Extracted from:

Programming Groovy 2

Dynamic Productivity for the Java Developer

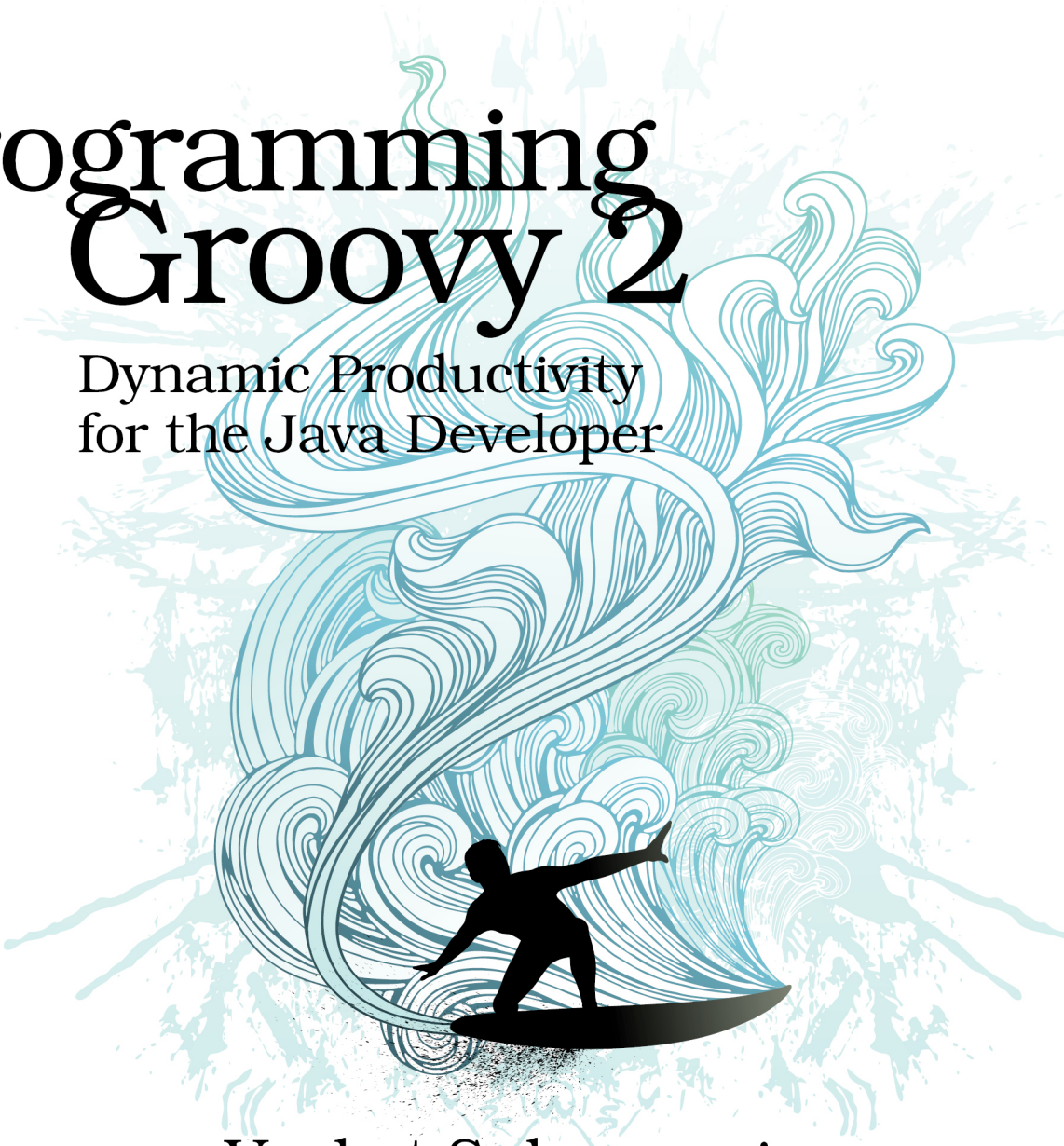
The Pragmatic Bookshelf

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

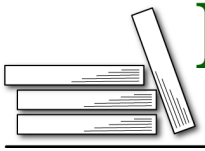
Programming Groovy 2

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.

Since Groovy supports Java syntax and preserves the Java semantics, we can intermix Java style and Groovy style at will. In this chapter we'll start on familiar ground and transition to a more Groovy style of coding. We'll begin with tasks we're used to doing in Java, and as we transition them to Groovy code we'll see how the Groovy versions are more concise and expressive. At the end of this chapter, we'll look at some "gotchas"—a few things that might catch us off guard if we aren't expecting them.

2.1 From Java to Groovy

Let's start with a piece of Java code with a simple loop. We'll first run it through Groovy. Then we'll refactor it from Java style to Groovy style. As we evolve the code, each version will do the same thing, but the code will be more expressive and concise. It will feel like our refactoring is on steroids. Let's begin.

Hello, Groovy

Let's start with a Java code example that's also Groovy code, saved in a file named Greetings.groovy.

```
// Java code
public class Greetings {
  public static void main(String[] args) {
    for(int i = 0; i < 3; i++) {
      System.out.print("ho ");
    }

    System.out.println("Merry Groovy!");
  }
}
```

Let's execute this code using the command groovy Greetings.groovy and take a look at the output:

```
ho ho ho Merry Groovy!
```

That's a lot of code for such a simple task. Still, Groovy obediently accepted and executed it.

Groovy has a higher signal-to-noise ratio than Java. Hence, less code, more result. In fact, we can get rid of most of the code from the previous program and still have it produce the same result. Let's start by removing the line-terminating semicolons. Losing the semicolons reduces noise and makes the code more fluent.

Now let's remove the class and method definitions. Groovy is still happy (or is it happier?).

Default Imports

We don't have to import all the common classes/packages when we write Groovy code. For example, Calendar readily refers to java.util.Calendar. Groovy automatically imports the following Java packages: java.lang, java.util, java.io, and java.net. It also imports the classes java.math.BigDecimal and java.math.BigInteger. In addition, the Groovy packages groovy.lang and groovy.util are imported.

GroovyForJavaEyes/LightGreetings.groovy

```
for(int i = 0; i < 3; i++) {
  System.out.print("ho ")
}

System.out.println("Merry Groovy!")
```

We can go even further. Groovy understands println() because it has been added on java.lang.Object. It also has a lighter form of the for loop that uses the Range object, and Groovy is lenient with parentheses. So, we can reduce the previous code to the following:

GroovyForJavaEyes/LighterGreetings.groovy

```
for(i in 0..2) { print 'ho ' }

println 'Merry Groovy!'
```

The output from the previous code is the same as the Java code we started with, but the code is a lot lighter. Simple things are simple to do in Groovy.

Ways to Loop

We're not restricted to the traditional for loop in Groovy. We already used the range 0..2 in the for loop. Groovy provides quite a number of elegant ways to iterate; let's look at a few.

Groovy has added a convenient upto() instance method to java.lang.Integer; let's use that to iterate.

GroovyForJavaEyes/WaysToLoop.groovy

```
0.upto(2) { print "$it "}
```

Here we called upto() on 0, which is an instance of Integer. The output should display each of the values in the range we picked.

```
0 1 2
```

So, what's that $it in the code block? In this context, it represents the index value through the loop. The upto() method accepts a closure as a parameter. If the closure expects only one parameter, we can use the default name it for it in Groovy. Keep that in mind, and move on for now; we'll discuss closures in more detail in Chapter 4, *Using Closures*, on page ?. The $ in front of the variable it tells the method print() to print the value of the variable instead of the characters "it"—using this feature we can embed expressions within strings, as you'll see in Chapter 5, *Working with Strings*, on page ?.

With the upto() method we can set both lower and upper limits. If we start at 0, we can also use the times() method, like in the next example.

GroovyForJavaEyes/WaysToLoop.groovy
```
3.times { print "$it "}
```

This version of code will produce the same output as the previous version, as we can see:

```
0 1 2
```

By using the step() method, we can skip values while looping.

GroovyForJavaEyes/WaysToLoop.groovy
```
0.step(10, 2) { print "$it "}
```

The output from the code will show select values in the range:

```
0 2 4 6 8
```

We can also iterate or traverse a collection of objects using similar methods, as you'll see later in Chapter 6, *Working with Collections*, on page ?.

To go further, we can rewrite the greetings example using the methods you learned earlier. Look at how short the following Groovy code is compared to the Java code we started with:

GroovyForJavaEyes/WaysToLoop.groovy
```
3.times { print 'ho ' }
println 'Merry Groovy!'
```

To confirm that this works, let's run the code and take a look at the output.

```
ho ho ho Merry Groovy!
```

A Quick Look at the GDK

One of the Java Platform's key strengths is its Java Development Kit (JDK). To program in Groovy, we're not forced to learn a new set of classes and libraries. Groovy extends the powerful JDK by adding convenience methods to various classes. These extensions are available in the library called the

GDK, or the Groovy JDK (http://groovy.codehaus.org/groovy-jdk). We can leverage the JDK even further in Groovy by using the Groovy convenience methods. Let's whet our appetites by making use of a GDK convenience method for talking to an external process.

I spend part of my life maintaining version-control systems. Whenever a file is checked in, back-end hooks exercise some rules, execute processes, and send out notifications. In short, I have to create and interact with processes. Let's see how Groovy can help here.

In Java, we can use java.lang.Process to interact with a system-level process. Suppose we want to invoke Subversion's help from within our code; well, here's the Java code for that:

```
//Java code
import java.io.*;
public class ExecuteProcess {
  public static void main(String[] args) {
    try {
      Process proc = Runtime.getRuntime().exec("svn help");
      BufferedReader result = new BufferedReader(
        new InputStreamReader(proc.getInputStream()));
      String line;
      while((line = result.readLine()) != null) {
        System.out.println(line);
      }
    } catch(IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

java.lang.Process is very helpful, but we had to jump through some hoops to use it in the previous code; in fact, all the exception-handling code and effort to get to the output can make us dizzy. The GDK makes this insanely simple by adding an execute() method on the java.lang.String class:

GroovyForJavaEyes/Execute.groovy

```
println "svn help".execute().text
```

Compare the two pieces of code. They remind me of the swordfight scene from the movie *Raiders of the Lost Ark*; the Java code is pulling a major stunt like the villain with the sword.[1] Groovy, on the other hand, like Indy, effortlessly gets the job done. Don't get me wrong—I am certainly not calling Java the villain. We're still using Process and the JDK in Groovy code. Our enemy is the

1. http://www.youtube.com/watch?v=anEuw8F8cpE

unnecessary complexity that makes it harder and more time-consuming to utilize the power of the JDK and the Java platform.

In one of the Subversion hooks I maintain, a refactoring session helped reduce more than fifty lines of Java code to a mere three lines of Groovy code. Which of the previous two versions would we prefer? The short and sweet one-liner, of course (unless we're consultants who get paid by the number of lines of code we write...).

When we called the execute() method on the instance of String, Groovy created an instance that extends java.lang.Process, just like the exec() method of Runtime did in the Java code. We can verify this by using the following code:

GroovyForJavaEyes/Execute.groovy
```
println "svn help".execute().getClass().name
```

When run on a Unix-like machine, the code will report as follows:

```
java.lang.UNIXProcess
```

On a Windows machine, we'll get this:

```
java.lang.ProcessImpl
```

When we call text, we're calling the Groovy-added method getText() on the Process to read the process's entire standard output into a String. If we simply want to wait for a process to finish, either waitFor() or the Groovy-added method waitForOrKill() that takes a timeout in milliseconds will help. Go ahead—try the previous code.

Instead of using Subversion, we can try other commands; simply substitute svn help for some other program (such as groovy -v):

GroovyForJavaEyes/Execute.groovy
```
println "groovy -v".execute().text
```

The separate Groovy process we invoked from within our Groovy script will report the version of Groovy.

GroovyForJavaEyes/Execute.output
```
Groovy Version: 2.1.1 JVM: 1.7.0_04-ea Vendor: Oracle Corporation OS: Mac OS X
```

This code sample works on Unix-like systems and on Windows. Similarly, on a Unix-like system, to get the current-directory listing, we can call ls:

GroovyForJavaEyes/Execute.groovy
```
println "ls -l".execute().text
```

If we're on Windows, simply replacing ls with dir will not work. The reason is that although ls is a program we're executing on Unix-like systems, dir is not

a program—it's a shell command. So, we have to do a little more than call dir. Specifically, we need to invoke cmd and ask it to execute the dir command:

GroovyForJavaEyes/Windows/ExecuteDir.groovy
```
println "cmd /C dir".execute().text
```

We've looked at how the GDK extensions can make our coding life much easier, but we've merely scratched the GDK's surface. We'll look at more GDK goodness in Chapter 7, *Exploring the GDK*, on page ?.

safe-navigation operator

Groovy has a number of little features that are exciting and help ease the development effort. You'll find them throughout this book. One such feature is the safe navigation operator (?.). It eliminates the mundane check for null, as in the next example:

GroovyForJavaEyes/Ease.groovy
```
def foo(str) {
  //if (str != null) { str.reverse() }
  str?.reverse()
}

println foo('evil')
println foo(null)
```

The ?. operator in the method foo() (programming books are required to have at least one method named "foo") calls the method or property only if the reference is not null. Let's run the code and look at the output:

```
live
null
```

The call to reverse() on the null reference using ?. resulted in a null instead of a NullPointerException—another way Groovy reduces noise and effort.

Exception Handling

Groovy has less ceremony than Java. That's crystal-clear in exception handling. Java forces us to handle checked exceptions. Consider a simple case: we want to call Thread's sleep() method. (Groovy provides an alternate sleep() method; see *Using sleep*, on page ?.) Java is adamant that we catch java.lang.InterruptedException. What does a Java developer do when forced? Finds a way around doing it. The result? Lots of empty catch blocks, right? Check this out:

GroovyForJavaEyes/Sleep.java
```
// Java code
try {
```

```
  Thread.sleep(5000);
} catch(InterruptedException ex) {
  // eh? I'm losing sleep over what to do here.
}
```

Having an empty catch block is worse than not handling an exception. If we put in an empty catch block, we're suppressing the exception. If we don't handle it in the first place, it is propagated to the caller, who either can do something about it or can pass it yet again to its caller.

Groovy does not force us to handle exceptions that we don't want to handle or that are inappropriate at the current level of code. Any exception we don't handle is automatically passed on to a higher level. Here's an example of Groovy's answer to exception handling:

GroovyForJavaEyes/ExceptionHandling.groovy

```
def openFile(fileName) {
  new FileInputStream(fileName)
}
```

The method openFile() does not handle the infamous FileNotFoundException. If the exception occurs, it's not suppressed. Instead, it's passed to the calling code, which can handle it, as in the next example:

GroovyForJavaEyes/ExceptionHandling.groovy

```
try {
  openFile("nonexistentfile")
} catch(FileNotFoundException ex) {
  // Do whatever you like about this exception here
  println "Oops: " + ex
}
```

If we are interested in catching all Exceptions that may be thrown, we can simply omit the exception type in the catch statement:

GroovyForJavaEyes/ExceptionHandling.groovy

```
try {
  openFile("nonexistentfile")
} catch(ex) {
  // Do whatever you like about this exception here
  println "Oops: " + ex
}
```

With the catch(ex) without any type in front of the variable ex, we can catch just about any exception thrown our way. Beware: this doesn't catch Errors or Throwables other than Exceptions. To catch *all* of them, use catch(Throwable throwable).

As we can see, Groovy lets us focus on getting our work done rather than on tackling annoying system-level details.

Groovy as Lightweight Java

Groovy has other features that make it lighter and easier to use. Here are some:

- The return statement is almost always optional (see Section 2.11, *Gotchas*, on page ?).
- The semicolon (;) is almost always optional, though we can use it to separate statements (see *The Semicolon Is Almost Always Optional*, on page ?).
- Methods and classes are public by default.
- The ?. operator dispatches calls only if the object reference is not null.
- We can initialize JavaBeans using named parameters (see Section 2.2, *JavaBeans*, on page ?).
- We're not forced to catch exceptions that we don't care to handle. They get passed to the caller of our code.
- We can use this within static methods to refer to the Class object. In the next example, the learn() method returns the class so we can chain calls:

```
class Wizard {
  def static learn(trick, action) {
    //...
    this
  }
}
Wizard.learn('alohomora', {/*...*/})
 .learn('expelliarmus', {/*...*/})
 .learn('lumos', {/*...*/})
```

We've seen the expressive and concise nature of Groovy. Next we'll look at how Groovy reduces clutter in one of the most fundamental features of Java.