Practically Groovy: Metaprogramming with closures, ExpandoMetaClass, and categories

Add methods where you want them, when you want them

Skill Level: Introductory

Scott Davis (scott@thirstyhead.com)
Founder
ThirstyHead.com

23 Jun 2009

Enter into the world of metaprogramming, Groovy-style. The ability to add new methods to classes dynamically at run time — even Java™ classes, and even `final` Java classes — is incredibly powerful. Whether used for production code, unit tests, or anything in between, Groovy's metaprogramming capabilities should pique the curiosity of even the most jaded Java developer.

You've heard people say for years that Groovy is a *dynamic* programming language for the JVM. But what does that really mean? In this *Practically Groovy* installment, you'll learn about *metaprogramming* — Groovy's ability to add new methods to classes dynamically at run time. This flexibility goes far beyond what the standard Java language can offer. Through a series of code examples (all available for download) you'll see that metaprogramming is one of Groovy's most powerful and practical features.

Modeling the world

Our job as programmers is to model the real world in software. When the real world is kind enough to offer up a simple domain — animals with scales or feathers lay eggs, and animals with fur have live births — it's easy to generalize the behavior in software, as shown in Listing 1:

Listing 1. Modeling animals in Groovy

```
class ScalyOrFeatheryAnimal{
  ScalyOrFeatheryAnimal layEgg(){
    return new ScalyOrFeatheryAnimal()
  }
}

class FurryAnimal{
  FurryAnimal giveBirth(){
    return new FurryAnimal()
  }
}
```

About this series

Groovy is a modern programming language that runs on the Java platform. It offers seamless integration with existing Java code while introducing dramatic new features like closures and metaprogramming. Put simply, Groovy is what the Java language would look like had it been written in the 21st century.

The key to incorporating any new tool into your development toolkit is knowing when to use it and when to leave it in the box. Groovy can be extremely powerful, but only when applied properly to appropriate scenarios. To that end, the *Practically Groovy* series explores the practical uses of Groovy, helping you learn when and how to apply them successfully.

Unfortunately, the real world is rife with exceptions and edge cases — duck-billed platypuses are furry and lay eggs. It's almost as if every one of our carefully considered software abstractions is being targeted by a dedicated team of contrarian ninjas.

If the software language you use to model the domain is too rigid to deal with the inevitable exceptions, you can end up sounding like an obstinate civil servant mired in a petty bureaucracy — "I'm sorry, Ms. Platypus, but you're going to have to give birth to live young if you'd like to be tracked by our system."

On the other hand, a dynamic language like Groovy gives you the flexibility to bend your software to model the real world more accurately, rather than presumptuously (and futilely) asking the world for concessions. If the `Platypus` class needs a `layEgg()` method, Groovy makes it possible, as shown in Listing 2:

Listing 2. Adding a layEgg() method dynamically

```
Platypus.metaClass.layEgg = {->
  return new FurryAnimal()
}

def baby = new Platypus().layEgg()
```

If all of this talk of furry animals and eggs seems frivolous, then consider the rigidity

of one of the most often used classes in the Java language: the `String`.

Groovy's new methods on java.lang.String

One of the joys of working with Groovy is all of the new methods that it adds to `java.lang.String`. Methods such as `padRight()` and `reverse()` offer simple `String` transformations, as shown in Listing 3. (For a link to the GDK's list of all of the new methods added to `String`, see Resources. As the GDK cheekily says on its first page, "This document describes the methods added to the JDK to make it more groovy.")

Listing 3. Methods added to String by Groovy

```
println "Introduction".padRight(15, ".")
println "Introduction".reverse()

//output
Introduction...
noitcudortnI
```

But the additions to `String` don't end with simple parlor tricks. If the `String` happens to be a well-formed URL, in a single line you can transform that `String` into a `java.net.URL` and return the results of an HTTP GET request, as shown in Listing 4:

Listing 4. Making an HTTP GET request

```
println "http://thirstyhead.com".toURL().text

//output
<html>
  <head>
    <title>ThirstyHead: Training done right.</title>
<!-- snip -->
```

For another example, running a local shell command is just as easy as making a remote network call. Normally I'd type `ifconfig en0` at the command prompt to check my network card's TCP/IP settings. (If you use Windows instead of Mac OS X or Linux®, try `ipconfig`.) In Groovy, I can do the same thing programmatically, as shown in Listing 5:

Listing 5. Making a shell command in Groovy

```
println "ifconfig en0".execute().text

//output
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether 00:17:f2:cb:bc:6b
```

```
        media: autoselect status: inactive
  //snip
```

I'm not suggesting that the joy of Groovy is that you *can't* do the same things in the Java language. You can. The joy is that these methods have seemingly been added directly onto the `String` class — which is no mean feat, given that `String` is a `final` class. (More on that in just a moment.) Listing 6 shows the `String.execute().text` Java equivalent:

Listing 6. Making a shell command in the Java language

```
Process p = new ProcessBuilder("ifconfig", "en0").start();
BufferedReader br = new BufferedReader(new InputStreamReader(p.getInputStream()));
String line = br.readLine();
while(line != null){
  System.out.println(line);
  line = br.readLine();
}
```

It almost feels like getting bumped from one window to the next at the Department of Motor Vehicles, doesn't it? "I'm sorry, Sir, but in order to see the `String` you requested, you'll first need to stand in that line over there to get a `BufferedReader`."

Yes, you can build convenience methods and utility classes to help abstract that ugliness away, but all of the `com.mycompany.StringUtil` workarounds in the world are a pale substitute for adding the method directly where it belongs — the `String` class. (`Platypus.layEgg()`, indeed!)

So how does Groovy do that, exactly — bolt new methods onto classes that can't be extended or modified directly? To understand, you need to know about *closures* and the `ExpandoMetaClass`.

Closures and the ExpandoMetaClass

Groovy offers an innocuous but powerful language feature — closures — without which the Platypus would never be able to lay an egg. A closure is, quite simply, a named hunk of executable code. It is a method without a surrounding class. Listing 7 demonstrates a simple closure:

Listing 7. A simple closure

```
def shout = {src->
  return src.toUpperCase()
}

println shout("Hello World")
```

```
//output
HELLO WORLD
```

Having free-standing methods is pretty cool, but not nearly as cool as the ability to bolt those methods onto existing classes. Consider the code in Listing 8, where instead of creating a method that accepts a `String` as a parameter, I add the method directly onto the `String` class:

Listing 8. Adding the shout method to String

```
String.metaClass.shout = {->
  return delegate.toUpperCase()
}

println "Hello MetaProgramming".shout()

//output
HELLO METAPROGRAMMING
```

The no-argument `shout()` closure is added to the `String`'s `ExpandoMetaClass` (EMC). Every class — both Java and Groovy — is surrounded by an EMC that intercepts method calls to it. This means that even though the `String` is `final`, methods can be added to its EMC. As a result, it now looks to the casual observer as if `String` has a `shout()` method.

Since this kind of relationship doesn't exist in the Java language, Groovy had to introduce a new concept — *delegates*. The `delegate` is the class that the EMC surrounds.

Knowing that method calls hit the EMC first and the `delegate` second, you can do all sorts of interesting things. For example, notice how Listing 9 actually redefines the `toUpperCase()` method on `String`:

Listing 9. Redefining the toUpperCase() method

```
String.metaClass.shout = {->
  return delegate.toUpperCase()
}

String.metaClass.toUpperCase = {->
  return delegate.toLowerCase()
}

println "Hello MetaProgramming".shout()


//output
hello metaprogramming
```

Again, this might seem frivolous (or even dangerous!). Although there's probably little real-world need to change the behavior of the `toUpperCase()` method, can

you imagine what a boon this is for unit testing your code? Metaprogramming offers a quick and easy way to make potentially random behavior deterministic. For example, Listing 10 demonstrates overriding the static `random()` method of the `Math` class:

Listing 10. Overriding the Math.random() method

```
println "Before metaprogramming"
3.times{
  println Math.random()
}

Math.metaClass.static.random = {->
  return 0.5
}

println "After metaprogramming"
3.times{
  println Math.random()
}

//output
Before metaprogramming
0.3452
0.9412
0.2932
After metaprogramming
0.5
0.5
0.5
```

Now imagine trying to unit test a class that makes an expensive SOAP call. No need to create an interface and stub out an entire mock object — you can strategically override that one method and return a simple mocked-out response. (You'll see examples of using Groovy for unit testing and mocking in the next section.)

Groovy metaprogramming is a run-time phenomenon — it lasts as long as the program is up and running. But what if you want your metaprogramming to be more limited (especially important when writing unit tests)? In the next section, you'll learn how to scope the metaprogramming magic.

Scoping your metaprogramming

Listing 11 wraps the demo code I've been writing in a `GroovyTestCase` so that I can begin testing the output a bit more rigorously. (See "Practically Groovy: Unit test your Java code faster with Groovy" for more on working with `GroovyTestCase`.)

Listing 11. Exploring metaprogramming with a unit test

```
class MetaTest extends GroovyTestCase{

  void testExpandoMetaClass(){
```

```
    String message = "Hello"
    shouldFail(groovy.lang.MissingMethodException){
      message.shout()
    }

    String.metaClass.shout = {->
      delegate.toUpperCase()
    }

    assertEquals "HELLO", message.shout()

    String.metaClass = null
    shouldFail{
      message.shout()
    }
  }
}
```

Type `groovy MetaTest` at the command prompt to run this test.

Notice that you can undo the metaprogramming by simply setting the `String.metaClass` to `null`.

But what if you don't want the `shout()` method to appear on all `Strings`? Well, you can simply tweak the EMC of a single instance instead of the class, as shown in Listing 12:

Listing 12. Metaprogramming a single instance

```
void testInstance(){
  String message = "Hola"
  message.metaClass.shout = {->
    delegate.toUpperCase()
  }

  assertEquals "HOLA", message.shout()
  shouldFail{
    "Adios".shout()
  }
}
```

If you are going to be adding or overriding several methods at once, Listing 13 shows you how to define the new methods in bulk:

Listing 13. Metaprogramming many methods at once

```
void testFile(){
  File f = new File("nonexistent.file")
  f.metaClass{
    exists{-> true}
    getAbsolutePath{-> "/opt/some/dir/${delegate.name}"}
    isFile{-> true}
    getText{-> "This is the text of my file."}
  }

  assertTrue f.exists()
  assertTrue f.isFile()
```

```
  assertEquals "/opt/some/dir/nonexistent.file", f.absolutePath
  assertTrue f.text.startsWith("This is")
}
```

Notice that I no longer care if that file actually exists in the filesystem. I can pass it around to other classes in this unit test, and it will behave as if it were a real file. Once the `f` variable falls out of scope at the end of this test, the custom behavior does as well.

While the `ExpandoMetaClass` is undeniably powerful, Groovy offers a second approach to metaprogramming with its own unique set of capabilities — *categories*.

Categories and the use block

The best way to explain a `Category` is to see it in action. Listing 14 demonstrates using a `Category` to add a `shout()` method onto `String`:

Listing 14. Using a Category for metaprogramming

```
class MetaTest extends GroovyTestCase{
  void testCategory(){
    String message = "Hello"
    use(StringHelper){
      assertEquals "HELLO", message.shout()
      assertEquals "GOODBYE", "goodbye".shout()
    }

    shouldFail{
      message.shout()
      "foo".shout()
    }
  }
}

class StringHelper{
  static String shout(String self){
    return self.toUpperCase()
  }
}
```

If you've ever done any Objective-C development, this technique should look familiar. The `StringHelper Category` is a normal class — it doesn't need to extend a special parent class or implement a special interface. To add new methods to a particular class of type `T`, it only needs to define static methods that accept type `T` as the first parameter. Because `shout()` is a static method that takes in a `String` as the first parameter, all `Strings` wrapped in a `use` block get a `shout()` method.

So, when would you choose a `Category` over an EMC? The EMC allows you to add methods to either a single instance or all instances of a particular class. As you can see, defining a `Category` allows you add methods to *some* instances — only

the instances inside of a `use` block.

While an EMC allows you to define new behavior on the fly, a `Category` allows you to save the behavior off in a separate class file. This means that you can use it in any number of different circumstances — unit tests, production code, and so on. The overhead of defining separate classes is paid back in terms of reusability.

Listing 15 demonstrates using both the `StringHelper` and a newly created `FileHelper` in the same `use` block:

Listing 15. Using several categories in a use block

```
class MetaTest extends GroovyTestCase{
  void testFileWithCategory(){
    File f = new File("iDoNotExist.txt")
    use(FileHelper, StringHelper){
      assertTrue f.exists()
      assertTrue f.isFile()
      assertEquals "/opt/some/dir/iDoNotExist.txt", f.absolutePath
      assertTrue f.text.startsWith("This is")

      assertTrue f.text.shout().startsWith("THIS IS")
    }

    assertFalse f.exists()
    shouldFail(java.io.FileNotFoundException){
      f.text
    }
  }
}


class StringHelper{
  static String shout(String self){
    return self.toUpperCase()
  }
}


class FileHelper{
 static boolean exists(File f){
   return true
 }

 static String getAbsolutePath(File f){
   return "/opt/some/dir/${f.name}"
 }

 static boolean isFile(File f){
   return true
 }

 static String getText(File f){
   return "This is the text of my file."
 }
}
```

But the most interesting aspect of categories is how they are implemented. EMCs require the use of closures, which means that you can only implement them in Groovy. Because categories are nothing more than classes with static methods, they

can be defined in Java code. As a matter of fact, you can reuse existing Java classes — one that were never expressly meant for metaprogramming — in Groovy.

Listing 16 demonstrates using classes from the Jakarta Commons Lang package (see Resources) for metaprogramming. All of the methods in `org.apache.commons.lang.StringUtils` coincidentally follow the `Category` pattern — static methods that accept a `String` as the first parameter. This means that you can use the `StringUtils` class right out of the box as a `Category`.

Listing 16. Using a Java class for metaprogramming

```
import org.apache.commons.lang.StringUtils

class CommonsTest extends GroovyTestCase{
  void testStringUtils(){
    def word = "Introduction"

    word.metaClass.whisper = {->
      delegate.toLowerCase()
    }

    use(StringUtils, StringHelper){
      //from org.apache.commons.lang.StringUtils
      assertEquals "Intro...", word.abbreviate(8)

      //from the StringHelper Category
      assertEquals "INTRODUCTION", word.shout()

      //from the word.metaClass
      assertEquals "introduction", word.whisper()
    }
  }
}

class StringHelper{
  static String shout(String self){
    return self.toUpperCase()
  }
}
```

Type `groovy -cp /jars/commons-lang-2.4.jar:. CommonsTest.groovy` to run the test. (Of course, you need to change the path to where you have the JAR saved on your system.)

Metaprogramming and REST

Just to be sure that you aren't left with the mistaken impression that metaprogramming is useful *only* for unit testing, here is a final example. Recall the RESTful Yahoo! Web service for current weather conditions discussed in "Practically Groovy: Building, parsing, and slurping XML." Combine the `XmlSlurper` skills you picked up in that article with the metaprogramming skills you picked up in this one, and you can check the weather for any ZIP code in 10 lines of code, as shown in Listing 17:

Listing 17. Adding a weather method

```
String.metaClass.weather={->
  if(!delegate.isInteger()){
    return "The weather() method only works with zip codes like '90201'"
  }
  def addr = "http://weather.yahooapis.com/forecastrss?p=${delegate}"
  def rss = new XmlSlurper().parse(addr)
  def results = rss.channel.item.title
  results << "\n" + rss.channel.item.condition.@text
  results << "\nTemp: " + rss.channel.item.condition.@temp
}

println "80020".weather()


//output
Conditions for Broomfield, CO at 1:57 pm MDT
Mostly Cloudy
Temp: 72
```

As you can see, metaprogramming is all about extreme flexibility. You can use any (or all) of the techniques outlined in this article to add methods easily to one, some, or all of the classes you want.

Conclusion

Asking the world to constrain itself to the arbitrary limitations of your language simply isn't a realistic option. Modeling the real world in software means that you need a tool flexible enough to handle all of the edge cases. Thankfully, with Groovy's closures, `ExpandoMetaClasses`, and categories, you have a razor-sharp set of tools to add behavior where you need it and when you need it.

Next time, I'll revisit the power of Groovy for unit testing. There are real benefits to writing your tests in Groovy, whether it is a `GroovyTestCase` or a JUnit 4.x test case with annotations. You'll also see GMock in action — a mocking framework written in Groovy. Until then, I hope that you find plenty of practical uses for Groovy.

Downloads

Description	Name	Size	Download method
Source code for this article's examples	j-pg06239.zip	7KB	HTTP

Information about download methods

Resources

Learn

- Groovy: Learn more about Groovy at the project Web site.
- `String`: See the Groovy JDK API Specification for all of the methods Groovy adds to the `String` class.
- AboutGroovy.com: Keep up with the latest Groovy news and article links.
- *Groovy Recipes* (Scott Davis, Pragmatic Programmers, 2008): Learn more about Groovy and Grails in Scott Davis' latest book.
- *Mastering Grails*: Scott Davis's companion series focuses on this Groovy-based platform for Web development.
- 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

- Jakarta Commons Lang: These helper utilities for the `java.lang` API include classes you can use for metaprogramming with Groovy.
- Groovy: Download the latest Groovy ZIP file or tarball.

Discuss

- Groovy mailing list: Browse, search, or subscribe to the Groovy mailing list.
- Get involved in the My developerWorks community.

About the author

Scott Davis
Scott Davis is an internationally recognized author, speaker, and software developer. He is the founder of ThirstyHead.com, a Groovy and Grails training company. His books include *Groovy Recipes: Greasing the Wheels of Java*, *GIS for Web Developers: Adding Where to Your Application*, *The Google Maps API*, and *JBoss At Work*. He writes two ongoing article series for IBM developerWorks: *Mastering Grails* and *Practically Groovy*.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.