Functional thinking: Tons of transformations

Synonyms hide similarities

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

25 September 2012

Functional programming constructs are appearing in all major languages now, but they can be hard to spot because they're identified by a wide variety of common names. This installment of *Functional thinking* shows the same example written using seven different functional frameworks and languages, investigating similarities and differences.

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.

Functional programming languages approach code reuse differently from object-oriented languages, a topic I investigated in "Coupling and composition, Part 2." Object-oriented languages tend to have many data structures with many operations, whereas functional languages exhibit few data structures with many operations. Object-oriented languages encourage you to create class-specific methods, and you can capture recurring patterns for later reuse. Functional languages help you achieve reuse by encouraging the application of common transformations to data structures, with higher-order functions to customize the operation for specific instances.

The same data structures and operations appear in all functional languages — and in many of the frameworks that support functional programming in Java — but their names often differ. Confused naming makes it difficult to translate knowledge from one language to the other even if the underlying concepts are the same.

This installment's goal is to facilitate that translation. I take a simple problem that requires decision and iteration and implement the solution in five languages (Java, Groovy, Clojure, JRuby,

and Scala) and two functional frameworks (Functional Java and Totally Lazy) for Java (see Resources). The implementation is the same, but the details differ quite a bit across languages.

Plain Java

The problem is to determine if an integer is a prime number — one whose only factors are 1 and itself. Several algorithms for determining prime numbers exist (some alternative solutions appear in "Coupling and composition, Part 1"); I'll use a solution that determines the number's factors, then checks to see if the sum of the factors equals the number plus 1, which indicates that the number is prime. This isn't the most efficient algorithm, but my goal is to show different implementations of common collection methods, not efficiency.

The plain Java version appears in Listing 1:

Listing 1. Plain Java prime-number classifier

```
public class PrimeNumberClassifier {
    private Integer number;

    public PrimeNumberClassifier(int number) {
        this.number = number;
    }

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

    public Set<Integer> getFactors() {
        Set<Integer> factors = new HashSet<Integer>();
        factors.add(1);
        factors.add(number);
        for (Integer i = 2; i < number; i++)
            if (isFactor(i))
                factors.add(i);
        return factors;
    }

    public int sumFactors() {
        int sum = 0;
        for (int i : getFactors())
            sum += i;
        return sum;
    }

    public boolean isPrime() {
        return sumFactors() == number + 1;
    }
}
```

If you've read previous *Functional thinking* installments, you'll recognize the algorithm found in the `getFactors()` method in Listing 1. Its core is the `isFactor()` method, which sifts though the candidate factors.

Groovy

Groovy has added many functional constructs in its evolution, leading to an implementation, shown in Listing 2, that looks very different from the Java one:

Listing 2. Groovy prime-number classifier

```
class PrimeNumberClassifier {
  private int number;

  PrimeNumberClassifier(int number) {
    this.number = number
  }

  public def isFactor(int potential) {
    number % potential == 0;
  }

  public def getFactors() {
    (1..number).findAll { isFactor(it) }.toSet()
  }

  public def sumFactors() {
    getFactors().inject(0, {i, j -> i + j})
  }

  public def isPrime() {
    sumFactors() == number + 1
  }
}
```

Two methods with counterparts in Listing 1 have changed more than just their syntax in Listing 2. The first, `getFactors()`, uses Groovy's `Range` class to represent the candidate numbers. The `findAll()` method applies the code block to each element of the collection, returning a list containing the items for which the block returns `true`. The code block accepts a single parameter, which is the element under inspection. I use convenient Groovy shorthand to make the code block more concise. For example, the code block could be written as `(1..number).findAdd {i-> isFactor(i) }`, but repeating the single parameter is redundant. Groovy offers the option shown in Listing 2 of replacing the lone parameter with the implicit `it`.

The other noteworthy method in Listing 2 is the `sumFactors()` method. Using the set of numbers generated from the `getFactors()` method, I call `inject()`, which is a Groovy collection method that performs a *fold* operation. The `inject()` method combines each element in the collection using the code block supplied in the second parameter, using the first parameter as the initial seed value. The code-block parameter in Listing 2 is `{i, j-> i + j}`, which returns the sum of two numbers. The `inject()` method applies this block, starting at the first element, to each element in turn, summing the list of numbers.

Using functional methods combined with higher-order functions can sometimes lead to dense code. Even though each method in Listing 2 is a single line, it is still beneficial to break them out into individual methods. Separating methods out by function, giving each a meaningful name in the context of the current problem, makes reasoning easier.

Scala

The Scala version of the prime-number classifier appears in Listing 3:

Listing 3. Scala prime-number classifier

```
object PrimeNumberClassifier {
  def isFactor(number: Int, potentialFactor: Int) =
    number % potentialFactor == 0

  def factors(number: Int) =
    (1 to number) filter (isFactor(number, _))

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

  def isPrime(number: Int) =
    sum(factors(number)) == number + 1
}
```

In addition to being much shorter, the Scala version looks different in many other ways. Because I'll only need a single instance, I make it an `object` rather than a `class`. The `factors()` method uses the same implementation as the Groovy version in Listing 2, but with different syntax; I apply the `filter` (Scala's version of Groovy's `findAll()`) method to the number range `(1 to number)`, using the `isFactor()` method defined at the start of Listing 3 as the predicate. Scala allows parameter placeholders as well — the `_` in this case.

The `sum()` method in Listing 3 uses Scala's `foldLeft()` method, which is synonymous with Groovy's `inject()`. In this case, I use zero as the seed value and use placeholders for both parameters.

Clojure

Clojure is a modern implementation of Lisp on the JVM, leading to the strikingly different syntax you see in Listing 4:

Listing 4. Clojure prime-number classifier

```
(ns prime)

(defn factor? [n, potential]
  (zero? (rem n potential)))

(defn factors [n]
  (filter #(factor? n %) (range 1 (+ n 1))))

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

(defn prime? [n]
  (= (inc n) (sum-factors n)))
```

All the methods in Listing 4 look foreign to Java developers, yet the code implements the same algorithm I've been using all along. The `(factor?)` method checks to see if the remainder (the `rem` function in Clojure) is zero. The `(factors)` method uses Clojure's `(filter)` method, which accepts two parameters. The first parameter is the predicate code block to execute on each element, expecting a boolean result as to whether it passes the filter criteria. The `#(factor? n %)` syntax represents a Clojure anonymous function, using Clojure's `%` replacement parameter. The second parameter to the `(filter)` function is the collection to be filtered, which in this case is the range from 1 to my target number plus 1; ranges are exclusive of the last element.

The `(sum-factors)` method in Listing 4 uses Clojure's `(reduce)` method, a synonym for Groovy's `inject()` and Scala's `foldLeft()`. In this case, the operation is the simple `+` operator, which to Clojure is indistinguishable from any other method that accepts two parameters and returns a result.

Although the syntax might be daunting if you aren't accustomed to it, the Clojure version is very concise. Just as in the Groovy version, good function names matter, even if each function is a single line, because the lines sometimes pack quite a punch.

JRuby

JRuby provides a JVM implementation of Ruby, which has also gained many functional constructs over its lifetime. Consider the (J)Ruby version of the prime-number classifier appearing in Listing 5:

Listing 5. JRuby prime-number classifier

```
class PrimeNumberClassifier
  def initialize(num)
    @num = num
  end

  def factor?(potential)
    @num % potential == 0
  end

  def factors
    (1..@num).select { |i| factor?(i) }
  end

  def sum_factors
    factors.reduce(0, :+)
  end

  def prime?
    (sum_factors == @num + 1)
  end
end
```

The `factors` method in Listing 5 adds the `select` synonym for Groovy's `findAll` and both Scala's and Clojure's `filter` method. One of the nice features of JRuby is the easy ability to alias methods, providing more convenient names for method use in different contexts. Sure enough, JRuby includes an alias for the `select` method named `find_all`, but it isn't as common in idiomatic usage.

For the `sum_factors` method in Listing 5, I use JRuby's `reduce` method, mimicking several other languages. In JRuby, as in Clojure, operators are methods with funny names; Ruby allows me to specify the plus method name by its symbol, `:+`. As a readability aid, both Clojure and Ruby allow me to add question marks to functions expected to return boolean values. And, consistent with its nature, Ruby includes an `inject` method alias for `reduce`.

Functional Java

Not to leave out anyone still using variants of Java, several functional programming libraries augment Java with functional constructs. Functional Java is such a framework; the prime-number classifier using Java plus Functional Java appears in Listing 6:

Listing 6. Functional Java prime-number classifier

```
public class FjPrimeNumberClassifier {
    private int number;

    public FjPrimeNumberClassifier(int number) {
        this.number = number;
    }

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

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

    public int sumFactors() {
        return getFactors().foldLeft(fj.function.Integers.add, 0);
    }

    public boolean isPrime() {
        return sumFactors() == number + 1;
    }
}
```

The `getFactors()` method in Listing 6 uses the `filter()` method on `range` (also exclusive, as in Clojure — hence the `number + 1` in the range definition). Because Java doesn't yet have higher-order functions, Functional Java cheats by using anonymous inner-class instances of its built-in `F` class, parameterizing the types using generics.

Like Scala, Functional Java includes a `foldLeft()` method, accepting in this case a predefined code block that sums the numbers and the seed value.

Totally Lazy

Totally Lazy is a functional library for Java that adds a number of *lazy* collections to Java. Lazy data structures don't predefine the elements but rather encode the rules for how to generate a next value whenever it's requested. The prime-number classifier implemented in Totally Lazy appears in Listing 7:

Listing 7. Totally Lazy prime-number classifier

```
public class TlPrimeNumberClassifier {
```

```
    private int number;

    public TlPrimeNumberClassifier(int number) {
        this.number = number;
    }

    public boolean isFactor(Integer potential) {
        return (number % potential) == 0;
    }

    private List<Integer> listOfPotentialFactors() {
        List<Integer> results = new ArrayList<Integer>();
        for (int i = 1; i <= number + 1; i++)
            results.add(i);
        return results;
    }

    public boolean isPrime() {
        return (this.number + 1) ==
            Sequences.init(listOfPotentialFactors())
                     .filter(new Predicate<Integer>() {
                         @Override
                         public boolean matches(Integer other) {
                             return isFactor(other);
                      }})
                     .foldLeft(0, sum())
                     .intValue();
    }
}
```

The `isPrime()` method in Listing 7 uses the `Sequences` class, initialized with a list of all the potential factors (that is, all the numbers from 1 to the target number), then applies its `filter()` method. In Totally Lazy, the `filter()` method expects a subclass of the `Predicate` class, many of which are already implemented for common cases. In my example, I override the `matches()` method, supplying my `isFactor()` method to determine inclusion. After I have the list of factors, I use the `foldLeft` method, using the supplied `sum()` method as the folding operation.

In the example shown in Listing 7, the `isPrime()` method does most of the heavy lifting. The fact that all the data structures in Totally Lazy are lazy sometimes adds complications when you combine them. Consider the version of a `getFactors()` method, shown in Listing 8:

Listing 8. Totally Lazy `getFactors` with a lazy iterator

```
public Iterator<Integer> getFactors() {
    return Sequences
            .init(listOfPotentialFactors())
            .filter(new Predicate<Integer>() {
                @Override
                public boolean matches(Integer other) {
                    return isFactor(other);
                }
            })
            .iterator();
}
```

The return type from the `getFactors()` method in Listing 8 is `Iterator<Integer>` — but it's a lazy iterator, meaning that the collection won't have values until you iterate over it. This makes lazy collections a challenge to test. Consider the unit test for the Totally Lazy example from Listing 8, shown in Listing 9:

Listing 9. Test for Totally Lazy collection

```
@Test
public void factors() {
    TlPrimeNumberClassifier pnc = new TlPrimeNumberClassifier(6);
    List<Integer> expected = new ArrayList<Integer>() {{
        add(1);
        add(2);
        add(3);
        add(6);
    }};
    Iterator<Integer> actual = pnc.getFactors();
    assertTrue(actual.hasNext());
    int count = 0;
    for (int i = 0; actual.hasNext(); i++) {
        assertEquals(expected.get(i), actual.next());
        count++;
    }
    assertTrue(count == expected.size());
}
```

For a lazy collection, I must walk the collection to retrieve the values, then ensure that there aren't more elements in the lazy list than expected.

Although it's possible to write a version in Totally Lazy that is as well-factored as the others, you might find yourself fighting with increasingly byzantine data structures, such as `<Iterator<Iterator<Number>>>`.

Conclusion

In this installment, I demystified the proliferation of names for the same behavior in a variety of functional languages and frameworks. The method names across those languages and frameworks will never reconcile, but as the Java runtime adds constructs such as closures, interoperability will become easier because they can share common standard representations (instead of needing, as in the case of Totally Lazy, awkward constructs like `<Iterator<Iterator<Number>>>`).

In the next installment, I'll continue investigating similarities in transformations by looking at `map`s.

Resources

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's book discusses tools and practices that help you improve your coding efficiency.
- Scala: Scala is a modern, functional language on the JVM.
- Clojure: Clojure is a modern, functional Lisp that runs on the JVM.
- Totally Lazy: Totally Lazy is a framework that adds a variety of lazy collections to Java.
- JRuby: JRuby is an implementation of Ruby on the JVM that offers many functional constructs.
- Groovy: Groovy is a dynamic Java-like language that contains many functional extensions to Java syntax.
- Functional Java: Functional Java is a framework that adds many functional language constructs to Java.
- 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. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

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.