Extracted from:

Programming Groovy

Dynamic Productivity for the Java Developer

Chapter 1

Introduction

As a busy Java developer, you're constantly looking for ways to be more productive, right? You're probably willing to take all the help you can get from the platform and tools available to you. When I wax poetic about the "strength of Java," I'm not talking about the language or its syntax. It's the Java platform that has become more capable and more performant. To reap the benefit of the platform and to tackle the inherent complexities of your applications, you need another tool—one with a dynamic and metaprogramming capabilities. Java—the language—has been flirting with that idea for a while and will support these features to various degrees in future versions. However, you don't have to wait for that day. You can build performant Java applications with all the dynamic capabilities today, right now, using Groovy.

1.1 Why Dynamic Languages?

Dynamic languages have the ability to extend a program at runtime, including changing the structure of objects, types, and behavior. Dynamic languages allow you to do things at runtime that static languages do at compile time; they allow you to execute program statements that were created on the fly at runtime.

For example, if you want to get the date five days from now, you can write this:

```
5.days.from.now
```

Yes, that's your friendly java.lang.Integer chirping dynamic behavior in Groovy, as you'll learn later in this book.

The flexibility offered by dynamic languages gives you the advantage of evolving your application as it executes. You are probably familiar with code generation and code generation tools. I consider code generation to be soooo 20th century. In fact, generated code is like an incessant itch on your back; if you keep scratching it, it turns into a sore. With dynamic languages, there are better ways. I prefer *code synthesis*, which is in-memory code creation at runtime. Dynamic languages make it easy to "synthesize code." The code is synthesized based on the flow of logic through your application and becomes active "just in time."

By carefully applying these capabilities of dynamic languages, you can be more productive as an application developer. This higher productivity means you can easily create higher levels of abstractions in shorter amounts of time. You can also use a smaller, yet more capable, set of developers to create applications. In addition, greater productivity means you can create parts of your application quickly and get feedback from your fellow developers, testers, domain experts, and customer representatives. And all this leads to greater agility.[1]

Dynamic languages have been around for a long time, so you may be asking, why is now a great time to get excited about them? I can answer that with four reasons:[2]

- Machine speed
- Availability
- Awareness of unit testing
- Killer applications

Let's discuss each of these reasons for getting excited about dynamic languages, starting with machine speed. Doing at runtime what other languages do at compile time first raises the concern of the speed of dynamic languages. Furthermore, interpreting code at runtime rather than simply executing compiled code adds to that concern. Fortunately, machine speed has consistently increased over the years—handhelds have more computing and memory power today than what large computers had decades ago. Tasks that were quite unimaginable using a

1. Tim O'Reilly observes the following about developing web applications: "Rather than being finished paintings, they are sketches, continually being redrawn in response to new data." He also makes the point that dynamic languages are better suited for these in "Why Scripting Languages Matter" (see Appendix A, on page 293).
2. A fifth reason is the ability to run dynamic languages on the JVM, but that came much later.

1980s processor are easy to achieve today. The performance concerns of dynamic languages are greatly eased because of processor speeds and other improvements in our field, including better just-in-time compilation techniques.

Now let's talk about availability. The Internet and active "public" community-based development have made recent dynamic languages easily accessible and available. Developers can now easily download languages and tools and play with them. They can even participate in community forums to influence the evolution of these languages.[3] This is leading to greater experimentation, learning, and adaptation of languages than in the past.

Now it's time to talk about the awareness of unit testing. Most dynamic languages are dynamically typed. The types are often inferred based on the context. There are no compilers to flag type-casting violations at compile time. Since quite a bit of code may be synthesized and your program can be extended at runtime, you can't simply rely upon coding-time verification alone. Writing code in dynamic languages requires a greater discipline from the testing point of view. Over the past few years, we've seen greater awareness among programmers (though not sufficiently greater adoption yet) in the area of testing in general and unit testing in particular. Most of the programmers who have taken advantage of these dynamic languages for commercial application development have also embraced testing and unit testing.[4]

Finally, let's discuss the fourth bullet point listed earlier. Many developers have in fact been using dynamic languages for decades. However, for the majority of the industry to be excited about them, we had to have killer applications—those compelling stories to share with your developers and managers. That tipping point, for Ruby in particular and for dynamic languages in general, came in the form of Rails ([TH05], [SH07], [Tat06]). Rails showed struggling web developers how they could quickly develop applications using the dynamic capabilities of Ruby. Along the same vein came Grails built using Groovy and Java, Django built using Python, and Lift built using Scala, to mention a few.

3. The Groovy users mailing list is very active, with constant discussions from passionate users expressing opinions, ideas, and criticisms on current and future features. Visit http://groovy.codehaus.org/Mailing+Lists and http://groovy.markmail.org if you don't believe me.
4. "Legacy code is simply code without tests." —Michael C. Feathers [Fea04]

These frameworks have caused enough stir in the development community to make the industry-wide adoption of dynamic languages a highly probable event in the near future.

I find that dynamic languages, along with metaprogramming capabilities, make simple things simpler and harder things manageable. You still have to deal with the inherent complexity of your application, but dynamic languages let you focus your effort where it's deserved. When I got into Java (after years of C++), features such as reflection, a good set of libraries, and evolving framework support made me productive. The JVM, to a certain extent, provided me with the ability to take advantage of metaprogramming. However, I had to use something in addition to Java to tap into that potential—heavyweight tools such as AspectJ. Like several other productive programmers, I found myself left with two options. The first option was to use the exceedingly complex and not-so-flexible Java along with heavyweight tools. The second option was to move on to using dynamic languages such as Ruby that are object-oriented and have metaprogramming capability built in (for instance, it takes only a couple of lines of code to do AOP in Ruby and Groovy). A few years ago, taking advantage of dynamic capabilities and metaprogramming and being productive at the same time meant leaving behind the Java platform. (After all, you use these features to be productive and can't let them slow you down, right?) That is not the case anymore. Languages such as Groovy and JRuby are dynamic and run on the JVM. They allow you to take full advantage of both the rich Java platform and dynamic language capabilities.

1.2 What's Groovy?

Groovy[5] is a lightweight, low-ceremony, dynamic, object-oriented language that runs on the JVM. Groovy is open sourced under Apache License, version 2.0. It derives strength from different languages such as Smalltalk, Python, and Ruby while retaining a syntax familiar to Java programmers. Groovy compiles into Java bytecode and extends the Java API and libraries. It runs on Java 1.4 or newer. For deployment, all you need is a Groovy JAR in addition to your regular Java stuff, and you're all set.

5. Merriam-Webster defines Groovy as "marvelous, wonderful, excellent, hip, trendy."

I like to define Groovy as "a language that has been reborn several times."[6] James Strachan and Bob McWhirter started it in 2003, and it was commissioned into Java Specification Request (JSR 241) in March 2004. Soon after, it was almost abandoned because of various difficulties and issues. Guillaume Laforge and Jeremy Rayner decided to rekindle the efforts and bring Groovy back to life. Their first effort was to fix bugs and stabilize the language features. The uncertainty lingered on for a while. I know a number of people (committers and users) who simply gave up on the language at one time. Finally, a group of smart and enthusiastic developers joined force with Guillaume and Jeremy, and a vibrant developer community emerged. JSR version 1 was announced in August 2005.

Groovy version 1.0 release was announced on January 2, 2007. It was encouraging to see that, well before it reached 1.0, Groovy was put to use on commercial projects in a handful of organizations in the United States and Europe. In fact, I've seen growing interest in Groovy in conferences and user groups around the world. Several organizations and developers are beginning to use Groovy at various levels on their projects, and I think the time is ripe for major Groovy adoption in the industry. Groovy version 1.5 was released on December 7, 2007.

Grails ([Roc06], [Rud07]),[7] built using Groovy and Java, is a dynamic web development framework based on "coding by convention." It allows you to quickly build web applications on the JVM using Groovy, Spring, Hibernate, and other Java frameworks.

1.3 Why Groovy?

As a Java programmer, you don't have to switch completely to a different language. Trust me, Groovy feels like the Java language you already know with but with a few augmentations.

There are dozens of scripting languages[8] that can run on the JVM, such as Groovy, JRuby, BeanShell, Scheme, Jaskell, Jython, JavaScript, etc. The list could go on and on. Your language choice should depend on a number of criteria: your needs, your preferences, your background, the projects you work with, your corporate technical environment, and so

6. See "A bit of Groovy history," a blog by Guillaume Laforge at http://glaforge.free.fr/weblog/index.php?itemid=99.
7. See Jason Rudolph's "Getting Started with Grails" in Appendix A, on page 293.
8. https://scripting.dev.java.net

on. In this section, I will discuss whether Groovy is the *right* language for you.

As a programmer, I am shameless about languages. I can comfortably program in about eight structured, object-oriented, and functional programming languages and can come dangerously close to writing code in a couple more. In any given year, I actively code in about two to three languages at least. So, if one thing, I am pretty unbiased when it comes to choosing a language—I will pick the one that works the best for a given situation. I am ready to change to another language with the ease of changing a shirt, if that is the right thing to do, that is.

Groovy is an attractive language for a number of reasons:

- It has a flat learning curve.
- It follows Java semantics.
- It bestows dynamic love.
- It extends the JDK.

I'll now expand on these reasons. First, you can take almost any Java code[9] and run it as Groovy. The significant advantage of this is a flat learning curve. You can start writing code in Groovy and, if you're stuck, simply switch gears and write the Java code you're familiar with. You can later refactor that code and make it groovier.

For example, Groovy understands the traditional for loop. So, you can write this:

```
// Java Style
for(int i = 0; i < 10; i++)
{
        //...
}
```

As you learn Groovy, you can change that to the following code or one of the other flavors for looping in Groovy (don't worry about the syntax right now; after all, you're just getting started, and very soon you'll be a pro at it):

```
10.times {
        //...
}
```

9. See Section 3.8, *Gotchas*, on page 69 for known problem areas.

Second, when programming in Groovy, you can expect almost everything you expect in Java. Groovy classes extend the same good old java.lang.Object—Groovy classes are Java classes. The OO paradigm and Java semantics are preserved, so when you write expressions and statements in Groovy, you already know what those mean to you as a Java programmer.

Here's a little example to show you that Groovy classes *are* Java classes:

Download Introduction/UseGroovyClass.groovy

```
println XmlParser.class
println XmlParser.class.superclass
```

If you run groovy UseGroovyClass, you'll get the following output:

```
class groovy.util.XmlParser
class java.lang.Object
```

Now let's talk about the third reason to love Groovy. Groovy is dynamic, and it is optionally typed. If you've enjoyed the benefits of other dynamic languages such as Smalltalk, Python, JavaScript, and Ruby, you can realize those in Groovy. If you had looked at Groovy 1.0 support for metaprogramming, it probably left you desiring for more. Groovy has come a long way since 1.0, and Groovy 1.5 has pretty decent metaprogramming capabilities.

For instance, if you want to add the method isPalindrome() to String—a method that tells whether a word spells the same forward and backward—you can add that easily with only a couple lines of code (again, don't try to figure out all the details of how this works right now; you have the rest of the book for that):

Download Introduction/Palindrome.groovy

```
String.metaClass.isPalindrome = {->
  delegate == delegate.reverse()
}

word = 'tattarrattat'
println "$word is a palindrome? ${word.isPalindrome()}"
word = 'Groovy'
println "$word is a palindrome? ${word.isPalindrome()}"
```

The following output shows how the previous code works:

```
tattarrattat is a palindrome? true
Groovy is a palindrome? false
```

Finally, as a Java programmer, you rely heavily on the JDK and the API to get your work done. These are still available in Groovy. In addition,

Groovy extends the JDK with convenience methods and closure support through the GDK. Here's a quick example of an extension in GDK to the java.util.ArrayList class:

```
lst = ['Groovy', 'is', 'hip']
println lst.join(' ')
println lst.getClass()
```

The output from the previous code confirms that you're still working with the JDK but that you used the Groovy-added join() method to concatenate the elements in the ArrayList:

```
Groovy is hip
class java.util.ArrayList
```

You can see how Groovy takes the Java you know and augments it. If your project team is familiar with Java, if they're using it for most of your organization's projects, and if you have a lot of Java code to integrate and work with, then you will find that Groovy is a nice path toward productivity gains.

1.4 What's in This Book?

This book is about programming using the Groovy language. I make no assumptions about your knowledge of Groovy or dynamic languages, although I do assume you are familiar with Java and the JDK. Throughout this book, I will walk you through the concepts of the Groovy language, presenting you with enough details and a number of examples to illustrate the concepts. My objective is for you to get proficient with Groovy by the time you put this book down, after reading a substantial portion of it, of course.

The rest of this book is organized as follows:

The book has has three parts: "Beginning Groovy," "Using Groovy," and "MOPping Groovy."

In the chapters in Part 1, "Beginning Groovy," I focus on the whys and whats of Groovy—those fundamentals that'll help you get comfortable with general programming in Groovy. Since I assume you're familiar with Java, I don't spend any time with programming basics, like what an if statement is or how to write it. Instead, I take you directly to the similarities of Groovy and Java and topics that are specific to Groovy.

In the chapters in Part 2, "Using Groovy," I focus on how to use Groovy for everyday coding—working with XML, accessing databases, and

working with multiple Java/Groovy classes and scripts—so you can put Groovy to use right away for your day-to-day tasks. I also discuss the Groovy extensions and additions to the JDK so you can take advantage of both the power of Groovy and the JDK at the same time.

In the chapters in Part 3, "MOPping Groovy," I focus on the metaprogramming capabilities of Groovy. You'll see Groovy really shine in these chapters and learn how to take advantage of its dynamic nature. You'll start with the fundamentals of MetaObject Protocol (MOP), learn how to do aspect-oriented programming (AOP) such as operations in Groovy, and learn about dynamic method/property discovery and dispatching. Then you'll apply those right away to creating and using builders and domain-specific languages (DSLs). Unit testing is not only necessary in Groovy because of its dynamic nature, but it is also easier to do—you can use Groovy to unit test your Java and Groovy code, as you'll see in this part of the book.

Here's what's in each chapter:

Part 1: "Beginning Groovy"

In Chapter 2, *Getting Started*, on page 32, you'll download and install Groovy and take it for a test-drive right away using groovysh and groovyConsole. You'll also learn how to run Groovy without these tools—from the command line and within your IDEs.

In Chapter 3, *Groovy for the Java Eyes*, on page 39, you'll start with familiar Java code and refactor that to Groovy. After a quick tour of Groovy features that improve your everyday Java coding, you'll learn about Groovy's support for Java 5 features. Groovy follows Java semantics, except in places it does not—you'll also learn gotchas that'll help avoid surprises.

In Chapter 4, *Dynamic Typing*, on page 77, you'll see how Groovy's typing is similar and different from Java's typing, what Groovy really does with the type information you provide, and when to take advantage of dynamic typing vs. optional typing. You'll also learn how to take advantage of Groovy's dynamic typing, design by capability, and multimethods.

In Chapter 5, *Using Closures*, on page 94, you'll learn all about the exciting Groovy feature called *closures*, including what they are, how they work, and when and how to use them.

In Chapter 6, *Working with Strings*, on page 113, you'll learn about Groovy strings, working with multiline strings, and Groovy's support for regular expressions.

In Chapter 7, *Working with Collections*, on page 126, you'll explore Groovy's support for Java collections—lists and maps. You'll learn various convenience methods on collections, and after this chapter, you'll never again want to use your collections the old way.

Part 2: "Using Groovy"

Groovy embraces and extends the JDK. You'll explore the GDK and learn the extensions to Object and other Java classes in Chapter 8, *Exploring the GDK*, on page 143.

Groovy has pretty good support for working with XML, including parsing and creating XML documents, as you'll see in Chapter 9, *Working with XML*, on page 157.

Chapter 10, *Working with Databases*, on page 166 presents Groovy's SQL support, which will make your database-related programming easy and fun. In this chapter, you'll learn about iterators, datasets, and how to perform regular database operations using simpler syntax and closures. I'll also show how to get data from Microsoft Excel documents.

One of the key strengths of Groovy is the integration with Java. In Chapter 11, *Working with Scripts and Classes*, on page 174, you'll learn ways to closely interact with multiple Groovy scripts, Groovy classes, and Java classes from within your Groovy and Java code.

Part 3: "MOPping Groovy"

Metaprogramming is one of the biggest benefits of dynamic languages and Groovy; it has the ability to inspect classes at runtime and dynamically dispatch method calls. You'll explore Groovy's support for metaprogramming in Chapter 12, *Exploring Meta-Object Protocol (MOP)*, on page 186, beginning with the fundamentals of how Groovy handles method calls to Groovy objects and Java objects.

Groovy allows you to perform AOP-like method interceptions using GroovyInterceptable and ExpandoMetaClass, as you'll see in Chapter 13, *Intercepting Methods Using MOP*, on page 196.

In Chapter 14, *MOP Method Injection and Synthesis*, on page 204, you'll dive into Groovy metaprogramming capabilities that allow you to inject and synthesize methods at runtime.

In Chapter 15, *MOPping Up*, on page 226, you will learn how to synthesize classes dynamically, how to use metaprogramming to delegate method calls, and how to choose between different metaprogramming techniques you've learned in the previous three chapters.

Unit testing is not a luxury or a "if-we-have-time" practice in Groovy. The dynamic nature of Groovy requires unit testing, and fortunately, at the same time, it facilitates writing tests and creating mock objects, as you'll learn in Chapter 16, *Unit Testing and Mocking*, on page 236. You will learn techniques that will help you use Groovy to unit test your Java code and Groovy code.

Groovy builders are specialized classes that help you build internal DSLs for a nested hierarchy. You can learn how to use them and to create your own builders in Chapter 17, *Groovy Builders*, on page 262.

You can apply Groovy's metaprogramming capabilities to build internal DSLs using the techniques you'll learn in Chapter 18, *Creating DSLs in Groovy*, on page 279. You'll start by learning about DSLs, including their characteristics, and quickly jump in to build them in Groovy.

Finally, Appendix A, on page 293 and Appendix B, on page 298, gather together all the references to web articles and books cited throughout this book.

1.5 Who Is This Book For?

This book is for developers working on the Java platform. It is better suited for programmers (and testers) who understand the Java language fairly well. Other developers who understand programming in other languages can use this book as well, but they should supplement it with books that provide them with an in-depth understanding of Java and the JDK.

Programmers who are somewhat familiar with Groovy can use this book to learn some tips and tricks of the language that they may not otherwise have the opportunity to explore. Finally, those already familiar with Groovy may find this book useful for training or coaching fellow developers in their organizations.

1.6 Acknowledgments

Writing a book is like writing a screenplay—a lot of things are added, changed, and deleted from the original manuscript. What you're hold-

ing in your hand is a work I started, but a number of people helped get it into its current form. If you find this book useful and interesting, it was a result of a collective effort. Any mistakes you find are my own—I take responsibility for those.

First, I thank Daniel Steinberg for editing this book. His command of the subject, attention to the detail, patience, and real-time response[10] were instrumental to the quality and record-time completion of this book. I call his edits "immense quality at Internet speed."

I thank Dave Thomas, Andy Hunt, Steve Peter, Kim Wimpsett, and the rest of the Pragmatic team who worked behind the scenes to get this book published. The Pragmatic Bookshelf's writing process is agile, and I can't imagine writing a book any other way without the simple yet effective tools, facilities, and practices they've created.

I had the privilege of a number of Groovy and Grails committers reviewing this book. I thank Alexandru Popescu, Dierk Konig, Graeme Rocher, Guillaume Laforge, Jason Rudolph, Jeff Brown, John Wilson, and Russel Winder for their valuable input, corrections, and clarifications. I also thank the other Groovy committers and community for their help through the Groovy users mailing list—for answering my questions, explaining things I didn't understand, and quickly fixing the bugs I found.

I thank Brian Sletten, David Geary, Joe McTee, Nathaniel Schutta, Scott Davis, Scott Leberknight, and Stuart Halloway for taking time away from their extremely busy schedules to review this book and offer their valuable input.

I also thank the developers who purchased this book in the beta form. You started giving feedback within 24 hours of the release of the beta book! Thank you Adam Rinehart, Alan Thompson, Frederic Jean, John Loizeaux, Kevin Hutchinson, Richard Boreiko, Tim Hennekey, and Todd W. Crone for your feedback, suggestions, and corrections.

I thank those wonderful developers who have endured my training, conference presentations, and podcasts. The questions you asked, your genuine interest, and your constructive feedback were very helpful—you gave me confidence and encouraged me to continue writing.

10. I was surprised when I checked in a chapter around 6 a.m. on a Sunday and got high-quality feedback from Dan within a couple of hours.

I thank Jay Zimmerman for giving me the opportunity to present a number of these concepts at the No Fluff Just Stuff conferences (http://www.nofluffjuststuff.com) around the world and for creating a community of exceptional speakers and developers.

Special thanks to the NFJS opinionated geeks—excuse me, I mean my friends and fellow speakers—who I meet several weekends each year for their friendship, passion, opinions, and discussions on various topics. Where else do you find guys who argue checked vs. unchecked exceptions for three hours in a London restaurant and then some back at the hotel?

Writing this book would not even have been imaginable without my wife's encouragement, support, and sacrifice. She has been too generous to me over the past several years, especially when I disappeared while writing this book. Thank you, Kavitha, for giving me the wings. My sincere thanks to my sons, Karthik and Krupakar, for being so kind and understanding—you guys are my inspiration.

Chapter 3

Groovy for the Java Eyes

I'll help you ease into Groovy in this chapter. Specifically, we'll start on familiar ground and then transition into the Groovy way of writing. Since Groovy preserves Java syntax and semantics, you can mix Java style and Groovy style at will. And, as you get comfortable with Groovy, you can make your code even groovier. So, get ready for a tour of Groovy. We'll wrap this chapter with some "gotchas"—a few things that might catch you off guard if you aren't expecting them.

3.1 From Java to Groovy

Groovy readily accepts your Java code. So, start with the code you're familiar with, but run it through Groovy. As you work, figure out elegant and Groovy ways to write your code. You'll see that your code is doing the same things, but it's a lot smaller. It'll feel like your refactoring is on steroids.

Hello, Groovy

Here a Java sample that's also Groovy code:

```
// 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!");
  }
}
```

Default Imports

You don't have to import some common classes/packages when you 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.

The output from the previous code is as follows:

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

That's a lot of code for such a simple task. Still, Groovy will obediently accept and execute it. Simply save that code to a file named Greetings.groovy, and execute it using the command groovy Greetings.

Groovy has a higher signal-to-noise ratio. Hence, less code, more result. In fact, you can get rid of most of the code from the previous program and still have it produce the same result. Start by removing the line-terminating semicolons first. Losing the semicolons not only reduces noise, but it also helps to use Groovy to implement internal DSLs (Chapter 18, *Creating DSLs in Groovy*, on page 279).

Then remove the class and method definition. Groovy is still happy (or is it happier?).

Download GroovyForJavaEyes/LightGreetings.groovy

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

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

You 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, you can reduce the previous code to the following:

Download 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 you started with, but the code is a lot lighter. That just goes to show you that simple things are simple to do in Groovy.

Ways to Loop

You're not restricted to the traditional for loop in Groovy. You already used the range 0..2 in the for loop. Wait, there's more.[1]

Groovy has added a convenient upto() instance method to java.lang.Integer, so you can loop using that method, as shown here:

Download GroovyForJavaEyes/WaysToLoop.groovy

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

Here you called upto() on 0, which is an instance of Integer. The output from the previous code is as follows:

```
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, you 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 5, *Using Closures*, on page 94. The $ in front of the variable it tells the method println() to print the value of the variable instead of the characters "it"—it allows you to embed expressions within strings, as you'll see in Chapter 6, *Working with Strings*, on page 113.

The upto() method allows you to set both lower and upper limits. If you start at 0, you can also use the times() method, as shown here:

Download GroovyForJavaEyes/WaysToLoop.groovy

```
3.times { print "$it "}
```

The output from previous code is as follows:

```
0 1 2
```

If you want to skip values while looping, use the step() method:

Download GroovyForJavaEyes/WaysToLoop.groovy

```
0.step(10, 2) { print "$it "}
```

The output from the previous code is as follows:

```
0 2 4 6 8
```

1. http://groovy.codehaus.org/Looping

You've now seen simple looping in action. You can also iterate or traverse a collection of objects using similar methods, as you'll see later in Chapter 7, *Working with Collections*, on page 126.

To go further, you 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 you started with:

Download GroovyForJavaEyes/WaysToLoop.groovy

```
3.times { print 'ho ' }
println 'Merry Groovy!'
```

To confirm, the output from the previous code is as follows:

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

A Quick Look at the GDK

Groovy extends the JDK with an extension called the GDK[2] or the Groovy JDK. I'll whet your appetite here with a quick example.

In Java, you can use java.lang.Process to interact with a system-level process. Suppose you want to invoke Subversion's help from within your 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();
    }
  }
}
```

2. http://groovy.codehaus.org/groovy-jdk.html

java.lang.Process is very helpful, but I 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 makes me dizzy. But the GDK, on the other hand, makes this insanely simple:

Download GroovyForJavaEyes/Execute.groovy

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

Compare the two pieces of code. They remind me of the sword-fight scene[3] from the movie *Raiders of the Lost Ark*; the Java code is pulling a major stunt like the villain with the sword. 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. You're still using Process and the JDK in Groovy code. Your enemy is the unnecessary complexity that makes it harder and time-consuming to utilize the power of the JDK and the Java platform.

Which of the previous two versions would you prefer? The short and sweet one-liner, of course (unless you're a consultant who gets paid by the number of lines of code you write...).

When you 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. You can verify this by using the following code:

Download GroovyForJavaEyes/Execute.groovy

```
println "svn help".execute().getClass().name
```

The output from the previous code, when run on a Unix-like machine, is as follows:

```
java.lang.UNIXProcess
```

On a Windows machine, you'll get this:

```
java.lang.ProcessImpl
```

When you call text, you're calling the Groovy-added method getText() on the Process to read the process's entire standard output into a String.[4] Go ahead, try the previous code.

3. http://www.youtube.com/watch?v=m5TcfywPj0E

4. If you simply want to wait for a process to finish, use either waitFor() or the Groovy-added method waitForOrKill() that takes a timeout in milliseconds.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Programming Groovy's Home Page
http://pragprog.com/titles/vslg
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/vslg.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com