Extracted from:

Functional Programming: A PragPub Anthology

Exploring Clojure, Elixir, Haskell, Scala, and Swift

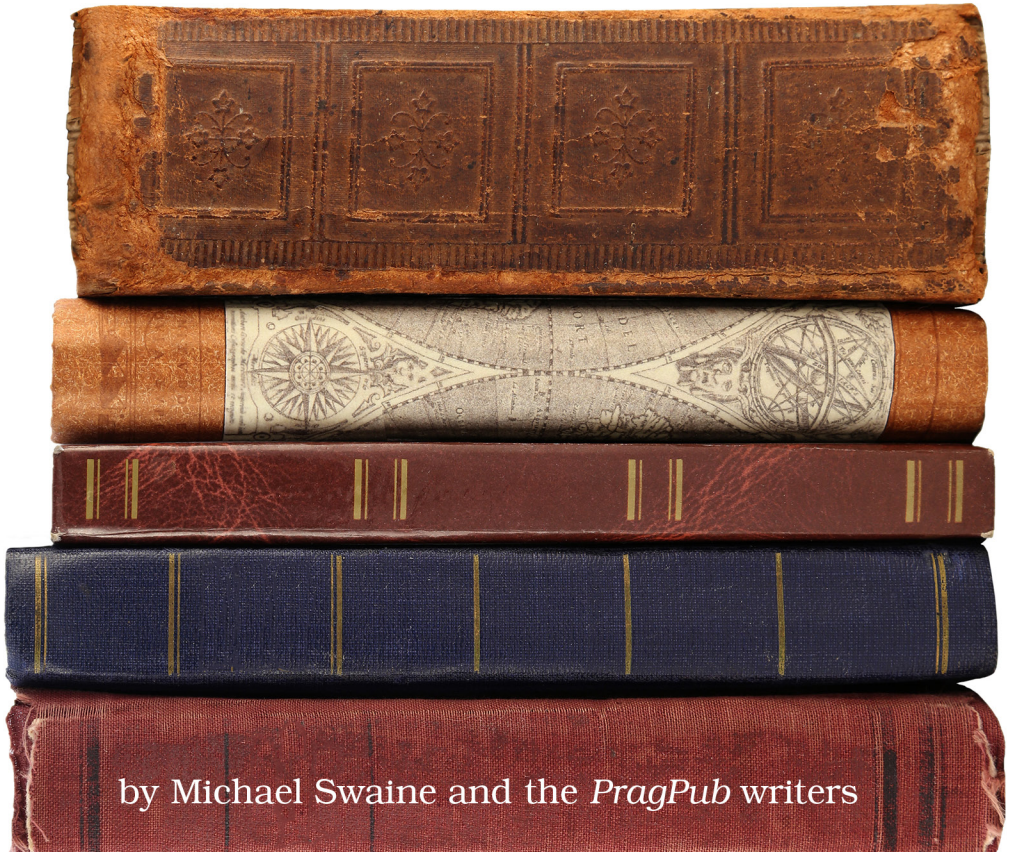
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Functional Programming: A *PragPub* Anthology

Exploring Clojure, Elixir, Haskell, Scala, and Swift



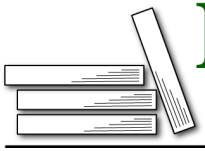
by Michael Swaine and the *PragPub* writers

Functional Programming: A PragPub Anthology

Exploring Clojure, Elixir, Haskell, Scala, and Swift

Michael Swaine
and the PragPub writers

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Indexing: Potomac Indexing, LLC
Copy Editor: Nicole Abramowitz
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

Printed in the United States of America.
ISBN-13: 978-1-68050-233-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2017

CHAPTER 3

Scala and Functional Style

by Venkat Subramaniam

When designing Scala, Martin Odersky took the bold, unconventional step of bringing together two different paradigms: the object-oriented and the functional. This is no trivial undertaking: these two styles are quite dissimilar, and this marriage of distinct paradigms poses some real challenges.

To see what Odersky was up against, let's first take a look at what it means to be functional. There are two defining aspects to the functional style of programming: the purity of functions, and programming with higher-order functions.

Functional Purity

Purity means that functions have no side effects. The output of a function is predictably the same as long as the input is the same. A pure function isn't affected by and doesn't affect anything outside; it also doesn't mutate any value.

There are two benefits of function purity. First, it's easier to understand and prove the correctness of a pure function. Second, pure functions promote referential transparency. Pure functions can be easily rearranged and reordered for execution on multiple threads, making it easier to program concurrency on multicore processors.

Scala does not enforce purity, though it makes it easy to detect where mutable variables are used—simply search for `vars`. It is good Scala practice to use immutability and specifically immutable `vals` as much as possible.

Higher-Order Functions

The other aspect of functional style is working with *higher-order functions*—this is the facility that treats functions as first-class citizens. It lets us pass functions to functions, create functions within functions, and return functions

from functions. This in turn allows for functional composition, and a virtue of Scala is that we can design using both functional composition and object composition, as we desire or as appropriate.

Let's explore Scala's approach to functional programming with some examples. We'll start with some small examples that just manipulate numbers so we can easily grasp how to use higher-order functions in Scala. And then we'll look at a practical example where we apply the concepts we've learned.

A Simple Example

Let's start with a simple iteration: given a list of stock prices, we'd like to print each price on a separate line.

Initially, the traditional for loop comes to mind, something like this in Java:

```
for(int i = 0; i <= prices.size(); i++)
```

Or is it < instead of <= in the loop condition?

There is no reason to burden ourselves with that. We can just use the for-each construct in Java, like so:

```
for(double price : prices)
```

Let's follow this style in Scala:

```
val prices = List(211.10, 310.12, 510.45, 645.60, 832.33)
for(price <- prices) {
  println(price)
}
```

Scala's type inference determines the type of the prices list to be List[Double] and the type of price to be Double. The previous style of iteration is often referred to as an external iterator. Scala also supports internal iteration, so we could write the previous example using the foreach function of List.

```
prices.foreach { e => println(e) }
```

The foreach function is our first example of a higher-order function: it receives another function as a parameter.

Let's reflect on this code for a moment. In general, a function has four parts: name, parameter list, return type, and body. Of these four, the body is the most important. In the previous function that's passed to foreach, the body is between the => and the closing }. The parameter list has one parameter e and since Scala can infer type, we did not have to say e : Double, though we could. Scala already knows the return type of this function based on what

foreach expects and this function is anonymous, so does not have any name. The anonymous function we passed previously is called a function value in Scala.

But Scala has the smarts to allow us to reduce the code even further. If we simply pass the parameter received in the function value to another function in the body, as in this example, we can let Scala do that job:

```
prices.foreach { println }
```

That reduced some noise in the code; the parameter was received in the function value and passed to println.

We can take this up another notch: Scala makes quite a few things optional, including dot, so we can refactor the previous code to

```
prices foreach println
```

Here we're sending the println function itself as a parameter to the foreach method instead of wrapping a call to it into a function value—the println function itself is treated here as a function value.

All right, we know how to use the functional style to iterate over elements: simply pass a function value that operates on an element to the internal iterator and it takes care of calling that function value with each element in the list as a parameter. And this can clearly produce some concise code.

We can explore this further. There are different flavors of internal iterators available on collections in Scala. Let's look at the benefits a few of these offer.

If we want to pick the first price that's greater than $500, we can do that without mutating any variables (functional purity):

```
prices find { price => price > 500 }
```

If we want all the prices that are greater than $500, we just replace find with the filter function.

```
prices filter { price => price > 500 }
```

If we have to compute ten percent of each of the prices given, we can achieve this elegantly using the map function.

```
println(prices map { price => price * 0.1 })
```

will print

```
List(21.11, 31.012, 51.045, 64.56, 83.233)
```

The map function applies the function value given, once for each element in the list, collects the result from the function value (which is ten percent of the price in this example) into a list, and returns the collected list.

Finally, say we're asked to total the prices given. We're quite familiar with how to do that in the imperative style.

```
var total = 0.0
for(price <- prices) {
  total += price
}

println("Total is " + total)
```

to get the output of

```
Total is 2509.6
```

However, a variable was tortured in the making of this example. We can avoid that with functional style again.

```
println("Total is " + prices.reduce { (price1, price2) => price1 + price2 })
//Total is 2509.6
```

The reduce function takes a function value that accepts two values. In the first call to the function value, price1 is bound to the first element in the list and price2 is bound to the second element. In each of the subsequent calls to the function value, price1 is bound to the result of the previous call to this function value and price2 to the subsequent elements in the list. The reduce function returns the result from the last call to the function value once it has been applied for each of the elements in the list.

Scala also provides a specialized version of the reduce function specifically to sum up values—the sum function. Here's how we can use it for the preceding example:

```
println("Total is " + prices.sum)
```

We've seen a few functions here that are typical of functional style: foreach, find, filter, map, and reduce. It's time to put these to a practical use.

A Practical Example

Functional programming emphasizes immutability, but it's equally about designing with state transformation and function composition.

In object-oriented programming, we strive for good object composition. In functional programming, we design with function composition. Rather than mutating state, state is transformed as it flows through the sequence of functions.

Let's construct an example to see what this difference between imperative and functional style looks like in practice. Let's say we're given a list of ticker symbols and our goal is to find the highest-priced stock not exceeding $500.

Let's start with a sample list of ticker symbols.

```
val tickers = List("AAPL", "AMD", "CSCO", "GOOG", "HPQ", "INTC",
    "MSFT", "ORCL", "QCOM", "XRX")
```

For convenience (and to avoid cryptic symbols in code), let's create a case class to represent a stock and its price (case classes are useful to create immutable instances in Scala that provide quite a few benefits, especially ease in pattern matching, a common functional style you'll see explored in depth in Chapter 10, *Patterns and Transformations in Elixir*, on page ?).

```
case class StockPrice(ticker : String, price : Double) {
  def print =
        println("Top stock is " + ticker + " at price $" + price)
}
```

Given a ticker symbol, we want to get the latest stock price for that symbol. Thankfully, Yahoo makes this easy.

```
def getPrice(ticker : String) = {
  val url = s"http://download.finance.yahoo.com/d/quotes.csv?s=${ticker}&f=snbaopl1"
  val data = io.Source.fromURL(url).mkString
  val price = data.split(",")(4).toDouble
  StockPrice(ticker, price)
}
```

We fetch the latest stock price from the Yahoo URL, parse the result, and return an instance of StockPrice with the ticker symbol and the price value.

To help us pick the highest-priced stock valued not over $500, we need two functions: one to compare two stock prices, and the other to determine if a given stock price is not over $500.

```
def pickHighPriced(stockPrice1 : StockPrice, stockPrice2 :
        StockPrice) =
  if(stockPrice1.price > stockPrice2.price) stockPrice1
        else stockPrice2

def isNotOver500(stockPrice : StockPrice) = stockPrice.price < 500
```

Given two StockPrice instances, the pickHighPriced function returns the higher priced. The isNotOver500 will return a true if the price is less than or equal to $500, false otherwise.

Here's how we would approach the problem in imperative style:

```
import scala.collection.mutable.ArrayBuffer

val stockPrices = new ArrayBuffer[StockPrice]
for(ticker <- tickers) {
  stockPrices += getPrice(ticker)
}

val stockPricesLessThan500 = new ArrayBuffer[StockPrice]
for(stockPrice <- stockPrices) {
  if(isNotOver500(stockPrice)) stockPricesLessThan500 += stockPrice
}

var highestPricedStock = StockPrice("", 0.0)

for(stockPrice <- stockPricesLessThan500) {
  highestPricedStock =
        pickHighPriced(highestPricedStock, stockPrice)
}

highestPricedStock print
//Top stock is AAPL at price $377.41
```

Let's walk through the code to see what we did.

First we create an instance of ArrayBuffer, which is a mutable collection. We invoke the getPrice() function for each ticker and populate the stockPrices ArrayBuffer with the StockPrice instances.

Second, we iterate over these stock prices and pick only stocks that are valued less than $500 and add to the stockPricesLessThan500 ArrayBuffer. This results in possibly fewer elements than we started with.

Finally, we loop through the second collection to pick the stock that is valued the highest among them, again mutating the highestPricedStock variable as we navigate the collection using the external iterator.

We can improve on this code further, use multiple collections if we desire, wrap the code into separate functions, and put them into a class if we like. However, that will not affect the fundamental approach we took: imperative style with mutable data structure. The state of the collection of stocks and their prices went through quite a few mutations.

Now let's write this code in functional style. Ready?

```
tickers map getPrice filter isNotOver500 reduce pickHighPriced print
```

We're done. That's it, small enough to fit in a tweet. OK, this conciseness does take some getting used to. Let's walk through it.

tickers map getPrice first transforms the collection of tickers into a collection of StockPrice instances. For each ticker symbol, we now have the name and price in this collection. The filter function then applies the isNotOver500 on that collection and transforms that into a smaller collection of StockPrices with only stocks whose prices do not exceed $500. The reduce function takes that further to pick the highest-priced stock, which we finally hand over to the print method of StockPrice.

In addition to being concise, the code does not mutate any state. The state goes through transformations as it flows through the composed functions.

Granted that this functional code is elegant and concise, but what about other considerations, like ease of debugging and performance? These are two concerns I often see raised.

What About Debugging and Performance?

From the debugging point of view, functional style is a winner. Since there are no mutable states, there are fewer opportunities for errors than in code with several mutable parts. We can write unit tests on each of the intermediate steps separately and also on the collective results. We can step through the code individually or collectively. We can also store the intermediate values in immutable vals along the way so we can examine those.

But what about performance? Surely immutability comes at a cost of performance? Well, if the collection is fairly small, we won't see any significant performance impact, but if the collection is fairly large, we may indeed face some copy overhead, but don't assume. Languages and libraries may offer optimizations and perform lazy evaluations. It would be a mistake to blindly reject the virtues of functional style in the name of performance. Prototype and see if performance is a concern for what you have to implement. If the performance is adequate, then you have gained from what the paradigm has to offer. If performance is not reasonable, then you can explore options to improve it, and modern functional style offers such options. One such is to use data structures that perform intelligent selective copying to provide close to constant-time copy performance, like Vector in Scala.

In the next chapter, we'll see how Scala collections make use of this style to provide a concise and fluent interface.