Functional thinking: Thinking functionally, Part 3

Filtering, unit testing, and techniques for code reuse

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

28 June 2011

Functional thinking series author Neal Ford continues his guided tour of functional programming constructs and paradigms. You'll look at number-classification code in Scala and take a glance at unit testing in the functional world. Then you'll learn about partial application and currying — two functional approaches that facilitate code reuse — and see how recursion fits into the functional way of thinking.

View more content in this series

About this series

This series aims to reorient your perspective toward a functional mindset, helping you look at common problems in new ways and find ways to improve your day-to-day coding. It explores functional programming concepts, frameworks that allow functional programming within the Java language, functional programming languages that run on the JVM, and some future-leaning directions of language design. The series is geared toward developers who know Java and how its abstractions work but have little or no experience using a functional language.

In the first and second installments of *Functional thinking*, I surveyed some functional programming topics and how they relate to Java™ and its related languages. This installment continues this exploration, showing a Scala version of the number classifier from previous installments and discussing some academia-tinged topics such as *currying*, *partial application*, and *recursion*.

Number classifier in Scala

I've been saving a Scala version of the number classifier for last because it's the one with the least syntactic mystery, at least for Java developers. (Recapping the classifier's requirements: Given any positive integer greater than 1, you must classify it as *perfect*, *abundant*, or *deficient*. A perfect number is a number whose factors, excluding the number itself as a factor, add up to the number. An abundant number's sum of factors is greater than the number, and a deficient number's sum of factors is less.) Listing 1 shows the Scala version:

Listing 1. Number classifier in Scala

```
package com.nealford.conf.ft.numberclassifier

object NumberClassifier {

  def isFactor(number: Int, potentialFactor: Int) =
    number % potentialFactor == 0

  def factors(number: Int) =
    (1 to number) filter (number % _ == 0)

  def sum(factors: Seq[Int]) =
    factors.foldLeft(0)(_ + _)

  def isPerfect(number: Int) =
    sum(factors(number)) - number == number

  def isAbundant(number: Int) =
    sum(factors(number)) - number > number

  def isDeficient(number: Int) =
    sum(factors(number)) - number < number
}
```

Even if you've never seen Scala until now, this code should be quite readable. As before, the two methods of interest are `factors()` and `sum()`. The `factors()` method takes the list of numbers from 1 to the target number and applies Scala's built-in `filter()` method, using the code block on the right-hand side as the filtering criterion (otherwise known as a *predicate*). The code block takes advantage of Scala's *implicit parameter*, which allows a placeholder with no name (the `_` character) when a named variable isn't needed. Thanks to Scala's syntactic flexibility, you can call the `filter()` method the same way you call an operator. If you prefer, `(1 to number).filter((number % _ == 0))` also works.

The `sum()` method uses the by-now familiar *fold left* operation (in Scala, implemented as the `foldLeft()` method). I don't need to name the variables in this case, so I use `_` as a placeholder, which leverages the simple, clean syntax for defining a code block. The `foldLeft()` method performs the same task as the similarly named method from the Functional Java library (see Resources), which appeared in the first installment:

1. Take an initial value and combine it via an operation on the first element of the list.
2. Take the result and apply the same operation to the next element.
3. Keep doing this until the list is exhausted.

This is a generalized version of how to apply an operation such as addition to a list of numbers: start with zero, add the first element, take that result and add it to the second, and continue until the list is consumed.

Unit testing

Even though I haven't shown the unit tests for previous versions, all the examples have tests. An effective unit-testing library named ScalaTest is available for Scala (see Resources). Listing 2 shows the first unit test I wrote to verify the `isPerfect()` method from Listing 1:

Listing 2. A unit test for the Scala number classifier

```
@Test def negative_perfection() {
  for (i <- 1 until 10000)
    if (Set(6, 28, 496, 8128).contains(i))
      assertTrue(NumberClassifier.isPerfect(i))
    else
      assertFalse(NumberClassifier.isPerfect(i))
}
```

But like you, I'm trying to learn to think more functionally, and the code in Listing 2 bothered me in two ways. First, it iterates to do something, which exhibits imperative thinking. Second, I don't care for the binary catch-all of the `if` statement. What problem am I trying to solve? I need to make sure that my number classifier doesn't identify an imperfect number as perfect. Listing 3 shows the solution to this problem, stated a little differently:

Listing 3. Alternate test for perfect-number classification

```
@Test def alternate_perfection() {
  assertEquals(List(6, 28, 496, 8128),
               (1 until 10000) filter (NumberClassifier.isPerfect(_)))
}
```

Listing 3 asserts that the only numbers from 1 to 100,000 that are perfect are the ones in the list of known perfect numbers. Thinking functionally extends not just to your code, but to the way you think about testing it as well.

Partial application and currying

The functional approach I've shown to filtering lists is common across functional programming languages and libraries. Using the ability to pass code as a parameter (as to the `filter()` method in Listing 3) illustrates thinking about code reuse in a different way. If you come from a traditional design-patterns-driven object-oriented world, compare this approach to the Template Method design pattern from the Gang of Four *Design Patterns* book (see Resources). The Template Method pattern defines the skeleton of an algorithm in a base class, using abstract methods and overriding to defer individual details to child classes. Using composition, the functional approach allows you to pass functionality to methods that apply that functionality appropriately.

Another way to achieve code reuse is via *currying*. Named after mathematician Haskell Curry (for whom the Haskell programming language is also named), currying transforms a multi-argument function so that it can be called as a chain of single-argument functions. Closely related is *partial application*, a technique for assigning a fixed value to one or more of the arguments to a function, thereby producing another function of smaller *arity* (the number of parameters to the function). To understand the difference, start by looking at the Groovy code in Listing 4, which illustrates currying:

Listing 4. Currying in Groovy

```
def product = { x, y -> return x * y }

def quadrate = product.curry(4)
def octate = product.curry(8)

println "4x4: ${quadrate.call(4)}"
println "5x8: ${octate(5)}"
```

In Listing 4, I define `product` as a code block accepting two parameters. Using Groovy's built-in `curry()` method, I use `product` as the building block for two new code blocks: `quadrate` and `octate`. Groovy makes calling a code block easy: you can either explicitly execute the `call()` method or use the supplied language-level syntactic sugar of placing a set of parentheses containing any parameters after the code-block name (as in `octate(5)`, for example).

Partial application is a broader technique that resembles currying but does not restrict the resulting function to a single argument. Groovy uses the `curry()` method to handle both currying and partial application, as shown in Listing 5:

Listing 5. Partial application vs. currying, both using Groovy's `curry()` method

```
def volume = { h, w, l -> return h * w * l }
def area = volume.curry(1)
def lengthPA = volume.curry(1, 1) //partial application
def lengthC = volume.curry(1).curry(1) // currying

println "The volume of the 2x3x4 rectangular solid is ${volume(2, 3, 4)}"
println "The area of the 3x4 rectangle is ${area(3, 4)}"
println "The length of the 6 line is ${lengthPA(6)}"
println "The length of the 6 line via curried function is ${lengthC(6)}"
```

The `volume` code block in Listing 5 computes the cubic volume of a rectangular solid using the well-known formula. I then create an `area` code block (which computes a rectangle's area) by fixing `volume`'s first dimension (`h`, for height) as `1`. To use `volume` as a building block for a code block that returns the length of a line segment, I can perform either partial application or currying. `lengthPA` uses partial application by fixing each of the first two parameters at `1`. `lengthC` applies currying twice to yield the same result. The difference is subtle, and the end result is the same, but if you use the terms *currying* and *partial application* interchangeably within earshot of a functional programmer, count on being corrected.

Functional programming gives you new, different building blocks to achieve the same goals that imperative languages accomplish with other mechanisms. The relationships among those building blocks are well thought out. Earlier, I showed composition as a code-reuse mechanism. It shouldn't surprise you that you can combine currying and composition. Consider the Groovy code in Listing 6:

Listing 6. Partial-application composition

```
def composite = { f, g, x -> return f(g(x)) }
def thirtyTwoer = composite.curry(quadrate, octate)

println "composition of curried functions yields ${thirtyTwoer(2)}"
```

In Listing 6, I create a `composite` code block that composes two functions. Using that code block, I create a `thirtyTwoer` code block, using partial application to compose the two methods together.

Using partial application and currying, you achieve similar goals to mechanisms like the Template Method design pattern. For example, you can create an `incrementer` code block by building it atop an `adder` code block, as shown in Listing 7:

Listing 7. Different building blocks

```
def adder = { x, y -> return x + y }
def incrementer = adder.curry(1)

println "increment 7: ${incrementer(7)}"
```

Of course, Scala supports currying, as illustrated by the snippet from the Scala documentation shown in Listing 8:

Listing 8. Currying in Scala

```
object CurryTest extends Application {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def dividesBy(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(filter(nums, dividesBy(2)))
  println(filter(nums, dividesBy(3)))
}
```

The code in Listing 8 shows how to implement a `dividesBy()` method used by a `filter()` method. I pass an anonymous function to the `filter()` method, using currying to fix the first parameter of the `dividesBy()` method to the value used to create the code block. When I pass the code block created by passing my target number as a parameter, Scala curries up a new function.

Filtering via recursion

Another topic closely associated with functional programming is *recursion*, which (according to Wikipedia) is the "process of repeating items in a self-similar way." In reality, it's a computer-sciencey way to iterate over things by calling the same method from itself (always carefully ensuring you have an exit condition). Many times, recursion leads to easy-to-understand code because the core of your problem is the need to do the same thing over and over to a diminishing list.

Consider filtering a list. Using an iterative approach, I accept a filtering criterion and loop over the contents, filtering out the elements I don't want. Listing 9 shows a simple implementation of filtering with Groovy:

Listing 9. Filtering in Groovy

```
def filter(list, criteria) {
  def new_list = []
  list.each { i ->
    if (criteria(i))
      new_list << i
  }
  return new_list
}

modBy2 = { n -> n % 2 == 0 }

l = filter(1..20, modBy2)
println l
```

The `filter()` method in Listing 9 accepts a `list` and a `criteria` (a code block specifying how to filter the list) and iterates over the list, adding each item to a new list if it matches the predicate.

Now look back at Listing 8, which is a recursive implementation of the filtering functionality in Scala. It follows a common pattern in functional languages for dealing with a list. One view of a list is that it consists of two parts: the item at the front of the list (the head), and all the other items. Many functional languages have specific methods to iterate over lists using this idiom. The `filter()` method first checks to see if the list is empty — the critically important exit condition for this method. If the list is empty, simply return. Otherwise, use the predicate condition (`p`) passed as a parameter. If this condition is true (meaning I want this item in my list), I return a new list constructed by taking the current head and a filtered remainder of the list. If the predicate condition fails, I return a new list that consists of just the filtered remainder (eliminating the first element). The list-construction operators in Scala make the return conditions for both cases quite readable and easy to understand.

My guess is that you don't use recursion at all now — it's not even a part of your tool box. However, part of the reason lies with the fact that most imperative languages have lackluster support for it, making it more difficult to use than it should be. By adding clean syntax and support, functional languages make recursion a candidate for simple code reuse.

Conclusion

This installment completes my survey of features in the functional-thinking world. Coincidentally, much of this article was about filtering, showing lots of ways to use it and implement it. But this isn't too much of a surprise. Many functional paradigms are built around lists because much of programming boils down to dealing with lists of things. It makes sense to create languages and frameworks that have supercharged facilities for lists.

In the next installment, I'll talk in depth about one of functional programming's building blocks: immutability.

Resources

Learn

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's most recent book discusses tools and practices that help you improve your coding efficiency.
- Scala: Scala is a modern, functional language on the JVM.
- Functional Java: Functional Java is a framework that adds many functional language constructs to Java.
- *The busy Java developer's guide to Scala*: Dig more deeply into Scala in this developerWorks series by Ted Neward.
- "Practically Groovy: Functional programming with curried closures" (Ken Barclay et al., developerWorks, August 2005): Read more about functional programming with Groovy.
- *Design Patterns: Elements of Reusable Object-Oriented Software* (Erich Gamma et al., Addison-Wesley, 1994): The Gang of Four's classic work on design patterns.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- ScalaTest: ScalaTest is a unit-testing library for Scala and Java code.
- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Neal Ford

Neal Ford es un arquitecto de software y Meme Wrangler en **Thought**Works, una consultoría de TI a nivel mundial. También diseña y desarrolla aplicaciones, materiales didácticos, artículos de revistas, cursos y presentaciones de video/DVD, y es el autor o editor de libros que abarcan una variedad de tecnologías, incluyendo las más recientes. *El programador productivo* . Se centra en el diseño y la construcción de aplicaciones empresariales a gran escala.También es un orador de renombre internacional en conferencias de desarrolladores en todo el mundo.Eche un vistazo a suWeb site.