Extracted from:

Programming Groovy 2

Dynamic Productivity for the Java Developer

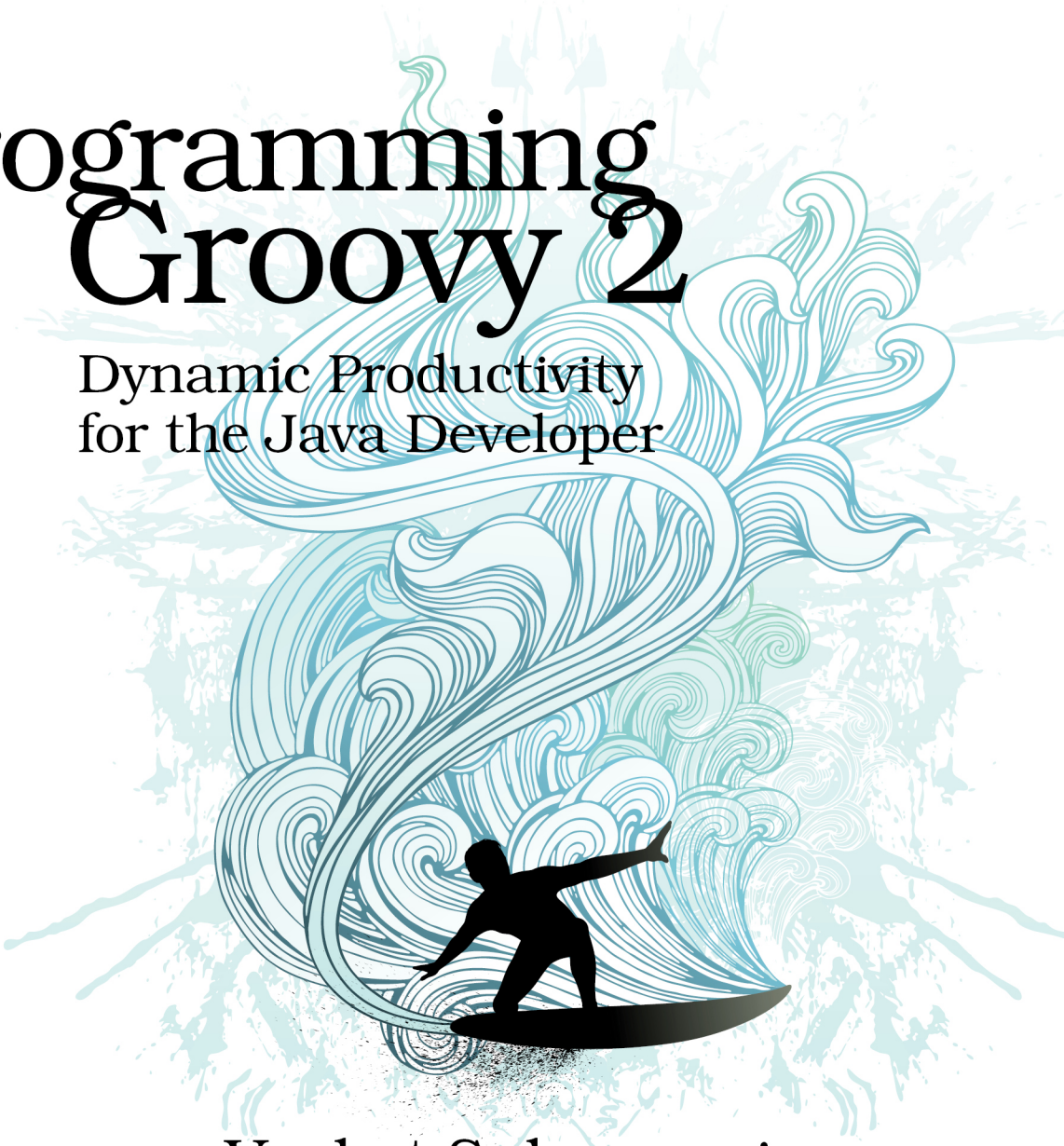
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Groovy 2

Dynamic Productivity for the Java Developer



Venkat Subramaniam

Foreword by Guillaume Laforge

Edited by Brian P. Hogan

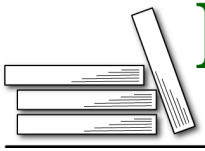
Programming Groovy 2

Dynamic Productivity for the Java Developer

Venkat Subramaniam

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-30-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2013

To Mythili and Balu—for being much more
than an aunt and an uncle—for being there
when I needed them most.

In Java, we can use reflection at runtime to explore our program's structure, plus its classes, their methods, and the parameters they take. However, we're still restricted to the static structure we've created. We can't change an object's type or let it acquire behavior dynamically at runtime—at least not yet. Imagine if we could add methods and behavior dynamically based on the current state of our application or the inputs it receives. This would make our code flexible, and we could be creative and productive. Well, we don't have to imagine that anymore—metaprogramming provides this functionality in Groovy.

How extensible can we design applications to be with these features? Quite. I recently had the opportunity to consult with a company that transitioned from creating Java-based web applications to using Groovy and Grails. Their product required certain customization in the field after deployment. In their existing system, this took them weeks of effort and the time of a few programmers and testers. Working closely with their key developers, we managed to automate the customization using Groovy metaprogramming and some backend services. Immediately, the organization realized higher throughput and productivity.

Metaprogramming means writing programs that manipulate programs, including themselves. Dynamic languages such as Groovy provide this capability through the metaobject protocol (MOP). Creating classes, writing unit tests, and introducing mock objects are all easy with Groovy's MOP.

In Groovy, we can use MOP to invoke methods dynamically and synthesize classes and methods on the fly. This can give us the feeling that our object favorably changed its class. Grails/GORM uses this facility, for example, to synthesize methods for database queries. With MOP we can create internal domain-specific languages (DSLs) in Groovy (see Chapter 19, *Creating DSLs in Groovy*, on page ?). Groovy builders (see Chapter 17, *Groovy Builders*, on page ?) rely on MOP as well. So, MOP is one of the most important concepts to learn and exploit. We'll investigate several concepts in MOP across this and the next few chapters.

In this chapter, we will explore MOP by looking at what makes a Groovy object and how Groovy resolves method calls for Java objects and Groovy objects. We'll then look at ways to query for methods and properties and, finally, see how to access objects dynamically.

Once you've absorbed the fundamentals in this chapter, you'll be ready to learn how to intercept method calls in Chapter 12, *Intercepting Methods Using MOP*, on page ?. We'll then look at how to inject and synthesize methods

into classes at runtime in Chapter 13, *MOP Method Injection*, on page ?, and Chapter 14, *MOP Method Synthesis*, on page ?. Finally, we'll wrap up the discussion on MOP in Chapter 15, *MOPping Up*, on page ?.

11.1 Groovy Object

The flexibility Groovy offers can be confusing at first, so if we want to take full advantage of MOP, we need to understand Groovy objects and Groovy's method handling.

Groovy objects are Java objects with additional capabilities. Groovy objects have a greater number of dynamic behaviors than do compiled Java objects in Groovy. Also, Groovy handles method calls to Java objects differently than to Groovy objects.

In a Groovy application we'll work with three kinds of objects: POJOs, POGOs, and Groovy interceptors. Plain old Java objects (POJOs) are regular Java objects—we can create them using Java or other languages on the Java Virtual Machine (JVM). Plain old Groovy objects (POGOs) are classes written in Groovy. They extend java.lang.Object but implement the groovy.lang.GroovyObject interface. Groovy interceptors are Groovy objects that extend GroovyInterceptable and have a method-interception capability, which we'll soon discuss. Groovy defines the GroovyObject interface like this:

```
//This is an excerpt of GroovyObject.java from Groovy source code
package groovy.lang;
public interface GroovyObject {
    Object invokeMethod(String name, Object args);
    Object getProperty(String property);
    void setProperty(String property, Object newValue);
    MetaClass getMetaClass();
    void setMetaClass(MetaClass metaClass);
}
```

invokeMethod(), getProperty(), and setProperty() make Groovy objects highly dynamic. We can use them to work with methods and properties created on the fly. getMetaClass() and setMetaClass() make it very easy to create proxies to intercept method calls on POGOs, as well as to inject methods on POGOs. Once a class is loaded into the JVM, we can't change the metaobject Class for it. However, we can change its MetaClass by calling setMetaClass(). This gives us a feeling that the object changed its class at runtime.

Let's look at the GroovyInterceptable interface next. It's a marker interface that extends GroovyObject, and all method calls—both existing methods and nonexistent methods—on an object that implements this interface are intercepted by its invokeMethod().

```
//This is an excerpt of GroovyInterceptable.java from Groovy source code

package groovy.lang;

public interface GroovyInterceptable extends GroovyObject {
}
```

Groovy allows metaprogramming for POJOs and POGOs. For POJOs, Groovy maintains a MetaClassRegistry class of MetaClasses, as the following figure shows. POGOs, on the other hand, have a direct reference to their MetaClass.

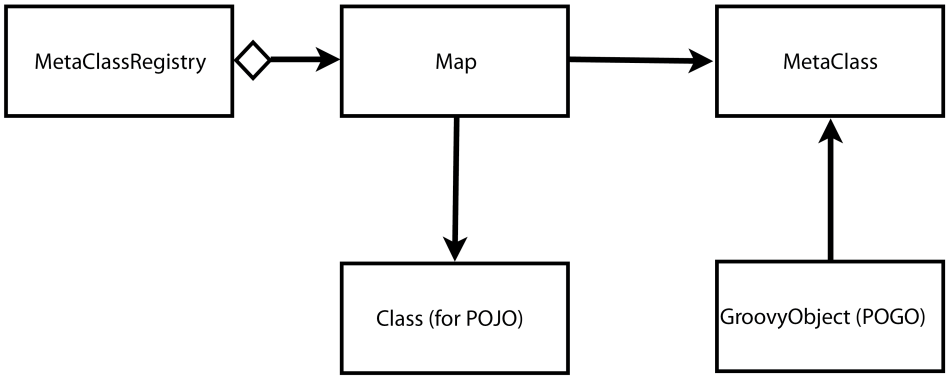


Figure 10—POJOs, POGOs, and their MetaClass

When we call a method, Groovy checks whether the target object is a POJO or a POGO. Groovy's method handling is different for each of these types.

For a POJO, Groovy fetches its MetaClass from the application-wide MetaClassRegistry and delegates method invocation to it. So, any interceptors or methods we've defined on its MetaClass take precedence over the POJO's original method.

For a POGO, Groovy takes a few extra steps, as illustrated in the following figure. If the object implements GroovyInterceptable, then *all* calls are routed to its invokeMethod(). Within this interceptor, we can route calls to the actual method, making aspect-oriented-programming–like operations possible.

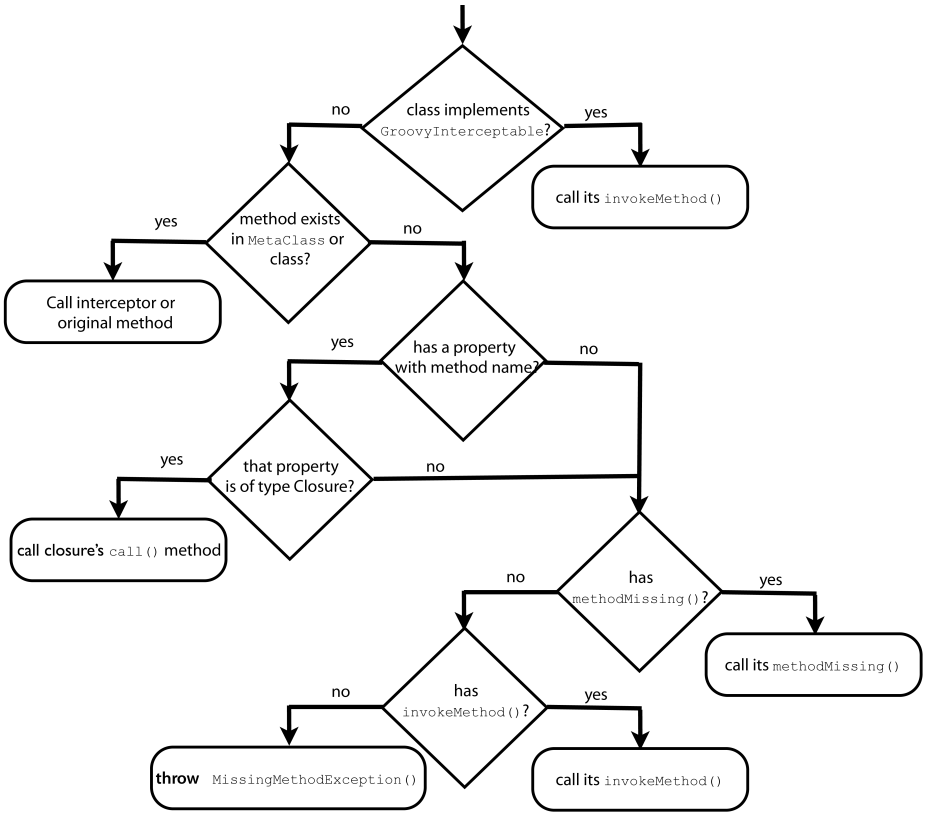


Figure 11—How Groovy handles method calls on a POGO

If the POGO does not implement GroovyInterceptable, then Groovy looks for the method first in the POGO's MetaClass and then, if it's not found, on the POGO itself. If the POGO has no such method, Groovy looks for a property or a field with the method name. If that property or field is of type Closure, Groovy invokes that in place of the method call. If Groovy finds no such property or field, it makes two last attempts. If the POGO has a method named methodMissing(), it calls it. Otherwise, it calls the POGO's invokeMethod(). If we've implemented this method on our POGO, it's used. The default implementation of invokeMethod() throws a MissingMethodException, indicating the failure of the call.

Let's see in code the mechanism discussed earlier, using classes with different options to illustrate Groovy's method handling. Study the code, and try to figure out which methods Groovy executes in each of the cases (while walking through the following code, refer to Figure 11, *How Groovy handles method calls on a POGO*, on page 10):

ExploringMOP/TestMethodInvocation.groovy

```
class TestMethodInvocation extends GroovyTestCase {
  void testInterceptedMethodCallonPOJO() {
    def val = new Integer(3)
    Integer.metaClass.toString = {-> 'intercepted' }

    assertEquals "intercepted", val.toString()
  }

  void testInterceptableCalled() {
    def obj = new AnInterceptable()
    assertEquals 'intercepted', obj.existingMethod()
    assertEquals 'intercepted', obj.nonExistingMethod()
  }

  void testInterceptedExistingMethodCalled() {
    AGroovyObject.metaClass.existingMethod2 = {-> 'intercepted' }
    def obj = new AGroovyObject()
    assertEquals 'intercepted', obj.existingMethod2()
  }

  void testUnInterceptedExistingMethodCalled() {
    def obj = new AGroovyObject()
    assertEquals 'existingMethod', obj.existingMethod()
  }

  void testPropertyThatIsClosureCalled() {
    def obj = new AGroovyObject()
    assertEquals 'closure called', obj.closureProp()
  }

  void testMethodMissingCalledOnlyForNonExistent() {
    def obj = new ClassWithInvokeAndMissingMethod()
    assertEquals 'existingMethod', obj.existingMethod()
    assertEquals 'missing called', obj.nonExistingMethod()
  }

  void testInvokeMethodCalledForOnlyNonExistent() {
    def obj = new ClassWithInvokeOnly()
    assertEquals 'existingMethod', obj.existingMethod()
    assertEquals 'invoke called', obj.nonExistingMethod()
  }

  void testMethodFailsOnNonExistent() {
    def obj = new TestMethodInvocation()
    shouldFail (MissingMethodException) { obj.nonExistingMethod() }
  }
}

class AnInterceptable implements GroovyInterceptable {
  def existingMethod() {}
```

```
  def invokeMethod(String name, args) { 'intercepted' }
}

class AGroovyObject {
  def existingMethod() { 'existingMethod' }
  def existingMethod2() { 'existingMethod2' }
  def closureProp = { 'closure called' }
}

class ClassWithInvokeAndMissingMethod {
  def existingMethod() { 'existingMethod' }
  def invokeMethod(String name, args) { 'invoke called' }
  def methodMissing(String name, args) { 'missing called' }
}

class ClassWithInvokeOnly {
  def existingMethod() { 'existingMethod' }
  def invokeMethod(String name, args) { 'invoke called' }
}
```

The following output confirms that all the tests pass and Groovy handles the method as discussed:

```
.........
Time: 0.047

OK (9 tests)
```

11.2 Querying Methods and Properties

At runtime, we can query an object's methods and properties to find out if the object supports a certain behavior. This is especially useful for behavior we add dynamically at runtime. We can add behavior not only to classes, but also to select instances of a class.

We can use MetaObjectProtocol's getMetaMethod() (MetaClass extends MetaObjectProtocol) to get a metamethod. We can use getStaticMetaMethod() if we're looking for a static method. To get a list of overloaded methods, we use the plural forms of these methods—getMetaMethods() and getStaticMetaMethods(). Similarly, we can use getMetaProperty() and getStaticMetaProperty() for a metaproperty. If we want simply to check for existence and not get the metamethod or metaproperty, we use respondsTo() to check for methods and hasProperty() to check for properties.

MetaMethod "represents a Method on a Java object a little like Method except without using reflection to invoke the method," according to the Groovy documentation. If we have a method name as a string, we can call getMetaMethod() and use the resulting MetaMethod to invoke our method, like so:

ExploringMOP/UsingMetaMethod.groovy
```
str = "hello"
methodName = 'toUpperCase'
// Name may come from an input instead of being hard coded

methodOfInterest = str.metaClass.getMetaMethod(methodName)

println methodOfInterest.invoke(str)
```

The dynamically invoked method produces this output:

```
HELLO
```

We don't have to know a method name at coding time. We can get it as input and invoke the method dynamically.

To find out whether an object would respond to a method call, we can use the respondsTo() method. It takes as parameters the instance we're querying, the name of the method we're querying for, and an optional comma-separated list of arguments intended for that method. It returns a list of MetaMethods for the matching methods. Let's use that in an example:

ExploringMOP/UsingMetaMethod.groovy
```
print "Does String respond to toUpperCase()? "
println String.metaClass.respondsTo(str, 'toUpperCase')? 'yes' : 'no'

print "Does String respond to compareTo(String)? "
println String.metaClass.respondsTo(str, 'compareTo', "test")? 'yes' : 'no'

print "Does String respond to toUpperCase(int)? "
println String.metaClass.respondsTo(str, 'toUpperCase', 5)? 'yes' : 'no'
```

Here's the output from the code:

```
Does String respond to toUpperCase()? yes
Does String respond to compareTo(String)? yes
Does String respond to toUpperCase(int)? no
```

getMetaMethod() and respondsTo() offer a nice convenience. We can simply send these methods the arguments for a method we're looking for. getMetaMethod() and respondsTo() don't insist on an array of the arguments' Class like the getMethod() method in Java reflection. Even better, if the method we're interested in does not take any parameters, don't send any arguments, not even a null. This is because the last parameter to these methods is an array of parameters and Groovy treats it as optional.

There was one more magical thing taking place in the previous code: we used Groovy's special treatment of boolean (for more information, see Section 2.7, *Groovy Boolean Evaluation*, on page ?). The respondsTo() method returns a list

of MetaMethods, and since we used the result in a conditional statement (the ?: operator), Groovy returned true if there were any methods, and false otherwise. So, we don't have to explicitly check whether the size of the returned list is greater than zero—Groovy does that for us.