Functional thinking: Functional features in Groovy, Part 2

Metaprogramming + Functional Java

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

20 December 2011

With Groovy, metaprogramming and functional programming form a potent combination. See how metaprogramming enables you to add methods to the `Integer` data type that take advantage of Groovy's built-in functional capabilities. And learn how to use metaprogramming to incorporate the Functional Java™ framework's rich set of functional features seamlessly into 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.

In the last installment, I showed some of Groovy's out-of-the-box functional features and how to use Groovy's primitives to build an infinite list. In this installment, I continue my exploration of the intersection of functional programming and Groovy.

Groovy is a multiparadigm language: it supports object orientation, metaprogramming, and functional programming styles, which are mostly orthogonal to one another (see the Orthogonality sidebar). Metaprogramming allows you to add features to a language and its core libraries. By combining metaprogramming with functional programming, you can make your own code more functional or augment third-party functional libraries to make them work better in Groovy. I'll first show how Groovy's `ExpandoMetaClass` works to augment classes, then how to use this mechanism to weave the Functional Java library (see Resources) into Groovy.

Open classes via `ExpandoMetaClass`

Orthogonality

The definition of *orthogonal* spans several disciplines, including mathematics and computer science. In math, two vectors that are at right angles to each other are orthogonal, meaning they never intersect. In computer science, orthogonal components don't have any effects (or side-effects) on one another. For example, functional programming and metaprogramming are orthogonal in Groovy because they don't interfere with each other: using metaprogramming doesn't restrict you from using functional constructs, and vice versa. The fact that they are orthogonal doesn't mean that they can't *work* together, merely that they don't *interfere* with each other.

One of Groovy's more powerful features is the *open class*, the ability to reopen an existing class to augment or remove its functionality. This is different from subclassing, whereby a new type is derived from an existing one. Open classes allow you to reopen a class such as `String` and add new methods to it. Testing libraries use this capability heavily to augment `Object` with verification methods, so that all classes in an application now have the verification methods.

Groovy has two open-class techniques: categories and `ExpandoMetaClass` (see Resources). Either will work for this example; I chose `ExpandoMetaClass` because it is a bit syntactically simpler.

If you've been following this series, you're familiar with my long-running example of number classification. The complete `Classifier` in Groovy, shown in Listing 1, uses Groovy's own functional constructs:

Listing 1. Complete `Classifier` in Groovy

```
class Classifier {
  def static isFactor(number, potential) {
    number % potential == 0;
  }

  def static factorsOf(number) {
    (1..number).findAll { i -> isFactor(number, i) }
  }

  def static sumOfFactors(number) {
    factorsOf(number).inject(0, {i, j -> i + j})
  }

  def static isPerfect(number) {
    sumOfFactors(number) == 2 * number
  }

  def static isAbundant(number) {
    sumOfFactors(number) > 2 * number
  }

  def static isDeficient(number) {
    sumOfFactors(number) < 2 * number
  }

  static def nextPerfectNumberFrom(n) {
    while (!isPerfect(++n));
    n
  }
}
```

If you have any questions about how the methods are implemented in this version, you can refer to previous installments (in particular, "Coupling and composition, Part 2" and "Functional features in Groovy, Part 1"). To use the methods of this class, I can call the methods in the "normal" functional way: `Classifier.isPerfect(7)`. However, using metaprogramming, I can "wire" these methods directly into the `Integer` class, allowing me to "ask" a number what category it's in.

To add these methods to the `Integer` class, I access the `metaClass` property of the class — predefined by Groovy for each class — as shown in Listing 2:

Listing 2. Adding classification to `Integer`

```
Integer.metaClass.isPerfect = {->
  Classifier.isPerfect(delegate)
}

Integer.metaClass.isAbundant = {->
  Classifier.isAbundant(delegate)
}

Integer.metaClass.isDeficient = {->
  Classifier.isDeficient(delegate)
}
```

Initializing metaprogramming methods

You must add metaprogramming methods before the first attempt to invoke them. The safest place to initialize them is in the static initializer for the class that uses them (because it's guaranteed to run before other initializers for the class), but this adds complexity when multiple classes need augmented methods. Generally, applications that use a lot of metaprogramming end up with a bootstrap class to ensure that initialization occurs at the appropriate time.

In Listing 2, I add the three `Classifier` methods to `Integer`. Now, all integers in Groovy have these methods. (Groovy has no notion of primitive data types; even constants in Groovy use `Integer` as the underlying data type.) Within the code block defining each method, I have access to the predefined `delegate` parameter, which represents the value of the object that is invoking the method on the class.

Once I've initialized my metaprogramming methods (see the Initializing metaprogramming methods sidebar), I can "ask" numbers about categories, as shown in Listing 3:

Listing 3. Using metaprogramming to classify numbers

```
@Test
void metaclass_classifiers() {
  def num = 28
  assertTrue num.isPerfect()
  assertTrue 7.isDeficient()
  assertTrue 6.isPerfect()
  assertTrue 12.isAbundant()
}
```

Listing 3 illustrates the newly added methods working on both variables and constants. It would now be trivial to add a method to `Integer` that returns the classification of a particular number, perhaps as an enumeration.

Adding new methods to existing classes isn't in itself particularly "functional," even if the code they call is strongly functional. However, the ability to add methods seamlessly makes it easy to incorporate third-party libraries — such as the Functional Java library — that add significant functional features. I implemented the number classifier using the Functional Java library in the second installment, and I will use it here to create an infinite stream of perfect numbers.

Mapping data types with metaprogramming

Groovy is essentially a dialect of Java, so pulling in third-party libraries such as Functional Java is trivial. However, I can further weave those libraries into Groovy by performing some metaprogramming mapping between data types to make the seams less visible. Groovy has a native closure type (using the `Closure` class). Functional Java doesn't have the luxury of closures yet (it relies on Java 5 syntax), forcing the authors to use generics and a general `F` class that contains an `f()` method. Using the Groovy `ExpandoMetaClass`, I can resolve the method/closure type differences by creating mapping methods between the two.

The class I want to augment is the `Stream` class from Functional Java, which provides an abstraction for infinite lists. I want to be able to pass Groovy closures in place of Functional Java `F` instances, so I add overloaded methods to the `Stream` class to map closures into `F`'s `f()` method, as shown in Listing 4:

Listing 4. Mapping data types using `ExpandoMetaClass`

```
Stream.metaClass.filter = { c -> delegate.filter(c as fj.F) }
//    Stream.metaClass.filter = { Closure c -> delegate.filter(c as fj.F) }
Stream.metaClass.getAt = { n -> delegate.index(n) }
Stream.metaClass.getAt = { Range r -> r.collect { delegate.index(it) } }
```

The first line creates a `filter()` method on `Stream` that accepts a closure (the `c` parameter of the code block). The second (commented) line is the same as the first, but with the added type declaration for the `Closure`; it doesn't affect how Groovy executes the code but might be preferable as documentation. The body of the code block calls `Stream`'s preexisting `filter()` method, mapping the Groovy closure to the Functional Java `fj.F` class. I use Groovy's semimagical `as` operator to perform the mapping.

Groovy's `as` operator coerces closures into interface definitions, allowing the closure methods to map to methods required by the interface. Consider the code in Listing 5:

Listing 5. Using `as` to create a lightweight iterator

```
def h = [hasNext : { println "hasNext called"; return true},
         next : {println "next called"}] as Iterator

h.hasNext()
h.next()
println "h instanceof Iterator? " + (h instanceof Iterator)
```

In the example in Listing 5, I create a hash with two name-value pairs. Each of the names is a string (Groovy doesn't require hash keys to be delimited with double quotes, because they are strings by default), and the values are code blocks. The `as` operator maps this hash to the

`Iterator` interface, which requires both `hasNext()` and `next()` methods. Once I've performed the mapping, I can treat the hash as an iterator; the last line of the listing prints `true`. In cases in which I have a single-method interface or when I want all the methods in the interface to map to a single closure, I can dispense with the hash and use `as` directly to map a closure onto a function. Referring back to the first line of Listing 4, I map the passed closure to the single-method `F` class. In Listing 4, I must map both `getAt` methods (one that accepts a number and another that accepts a `Range`) because `filter` needs those methods to operate.

Using this newly augmented `Stream`, I can play around with an infinite sequence, as shown in Listing 6:

Listing 6. Using infinite Functional Java streams in Groovy

```
@Test
void adding_methods_to_fj_classes() {

  def evens = Stream.range(0).filter { it % 2 == 0 }
  assertTrue(evens.take(5).asList() == [0, 2, 4, 6, 8])
  assertTrue(evens[3..6] == [6, 8, 10, 12])
}
```

In Listing 6, I create an infinite list of even integers, starting with 0, by filtering them with a closure block. You can't get all of an infinite sequence at once, so you must `take()` as many elements as you want. The remainder of Listing 6 shows testing assertions that demonstrate how the stream works.

Infinite streams in Groovy

In the last installment, I showed how to implement a lazy infinite list in Groovy. Rather than create it by hand, why not rely on an infinite sequence from Functional Java?

To create an infinite `Stream` of perfect numbers, I need two additional `Stream` method mappings to understand Groovy closures, as shown in Listing 7:

Listing 7. Two additional method mappings for perfect-number stream

```
Stream.metaClass.asList = { delegate.toCollection().asList() }
Stream.metaClass.static.cons = { head, closure -> delegate.cons(head, closure as fj.P1) }
// Stream.metaClass.static.cons =
//  { head, Closure c -> delegate.cons(head, ['_1':c] as fj.P1)}
```

In Listing 7, I create an `asList()` conversion method to make it easy to convert a Functional Java stream to a list. The other method I implement is an overloaded `cons()`, which is the method on `Stream` that constructs a new list. When creating an infinite list, the data structure typically contains a first element and a closure block as the tail of the list, which generates the next element when invoked. For my Groovy stream of perfect numbers, I need Functional Java to understand that `cons()` can accept a Groovy closure.

If I use `as` to map a single closure onto an interface that has multiple methods, that closure is executed for any method I call on the interface. That style of simple mapping works in most cases for Functional Java classes. However, a few methods require a `fj.P1` method rather than a `fj.F` method. In some of those cases, I can still get away with a simple mapping because the downstream methods don't rely on any of the other methods of `P1`. In cases in which more precision is required, I may have to use the more complex mapping shown in the commented line in Listing 7, which must create a hash with the `_1()` method mapped to the closure. Although that method looks odd, it's a standard method on the `fj.P1` class that returns the first element.

Once I have my metaprogrammatically mapped methods on `Stream`, I can use the `Classifier` from Listing 1 to create an infinite stream of perfect numbers, as shown in Listing 8:

Listing 8. Infinite stream of perfect numbers using Functional Java and Groovy

```
import static fj.data.Stream.cons
import static com.nealford.ft.metafunctionaljava.Classifier.nextPerfectNumberFrom

def perfectNumbers(num) {
  cons(nextPerfectNumberFrom(num), { perfectNumbers(nextPerfectNumberFrom(num))})
}

@Test
void infinite_stream_of_perfect_nums_using_functional_java() {
  assertEquals([6, 28, 496], perfectNumbers(1).take(3).asList())
}
```

I use static imports both for `cons()` from Functional Java and for my own `nextPerfectNumberFrom()` method from `Classifier` to make the code less verbose. The `perfectNumbers()` method returns an infinite sequence of perfect numbers by consing (yes, *cons* is a verb) the first perfect number after the seed number as the first element and adding a closure block as the second element. The closure block returns the infinite sequence with the next number as the head and the closure to calculate yet another one as the tail. In the test, I generate a stream of perfect numbers starting from 1, taking the next three perfect numbers and asserting that they match the list.

Conclusion

When developers think of metaprogramming, they usually think only of their own code, not of augmenting someone else's. Groovy allows me to add new methods not only to built-in classes like `Integer`, but also to third-party libraries like Functional Java. Combining metaprogramming and functional programming leads to great power with very little code, creating a seamless link.

Although I can call Functional Java classes directly from Groovy, many of the library's building blocks are clumsy when compared to real closures. By using metaprogramming, I can map the Functional Java methods to allow them to understand convenient Groovy data structures, achieving the best of both worlds. Until Java defines a native closure type, developers frequently need to perform these polyglot mappings between language types: a Groovy closure and a Scala closure aren't the same thing at the bytecode level. Having a standard in Java will push these conversations down to the runtime and eliminate the need for mappings like the ones I've shown here. Until that time, though, this facility makes for clean yet powerful code.

In the next installment, I'll talk about some optimizations that functional programming allows your runtime to make and show examples in Groovy of memoization.

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.
- Functional Java: The Functional Java framework adds many functional language constructs to Java.
- "Practically Groovy: Metaprogramming with closures, ExpandoMetaClass, and categories" (Scott Davis, developerWorks, June 2009): Learn more about Groovy metaprogramming in this installment of the *Practically Groovy* series.
- "Language designer's notebook: First, do no harm" (Brian Goetz, developerWorks, July 2011): Read about the design considerations behind *lambda expressions* (closures), a new language feature in the works for Java SE 8.
- 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

- Functional Java: Download the Functional Java framework.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- 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.