Functional thinking: Thinking functionally, Part 2

Exploring functional programming and control

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

31 May 2011

Functional languages and frameworks let the runtime control mundane coding details such as iteration, concurrency, and state. But that doesn't mean you can't take back control when you need to. One important aspect of thinking functionally is knowing how much control you want to give up, and when.

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 installment of this series, I began discussing some of the characteristics of functional programming, showing how those ideas manifest in both Java and more-functional languages. In this article, I'll continue this tour of concepts by talking about first-class functions, optimizations, and closures. But the underlying theme of this installment is *control*: when you want it, when you need it, and when you should let it go.

First-class functions and control

Using the Functional Java library (see Resources), I last showed the implementation of a number classifier with functional `isFactor()` and `factorsOf()` methods, as shown in Listing 1:

Listing 1. Functional version of the number classifier

```
import fj.F;
import fj.data.List;
import static fj.data.List.range;
import static fj.function.Integers.add;
import static java.lang.Math.round;
```

```
import static java.lang.Math.sqrt;

public class FNumberClassifier {

    public boolean isFactor(int number, int potential_factor) {
        return number % potential_factor == 0;
    }

    public List<Integer> factorsOf(final int number) {
        return range(1, number+1).filter(new F<Integer, Boolean>() {
            public Boolean f(final Integer i) {
                return number % i == 0;
            }
        });
    }

    public int sum(List<Integer> factors) {
        return factors.foldLeft(fj.function.Integers.add, 0);
    }

    public boolean isPerfect(int number) {
        return sum(factorsOf(number)) - number == number;
    }

    public boolean isAbundant(int number) {
        return sum(factorsOf(number)) - number > number;
    }

    public boolean isDeficiend(int number) {
        return sum(factorsOf(number)) - number < number;
    }
}
```

In the `isFactor()` and `factorsOf()` methods, I cede control of the looping algorithm to the framework — it now decides how best to iterate over the range of numbers. If the framework (or — if you choose a functional language such as Clojure or Scala — the language) can optimize the underlying implementation, you automatically benefit. Although you may at first be reluctant to give up this much control, notice that it follows a general trend in programming languages and runtimes: Over time, the developer becomes more abstracted away from details that the platform can handle more efficiently. I never worry about memory management on the JVM because the platform allows me to forget it. Sure, it occasionally makes something more difficult, but it's a good trade-off for the benefit you derive in day-to-day coding. Functional language constructs such as higher-order and first-class functions allow me to climb one more rung up the abstraction ladder and focus more on *what* the code does rather than *how* it does it.

Even with the Functional Java framework, coding in this style in Java is cumbersome because the language doesn't really have syntax and constructs for it. What does functional coding look like in a language that does?

Classifier in Clojure

Clojure is a functional Lisp designed for the JVM (see Resources). Consider the number classifier written in Clojure, shown in Listing 2:

Listing 2. Clojure implementation of the number classifier

```
(ns nealford.perfectnumbers)
```

```
(use '[clojure.contrib.import-static :only (import-static)])
(import-static java.lang.Math sqrt)

(defn is-factor? [factor number]
  (= 0 (rem number factor)))

(defn factors [number]
  (set (for [n (range 1 (inc number)) :when (is-factor? n number)] n)))

(defn sum-factors [number]
    (reduce + (factors number)))

(defn perfect? [number]
  (= number (- (sum-factors number) number)))

(defn abundant? [number]
  (< number (- (sum-factors number) number)))

(defn deficient? [number]
  (> number (- (sum-factors number) number)))
```

Most of the code in Listing 2 is pretty easy to follow even if you aren't a die-hard Lisp developer — especially if you can learn to read inside-out. For example, the `is-factor?` method takes two parameters and asks if the remainder equals 0 when `number` is multiplied by `factor`. Similarly, the `perfect?`, `abundant?`, and `deficient?` methods should be easy to decipher, especially if you refer to the Java implementation in Listing 1.

The `sum-factors` method uses the built-in `reduce` method. `sum-factors` reduces the list one element at a time, using the function (in this case, `+`) supplied as the first parameter on each element. The `reduce` method shows up in different guises in several languages and frameworks; you saw it in Listing 1's Functional Java version as the `foldLeft()` method. The `factors` method returns a list of numbers, so I'm processing the list one at a time, adding each element to the accumulating sum, which is the return value of `reduce`. You can see that once you become accustomed to *thinking* in terms of higher-order and first-class functions, you can reduce (pun intended) a lot of noise in your code.

The `factors` method might seem like a random collection of symbols. But it does make sense once you've seen *list comprehensions*, one of several powerful list-manipulation features in Clojure. As before, it's easiest to understand `factors` from the inside out. Don't become confused by colliding language terminology. The `for` keyword in Clojure does not signify a `for` loop. Rather, think of it as the grandfather of all filtering and transformation constructs. In this case, I'm asking it to filter the range of numbers from 1 to (`number` + 1), using the `is-factor?` predicate (which is the `is-factor` method I defined earlier in Listing 2 — note the heavy use of first-class functions), returning the numbers that match. The return from this operation is a list of numbers that meet my filter criteria, which I coerce into a set to remove duplicates.

Although learning a new language is a hassle, you get lots of bang for the buck from functional languages when you understand their features.

Optimization

One of the benefits of switching to a functional style is the ability to leverage higher-order function support provided by the language or framework. But what about times when you don't want to give

up that control? In my earlier example, I likened the internal behavior of iteration mechanisms to the internal workings of the memory manager: most of the time you're happy not worrying about those details. But occasionally you do care about them, as in the case of optimizations and similar tweaks.

In the two Java versions of the number classifier I showed in "Thinking functionally, Part 1," I optimized the code that determines factors. The original naive implementation used the modulus (%) operator, which is wildly inefficient, to check every number from 2 up to the target number itself to determine if it is a factor. You can optimize the algorithm by noticing that factors come in pairs. If you are looking for the factors of 28, for example, when you find 2 you can grab 14 as well. If you can harvest factors in pairs, you need only check factors up to the square root of the target number.

The optimization that was easy to do in the Java version seems impossible in the Functional Java version because I don't control the implementation of the iteration mechanism directly. But part of learning to think functionally requires surrendering notions about that kind of control, allowing you to exert another kind.

I can restate the original problem functionally: filter all the factors from 1 to `number`, retaining only the factors that match my `isFactor()` predicate. This is implemented in Listing 3:

Listing 3. The `isFactor()` method

```
public List<Integer> factorsOf(final int number) {
    return range(1, number+1).filter(new F<Integer, Boolean>() {
        public Boolean f(final Integer i) {
            return number % i == 0;
        }
    });
}
```

Although elegant from a declarative standpoint, the code in Listing 3 is quite inefficient because it checks every number. Once I understand the optimization (harvesting factors in pairs, only up to the square root), I can restate the problem like this:

1. Filter all of the target number's factors from 1 to the square root of the number.
2. Divide the target number by each of these factors to get the symmetrical factor, and add it to the list of factors.

With this goal in mind, I can write the optimized version of the `factorsOf()` method using the Functional Java library, as shown in Listing 4:

Listing 4. Optimized factors-finding method

```
public List<Integer> factorsOfOptimzied(final int number) {
    List<Integer> factors =
        range(1, (int) round(sqrt(number)+1))
        .filter(new F<Integer, Boolean>() {
            public Boolean f(final Integer i) {
                return number % i == 0;
            }});
    return factors.append(factors.map(new F<Integer, Integer>() {
                                      public Integer f(final Integer i) {
                                          return number / i;
                                      }}))
                                      .nub();
}
```

The code in Listing 4 is based on the algorithm I stated previously, with some funky syntax required by the Functional Java framework. First, I take the range of numbers from 1 to the target number's square root plus 1 (to make sure I catch all the factors). Second, I filter the results based on the use of the modulus operator as in previous versions, wrapped in a Functional Java code block. I save this filtered list in the `factors` variable. Fourth (reading inside out), I take this list of factors and execute the `map()` function, which produces a new list by executing my code block against each element (*mapping* each element onto a new value). My list of factors contains all the factors of my target number up to its square root; I need to divide each by the target number to get its symmetrical factor, which is what the code block sent to the `map()` method does. Fifth, now that I have the list of symmetrical factors, I append it to the original list. As the last step, I must account for the fact that I'm keeping the factors in a `List` instead of a `Set`. `List` methods are convenient for these types of manipulations, but a side-effect of my algorithm is a duplicate entry when a whole-number square root pops up. For example, if the target number is 16, the whole-number root of 4 would end up on the list of factors twice. To continue to use the convenient `List` methods, I need only call its `nub()` method at the end, which removes all duplicates.

Just because you usually surrender knowledge of implementation details when using higher-level abstractions like functional programming doesn't mean that you can't get down and dirty if you must. The Java platform mostly shields you from low-level stuff, but if you are determined, you can burrow down to the level you need. Similarly, in functional programming constructs, you generally willingly cede details to the abstraction, reserving the times you don't to when it really matters.

The dominating visual standout in all the Functional Java code I've shown so far is the block syntax, which uses generics and anonymous inner classes as a kind of pseudo-code-block, closure-type construct. Closures are one of the common features of functional languages. What makes them so useful in this world?

What's so special about closures?

A closure is a function that carries an implicit binding to all the variables referenced within it. In other words, the function (or method) encloses a context around the things it references. Closures are used quite often as a portable execution mechanism in functional languages and frameworks, passed to higher-order functions such as `map()` as the transformation code. Functional Java uses anonymous inner classes to mimic some of "real" closure behavior, but they can't go all the way because Java doesn't have support for closures. But what does that mean?

Listing 5 shows an example of what makes closures so special. It's written in Groovy, which supports closures via its code-block mechanism.

Listing 5. Groovy code illustrating closures

```
def makeCounter() {
  def very_local_variable = 0
  return { return very_local_variable += 1 }
}

c1 = makeCounter()
c1()
c1()
c1()
c2 = makeCounter()

println "C1 = ${c1()}, C2 = ${c2()}"
// output: C1 = 4, C2 = 1
```

The `makeCounter()` method first defines a local variable with an appropriate name, then returns a code block that uses that variable. Notice that the return type for the `makeCounter()` method is a code block, not a value. That code block does nothing but increment the value of the local variable and return it. I've placed explicit `return` calls in this code, both of which are optional in Groovy, but the code is even more cryptic without them!

To exercise the `makeCounter()` method, I assign the code block to a `C1` variable, then call it three times. I'm using Groovy's syntactic sugar to execute a code block, which is to place a set of parentheses adjacent to the code block's variable. Next, I call `makeCounter()` again, assigning a new instance of the code block to `C2`. Last, I execute `C1` again along with `C2`. Note that each of the code blocks has kept track of a separate instance of `very_local_variable`. That's what is meant by *enclosing context.* Even though a local variable is defined within the method, the code block is bound to that variable because it references it, meaning that it must keep track of it while the code block instance is alive.

The closest you could come to the same behavior in Java appears in Listing 6:

Listing 6. `MakeCounter` in Java

```
public class Counter {
    private int varField;

    public Counter(int var) {
        varField = var;
    }

    public static Counter makeCounter() {
        return new Counter(0);
    }

    public int execute() {
        return ++varField;
    }
}
```

Several variants of the `Counter` class are possible, but you're still stuck with managing the state yourself. This illustrates why the use of closures exemplifies functional thinking: allow the runtime

to manage state. Rather than forcing you to handle field creation and babying state (including the horrifying prospect of using your code in a multithreaded environment), let the language or framework invisibly manage that state for you.

We will eventually get closures in an upcoming Java release (a discussion of which is thankfully outside the scope of this article). Their appearance in Java will have two welcome benefits. First, it will greatly simplify framework and library writers' capabilities while improving their syntax. Second, it will provide a low-level common denominator for closure support in all the languages that run on the JVM. Even though lots of JVM languages support closures, they all must implement their own versions, which makes passing closures between languages cumbersome. If the Java language defined a single format, all other languages could leverage it.

Conclusion

Ceding your control over low-level details is a general trend in software development. We've happily given away responsibility for garbage collection, memory management, and hardware differences. Functional programming represents the next abstraction leap: ceding more mundane details such as iteration, concurrency, and state to the runtime as much as possible. This doesn't mean that you can't take control back if you need to — but you have to want it, not have it forced upon you.

In the next installment, I'll continue my exploration of functional programming constructs in Java and close relatives by introducing *currying* and *partial method application*.

Resources

Learn

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's most recent book expands on a number of the topics in this series.
- Scala: Scala is a modern, functional language on the JVM.
- Clojure: Clojure is a modern, functional Lisp that runs on the JVM.
- Podcast: Stuart Halloway on Clojure: Learn more about Clojure and find out the two main reasons why it has been quickly adopted and is rising rapidly in popularity.
- Functional Java: Functional Java is a framework that adds many functional language constructs to Java.
- "Practically Groovy: Metaprogramming with closures, ExpandoMetaClass, and categories" (Scott Davis, developerWorks, June 2009): Read about closures in Groovy.
- 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

- 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 is a software architect and Meme Wrangler at **Thought**Works, a global IT consultancy. He also designs and develops applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *The Productive Programmer*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his Web site.