and we're interested in analyzing `n` symbols. The sequential code will take about `n * d` seconds. Since the biggest delay in the code will be network access to fetch the data, we can reduce the time to about `d` seconds if we execute the code to fetch data for different symbols in parallel. Scala makes it trivial to turn the sequential code into parallel mode, with only one small change:

```
symbols.par.map { ticker => (ticker, getYearEndClosingPrice(ticker, year)) }
       .maxBy { stockPrice => stockPrice._2 }
```

We inserted a call to `par`; that's pretty much it. Rather than iterating sequentially, the code will now work on each symbol in parallel.

Let's highlight some nice qualities of the example we wrote:

- First, the code is concise. We took advantage of a number of powerful Scala features: function values, (parallel) collections, specialized iterators, values, immutability, and tuples, to mention a few. Of course, I have not introduced any of these yet; we're only in the introduction! So, don't try to understand all of that at this moment, because we have the rest of the book for that.
- We used functional style—function composition, in particular. We transformed the list of symbols to a list of tuples of symbols and their prices

using the map() method. Then we transformed that into the desired value using the maxBy() method. Rather than spending effort on controlling the iteration—as we'd do in imperative style—we ceded control to the library of functions to get the job done.

- We employed concurrency without pain. There was no need for wait() and notify() or synchronized. Since we handled only immutable state, we did not have to spend time or effort (and sleepless nights) with data contention and synchronization.

These benefits have removed a number of burdens from our shoulders. For one, we did not have to struggle to make the code concurrent. For an exhaustive treatise about how painful threads can be, refer to Brian Goetz's *Java Concurrency in Practice [Goe06]*. With Scala, we can focus on application logic instead of worrying about the low-level concerns.

We saw a concurrency benefit of Scala. Scala concurrently (pun intended) provides benefits for single-threaded applications as well. Scala provides the freedom to choose and mix two styles of programming: the imperative style and the assignment-less pure functional style. With the ability to mix these two styles, in Scala we can use the style that's most appropriate in the scope of a single thread. For multithreading or safe concurrency, we would lean toward the functional style.

What we treat as primitives in Java are objects in Scala. For example, 2.toString() will generate a compilation error in Java. However, that is valid in Scala—we're calling the toString() method on an instance of Int. At the same time, in order to provide good performance and interoperability with Java, Scala maps the instances of Int to the int representation at the bytecode level.

Scala compiles down to bytecode. We can run it the same way we run programs written using the Java language or we can also run it as a script. We can also intermix it well with Java. We can extend Java classes from Scala classes, and vice versa. We can also use Java classes in Scala and Scala classes in Java. We can program applications using multiple languages and be a true Polyglot Programmer—see "Polyglot Programming" in Appendix 2, *Web Resources*, on page ?.

Scala is a statically typed language, but, unlike Java, it has sensible static typing. Scala applies type inference in places it can. Instead of specifying the type repeatedly and redundantly, we can rely on the language to learn the type and enforce it through the rest of the code. We don't work for the compiler; instead, we let the compiler work for us. For example, when we define var i = 1, Scala immediately figures that the variable i is of type Int. Now, if we try

to assign a String to that variable as in i = "haha", Scala will give an error, like this:

```
sample.scala:2:
error: type mismatch;
 found   : String("haha")
 required: Int
i = "haha" //Error
    ^
one error found
```

Later in this book we'll see how type inference works beyond such simple definitions and transcends further to function parameters and return values.

Scala favors conciseness. Placing a semicolon at the end of statements is second nature to Java programmers. Scala provides a break for your right pinky finger from the years of abuse it has taken—semicolons are optional in Scala. But, that is only the beginning. In Scala, depending on the context, the dot operator (.) is optional as well, and so are the parentheses. Thus, instead of writing s1.equals(s2);, we can write s1 equals s2. By losing the semicolon, the parentheses, and the dot, code gains a high signal-to-noise ratio. It becomes easier to write domain-specific languages.

One of the most interesting aspects of Scala is *scalability*. We can enjoy a nice interplay of functional programming constructs along with the powerful set of libraries, to create highly scalable, concurrent applications and take full advantage of multithreading on multicore processors.

The real beauty of Scala is in what it does not have. Compared to Java, C#, and C++, the Scala language has a very small kernel of rules built into it. The rest, including operators, are part of the Scala library. This distinction has a far-reaching consequence. Because the language does not do more, we can do a lot more with it. It's truly extensible, and its library serves as a case study for that.

The code in this section showed how much we can get done with only a few lines of code. Part of that conciseness comes from the declarative style of functional programming—let's take a closer look at that next.