Functional thinking: Thinking functionally, Part 1

Learning to think like a functional programmer

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

03 May 2011

Functional programming has generated a recent surge of interest with claims of fewer bugs and greater productivity. But many developers have tried and failed to understand what makes functional languages compelling for some types of jobs. Learning the *syntax* of a new language is easy, but learning to *think* in a different way is hard. In the first installment of his *Functional thinking* column series, Neal Ford introduces some functional programming concepts and discusses how to use them in both Java™ and Groovy.

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.

Let's say for a moment that you are a lumberjack. You have the best axe in the forest, which makes you the most productive lumberjack in the camp. Then one day, someone shows up and extols the virtues of a new tree-cutting paradigm, the *chainsaw*. The sales guy is persuasive, so you buy a chainsaw, but you don't know how it works. You try hefting it and swinging at the tree with great force, which is how your other tree-cutting paradigm works. You quickly conclude that this newfangled chainsaw thing is just a fad, and you go back to chopping down trees with your axe. Then, someone comes by and shows you how to crank the chainsaw.

You can probably relate to this story, but with *functional programming* instead of a *chainsaw*. The problem with a completely new programming paradigm isn't learning a new language. After all, language syntax is just details. The tricky part is to learn to *think* in a different way. That's where I come in — chainsaw cranker and functional programmer.

Welcome to *Functional thinking*. This series explores the subject of functional programming but isn't solely about functional programming languages. As I'll illustrate, writing code in a "functional" manner touches on design, trade-offs, different reusable building blocks, and a host of other insights. As much as possible, I'll try to show functional-programming concepts in Java (or close-to-Java languages) and move to other languages to demonstrate capabilities that don't yet exist in the Java language. I won't leap off the deep end and talk about funky things like monads (see Resources) right away (although we'll get there). Instead, I'll gradually show you a new way of thinking about problems (which you're already applying in some places — you just don't realize it yet).

This installment and the next two serve as a whirlwind tour of some subjects related to functional programming, including core concepts. Some of those concepts will come back in much greater detail as I build up more context and nuance throughout the series. As a departure point for the tour, I'll have you look at two different implementations of a problem, one written imperatively and the other with a more functional bent.

Number classifier

To talk about different programming styles, you must have code for comparison. My first example is a variation of a coding problem that appears in my book *The Productive Programmer* (see Resources) and in "Test-driven Design, Part 1" and "Test-driven design, Part 2" (two installments of my previous developerWorks series *Evolutionary architecture and emergent design*). I chose this code at least partially because those two articles describe the code's design in depth. There's nothing wrong with the design extolled in those articles, but I'm going to provide a rationale for a different design here.

The requirements state that, given any positive integer greater than 1, you must classify it as either *perfect*, *abundant*, or *deficient*. A perfect number is a number whose factors (excluding the number itself as a factor) add up to the number. Similarly, an abundant number's sum of factors is greater than the number, and a deficient number's sum of factors is less.

Imperative number classifier

An imperative class that meets these requirements appears in Listing 1:

Listing 1. `NumberClassifier`, the imperative solution to the problem

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {
        if (number < 1)
            throw new InvalidNumberException(
            "Can't classify negative numbers");
        _number = number;
        _factors = new HashSet<Integer>>();
        _factors.add(1);
        _factors.add(_number);
    }

    private boolean isFactor(int factor) {
        return _number % factor == 0;
```

```
    }

    public Set<Integer> getFactors() {
        return _factors;
    }

    private void calculateFactors() {
        for (int i = 1; i <= sqrt(_number) + 1; i++)
            if (isFactor(i))

                addFactor(i);
    }

    private void addFactor(int factor) {
        _factors.add(factor);
        _factors.add(_number / factor);
    }

    private int sumOfFactors() {
        calculateFactors();
        int sum = 0;
        for (int i : _factors)
            sum += i;
        return sum;
    }

    public boolean isPerfect() {
        return sumOfFactors() - _number == _number;
    }

    public boolean isAbundant() {
        return sumOfFactors() - _number > _number;
    }

    public boolean isDeficient() {
        return sumOfFactors() - _number < _number;
    }

    public static boolean isPerfect(int number) {
        return new Classifier6(number).isPerfect();
    }
}
```

Several things about this code are worth noting:

- It has extensive unit tests (in part because I wrote it for a discussion of test-driven development).
- The class consists of a large number of cohesive methods, which is a side-effect of using test-driven development in its construction.
- A performance optimization is embedded in the `calculateFactors()` method. The meat of this class consists of gathering factors so that I can then sum them and ultimately classify them. Factors can always be harvested in pairs. For example, if the number in question is 16, when I grab the factor 2 I can also grab 8 because 2 x 8 = 16. If I harvest factors in pairs, I only need to check for factors up to the square root of the target number, which is precisely what the `calculateFactors()` method does.

(Slightly more) functional classifier

Using the same test-driven development techniques, I created an alternate version of the classifier, which appears in Listing 2:

Listing 2. Slightly more functional number classifier

```
public class NumberClassifier {

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

    static public Set<Integer> factors(int number) {
        HashSet<Integer> factors = new HashSet<Integer>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(number, i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    static public int sum(Set<Integer> factors) {
        Iterator it = factors.iterator();
        int sum = 0;
        while (it.hasNext())
            sum += (Integer) it.next();
        return sum;
    }

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

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

    static public boolean isDeficient(int number) {
        return sum(factors(number)) - number < number;
    }
}
```

The differences between these two versions of the classifier are subtle but important. The primary difference is the purposeful lack of shared state in Listing 2. Elimination (or at least diminishment) of shared state is one of the favored abstractions in functional programming. Instead of sharing state across the methods as intermediate results (see the `factors` field in Listing 1), I call methods directly, eliminating state. From a design standpoint, it makes the `factors()` method longer, but it prevents the `factors` field from "leaking out" of the method. Notice too that the Listing 2 version could consist entirely of static methods. No shared knowledge exists between the methods, so I have less need for encapsulation via scoping. All these methods work perfectly fine if you give them the input parameter types they expect. (This is an example of a *pure function*, a concept I'll investigate further in a future installment.)

Functions

Functional programming is a wide, sprawling area of computer science that has seen a recent explosion of interest. New functional languages on the JVM (such as Scala and Clojure), and frameworks (such as Functional Java and Akka), are on the scene (see Resources), along with the usual claims of fewer bugs, more productivity, better looks, more money, and so on. Rather than try to tackle the entire subject of functional programming out of the gate, I'll focus on several key concepts and follow some interesting implications derived from those concepts.

At the core of functional programming is the *function*, just as classes are the primary abstraction in object-oriented languages. Functions form the building blocks for processing and are imbued with several features not found in traditional imperative languages.

Higher-order functions

Higher-order functions can either take other functions as arguments or return them as results. We don't have this construct in the Java language. The closest you can come is to use a class (frequently an anonymous class) as a "holder" of a method you need to execute. Java has no stand-alone functions (or methods), so they cannot be returned from functions or passed as parameters.

This capability is important in functional languages for at least two reasons. First, having higher-order functions means that you can make assumptions about how language parts will fit together. For example, you can eliminate entire categories of methods on a class hierarchy by building a general mechanism that traverses lists and applies one (or more) higher-order functions to each element. (I'll show you an example of this construct shortly.) Second, by enabling functions as return values, you create the opportunity to build highly dynamic, adaptable systems.

The problems amenable to solution by using higher-order functions aren't unique to functional languages. However, the way you solve the problem differs when you are thinking functionally. Consider the example in Listing 3 (taken from a larger codebase) of a method that does protected data access:

Listing 3. Potentially reusable code template

```
public void addOrderFrom(ShoppingCart cart, String userName,
                     Order order) throws Exception {
    setupDataInfrastructure();
    try {
        add(order, userKeyBasedOn(userName));
        addLineItemsFrom(cart, order.getOrderKey());
        completeTransaction();
    } catch (Exception condition) {
        rollbackTransaction();
        throw condition;
    } finally {
        cleanUp();
    }
}
```

The code in Listing 3 performs initialization, performs some work, completes the transaction if everything was successful, rolls back otherwise, and finally cleans up resources. Clearly, the boilerplate portion of this code could be reused, and we typically do so in object-oriented languages by creating structure. In this case, I'll combine two of the Gang of Four Design Patterns (see Resources): the Template Method and Command patterns. The Template Method pattern suggests that I should move common boilerplate code up the inheritance hierarchy, deferring algorithmic details to child classes. The Command design pattern provides a way to encapsulate behavior in a class with well-known execution semantics. Listing 4 shows the result of applying these two patterns to the code in Listing 3:

Listing 4. Refactored order code

```
public void wrapInTransaction(Command c) throws Exception {
    setupDataInfrastructure();
    try {
        c.execute();
        completeTransaction();
    } catch (Exception condition) {
        rollbackTransaction();
        throw condition;
    } finally {
        cleanUp();
    }
}

public void addOrderFrom(final ShoppingCart cart, final String userName,
                         final Order order) throws Exception {
    wrapInTransaction(new Command() {
        public void execute() {
            add(order, userKeyBasedOn(userName));
            addLineItemsFrom(cart, order.getOrderKey());
        }
    });
}
```

In Listing 4, I extract the generic parts of the code into the `wrapInTransaction()` method (the semantics of which you may recognize — it's basically a simple version of Spring's `TransactionTemplate`), passing a `Command` object as the unit of work. The `addOrderFrom()` method collapses to the definition of an anonymous inner class creation of the command class, wrapping the two work items.

Wrapping the behavior I need in a command class is purely an artifact of the design of Java, which doesn't include any kind of stand-alone behavior. All behavior in Java must reside inside a class. Even the language designers quickly saw a deficiency in this design — in hindsight, it's a bit naive to think that there can never be behavior that isn't attached to a class. JDK 1.1 rectified this deficiency by adding anonymous inner classes, which at least provided syntactic sugar for creating lots of small classes with just a few methods that are purely functional, not structural. For a wildly entertaining and humorous essay about this aspect of Java, check out Steve Yegge's "Execution in the Kingdom of Nouns" (see Resources).

Java forces me to create an instance of a `Command` class, even though all I really want is the method inside the class. The class itself provides no benefits: it has no fields, no constructor (besides the autogenerated one from Java), and no state. It serves purely as a wrapper for the behavior inside the method. In a functional language, this would be handled via a higher-order function instead.

If I am willing to leave the Java language for a moment, I can get semantically close to the functional programming ideal by using closures. Listing 5 shows the same refactored example, but using Groovy (see Resources) instead of Java:

Listing 5. Using Groovy closures instead of command classes

```
def wrapInTransaction(command) {
  setupDataInfrastructure()
  try {
    command()
    completeTransaction()
  } catch (Exception ex) {
    rollbackTransaction()
    throw ex
  } finally {
    cleanUp()
  }
}

def addOrderFrom(cart, userName, order) {
  wrapInTransaction {
    add order, userKeyBasedOn(userName)
    addLineItemsFrom cart, order.getOrderKey()
  }
}
```

In Groovy, anything inside curly braces `{}` is a code block, and code blocks can be passed as parameters, mimicking a higher-order function. Behind the scenes, Groovy is implementing the Command design pattern for you. Each closure block in Groovy is really an instance of a Groovy closure type, which includes a `call()` method that is automatically invoked when you place an empty set of parentheses after the variable holding the closure instance. Groovy has enabled some functional-programming-like behavior by building the appropriate data structures, with corresponding syntactic sugar, into the language itself. As I'll show in future installments, Groovy also includes other functional programming capabilities that go beyond Java. I'll also circle back to some interesting comparisons between closures and higher-order functions in a later installment.

First-class functions

Functions in a functional language are considered *first class*, meaning that functions can appear anywhere that any other language construct (such as variables) can appear. The presence of first-class functions allows uses of functions in unexpected ways and forces thinking about solutions differently, such as applying relatively generic operations (with nuanced details) to standard data structures. This in turn exposes a fundamental shift in thinking in functional languages: *Focus on results, not steps.*

In imperative programming languages, I must think about each atomic step in my algorithm. The code in Listing 1 shows this. To solve the number classifier, I had to discern exactly how to gather factors, which in turn means I had to write the specific code to loop through numbers to determine factors. But looping through lists, performing operations on each element, sounds like a really common thing. Consider the reimplemented number-classification code, using the Functional Java framework, that appears in Listing 6:

Listing 6. Functional number classifier

```
public class FNumberClassifier {

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

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

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

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

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

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

The primary differences between Listing 6 and Listing 2 lie in two methods: `sum()` and `factors()`. The `sum()` method takes advantage of a method on the `List` class in Functional Java, the `foldLeft()` method. This is a specific variation on a list-manipulation concept called a *catamorphism*, which is a generalization on list folding. In this case, a "fold left" means:

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.

Notice that this is exactly what you do when you sum 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. Functional Java supplies the higher-order function (in this example, the `Integers.add` enumeration) and takes care of applying it for you. (Of course, Java doesn't really have higher-order functions, but you can write a good analog if you restrict it to a particular data structure and type.)

The other intriguing method in Listing 6 is `factors()`, which illustrates my "focus on results, not steps" advice. What is the essence of the problem of discovering factors of a number? Stated another way, given a list of all possible numbers up to a target number, how do I determine which ones are factors of the number? This suggests a filtering operation — I can filter the entire list of numbers, eliminating those that don't meet my criteria. The method basically reads like this description: take the range of numbers from 1 to my number (the range is noninclusive, hence the `+1`); filter the list based on the code in the `f()` method, which is Functional Java's way of allowing you to create a class with specific data types; and return values.

This code illustrates a much bigger concept as well, as a trend in programming languages in general. Back in the day, developers had to handle all sorts of annoying stuff like memory allocation, garbage collection, and pointers. Over time, languages have taken on more of that

responsibility. As computers have gotten more powerful, we've offloaded more and more mundane (automatable) tasks to languages and runtimes. As a Java developer, I've gotten quite used to ceding all memory issues to the language. Functional programming is expanding that mandate, encompassing more-specific details. As time goes on, we're going to spend less time worrying about the *steps* required to solve a problem and think more in terms of *processes*. As this series progresses, I'll show lots of examples of this.

Conclusion

Functional programming is more a mindset than a particular set of tools or languages. In this first installment, I started covering some topics in functional programming, ranging from simple design decisions to some ambitious problem rethinking. I rewrote a simple Java class to make it more functional, then began delving into some topics that set functional programming apart from using traditional imperative languages.

Two important, long-ranging concepts first appeared here. First, focus on results, not steps. Functional programming tries to present problems differently because you have different building blocks that foster solutions. The second trend I'll show throughout this series is the offloading of mundane details to programming languages and runtimes, allowing us to focus on the unique aspects of our programming problems. In the next installment, I continue looking at general aspects of functional programming and how it applies to software development today.

Resources

- *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.
- Monads: Monads, a legendarily hairy subject in functional languages, will be covered in a future installment of 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.
- Akka: Akka is a framework for Java that allows sophisticated actor-based concurrency.
- Functional Java: Functional Java is a framework that adds many functional language constructs to Java.
- *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.
- "Execution in the Kingdom of Nouns" (Steve Yegge, March 2006): An entertaining rant about some aspects of Java language design.
- 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 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.