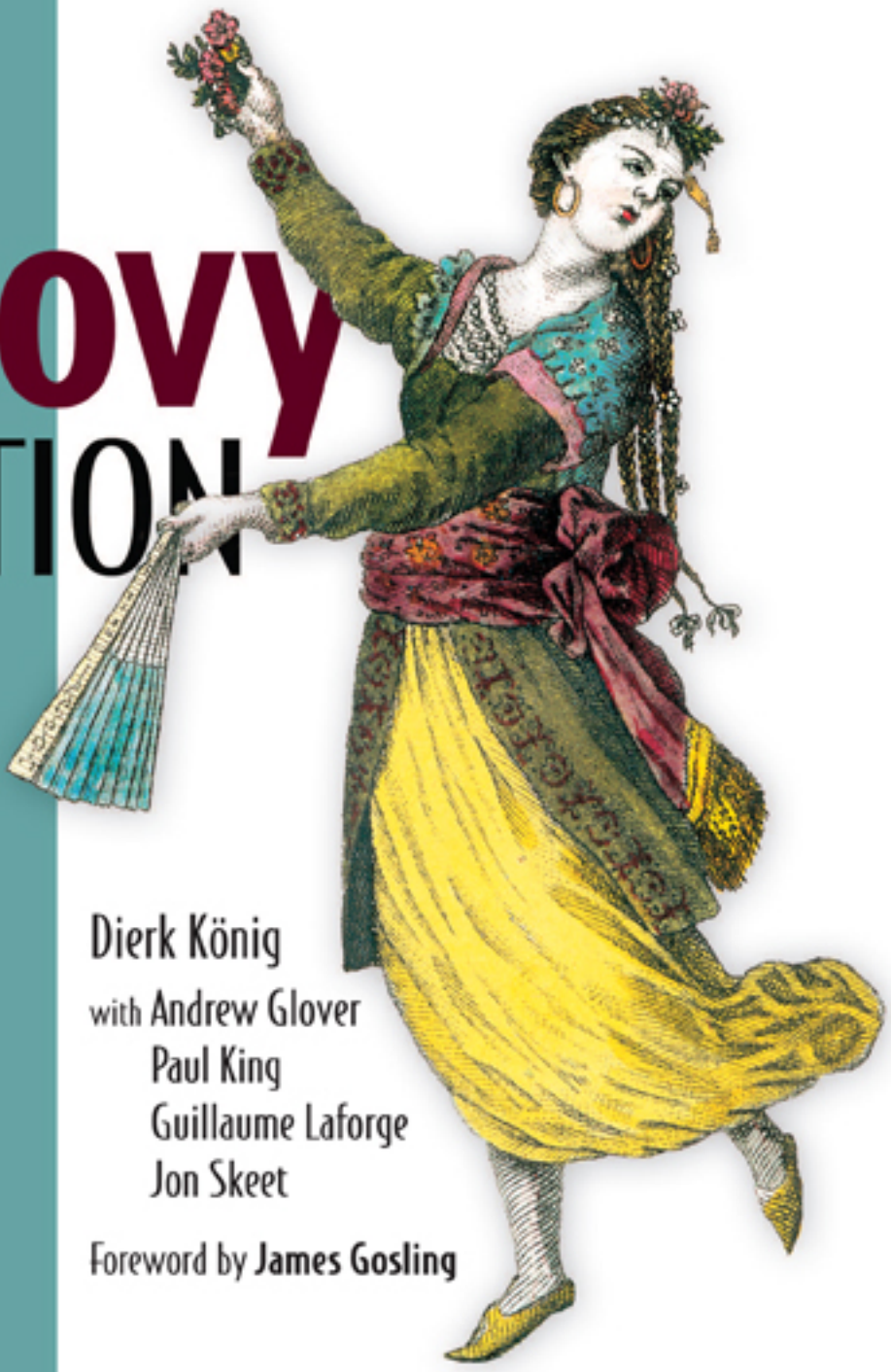
SAMPLE CHAPTER

Groovy IN ACTION



Dierk König
with Andrew Glover
Paul King
Guillaume Laforge
Jon Skeet

Foreword by **James Gosling**

MANNING

Groovy in Action

by Dierk König
with Andrew Glover, Paul King
Guillaume Laforge, and Jon Skeet

Sample Chapter 1

brief contents

1 Your way to Groovy

One main factor in the upward trend of animal life has been the power of wandering.

—Alfred North Whitehead

Welcome to the world of Groovy.

You've heard of Groovy on blogs and mailing lists. Maybe you've seen a snippet here and there. Perhaps a colleague has pointed out a page of your code and claimed the same work could be done in just a few lines of Groovy. Maybe you only picked up this book because the name is catchy. Why should you learn Groovy? What payback can you expect?

Groovy will give you some quick wins, whether it's by making your Java code simpler to write, by automating recurring tasks, or by supporting ad-hoc scripting for your daily work as a programmer. It will give you longer-term wins by making your code simpler to *read*. Perhaps most important, it's fun to use.

Learning Groovy is a wise investment. Groovy brings the power of advanced language features such as closures, dynamic typing, and the meta object protocol to the Java platform. Your Java knowledge will not become obsolete by walking the Groovy path. Groovy will build on your existing experience and familiarity with the Java platform, allowing you to pick and choose when you use which tool—and when to combine the two seamlessly.

If you have ever marveled at the Ruby folks who can implement a full-blown web application in the afternoon, the Python guys juggling collections, the Perl hackers managing a server farm with a few keystrokes, or Lisp gurus turning their whole codebase upside-down with a tiny change, then think about the *language* features they have at their disposal. The goal of Groovy is to provide language capabilities of comparable impact on the Java platform, while obeying the Java object model and keeping the perspective of a Java programmer.

This first chapter provides background information about Groovy and everything you need to know to get started. It starts with the Groovy story: why Groovy was created, what considerations drive its design, and how it positions itself in the landscape of languages and technologies. The next section expands on Groovy's merits and how they can make life easier for you, whether you're a Java programmer, a script aficionado, or an agile developer.

We strongly believe that there is only one way to learn a programming language: by trying it. We present a variety of scripts to demonstrate the compiler, interpreter, and shells, before listing some plug-ins available for widely used IDEs and where to find the latest information about Groovy.

By the end of this chapter, you will have a basic understanding of what Groovy is and how you can experiment with it.

We—the authors, the reviewers, and the editing team—wish you a great time programming Groovy and using this book for guidance and reference.

1.1 The Groovy story

At GroovyOne 2004—a gathering of Groovy developers in London—James Strachan gave a keynote address telling the story of how he arrived at the idea of inventing Groovy.

Some time ago, he and his wife were waiting for a late plane. While she went shopping, he visited an Internet café and spontaneously decided to go to the Python web site and study the language. In the course of this activity, he became more and more intrigued. Being a seasoned Java programmer, he recognized that his home language lacked many of the interesting and useful features Python had invented, such as native language support for common datatypes in an expressive syntax and, more important, dynamic behavior. The idea was born to bring such features to Java.

This led to the main principles that guide Groovy's development: to be a feature rich and Java friendly language, bringing the attractive benefits of dynamic languages to a robust and well-supported platform.

Figure 1.1 shows how this unique combination defines Groovy's position in the varied world of languages for the Java platform.[1] We don't want to offend anyone by specifying exactly where we believe any particular other language might fit in the figure, but we're confident of Groovy's position.

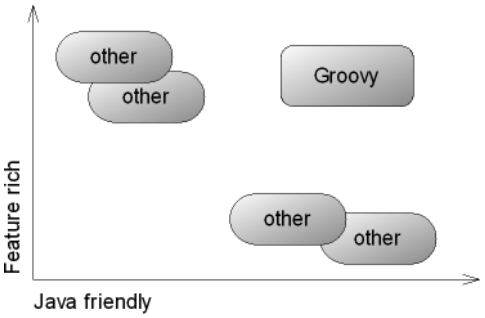


Figure 1.1 The landscape of JVM-based languages. Groovy is feature rich *and* Java friendly—it excels at both sides instead of sacrificing one for the sake of the other.

Some languages may have a few more features than Groovy. Some languages may claim to integrate better with Java. None can currently touch Groovy when you consider both aspects together: Nothing provides a better combination of Java friendliness *and* a complete range of modern language features.

Knowing some of the aims of Groovy, let's look at what it *is*.

[1] http://www.robert-tolksdorf.de/vmlanguages.html lists close to 200 (!) languages targeting the Java Virtual Machine.

1.1.1 What is Groovy?

The Groovy web site (http://groovy.codehaus.org) gives one of the best definitions of Groovy: "Groovy is an agile dynamic language for the Java Platform with many features that are inspired by languages like Python, Ruby and Smalltalk, making them available to Java developers using a Java-like syntax."

Groovy is often referred to as a scripting language—and it works very well for scripting. It's a mistake to label Groovy purely in those terms, though. It can be precompiled into Java bytecode, be integrated into Java applications, power web applications, add an extra degree of control within build files, and be the basis of whole applications on its own—Groovy is too flexible to be pigeon-holed.

What we *can* say about Groovy is that it is closely tied to the Java platform. This is true in terms of both implementation (many parts of Groovy are written in Java, with the rest being written in Groovy itself) and interaction. When you program in Groovy, in many ways you're writing a special kind of Java. All the power of the Java platform—including the massive set of available libraries—is there to be harnessed.

Does this make Groovy just a layer of syntactic sugar? Not at all. Although everything you do in Groovy *could* be done in Java, it would be madness to write the Java code required to work Groovy's magic. Groovy performs a lot of work behind the scenes to achieve its agility and dynamic nature. As you read this book, try to think every so often about what would be required to mimic the effects of Groovy using Java. Many of the Groovy features that seem extraordinary at first—encapsulating logic in objects in a natural way, building hierarchies with barely any code other than what is *absolutely* required to compute the data, expressing database queries in the normal application language before they are translated into SQL, manipulating the runtime behavior of individual objects after they have been created—all of these are tasks that Java cannot perform. You might like to think of Groovy as being a "full color" language compared with the monochrome nature of Java—the miracle being that the color pictures are created out of lots of carefully engineered black and white dots.

Let's take a closer look at what makes Groovy so appealing, starting with how Groovy and Java work hand-in-hand.

1.1.2 Playing nicely with Java: seamless integration

Being Java friendly means two things: seamless integration with the Java Runtime Environment and having a syntax that is aligned with Java.

Seamless integration

Figure 1.2 shows the integration aspect of Groovy: It runs inside the Java Virtual Machine and makes use of Java's libraries (together called the Java Runtime Environment or *JRE*). Groovy is only a new way of creating *ordinary* Java classes—from a runtime perspective, Groovy *is* Java with an additional jar file as a dependency.

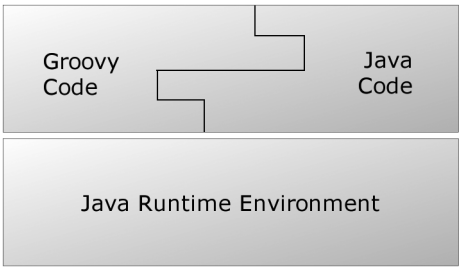


Figure 1.2 Groovy and Java join together in a tongue-and-groove fashion.

Consequently, calling Java from Groovy is a nonissue. When developing in Groovy, you end up doing this all the time without noticing. Every Groovy type is a subtype of `java.lang.Object`. Every Groovy object is an instance of a type in the normal way. A Groovy date *is* a `java.util.Date`, and so on.

Integration in the opposite direction is just as easy. Suppose a Groovy class `MyGroovyClass` is compiled into a *.class file and put on the classpath. You can use this Groovy class from within a Java class by typing

```
new MyGroovyClass();    // create from Java
```

In other words, instantiating a Groovy class is identical to instantiating a Java class. After all, a Groovy class *is* a Java class. You can then call methods on the instance, pass the reference as an argument to methods, and so forth. The JVM is blissfully unaware that the code was written in Groovy.

Syntax alignment

The second dimension of Groovy's friendliness is its syntax alignment. Let's compare the different mechanisms to obtain today's date in Java, Groovy, and Ruby in order to demonstrate what alignment *should* mean:

```
import java.util.*;           // Java
Date today = new Date();      // Java

today = new Date()            // a Groovy Script

require 'date'                 # Ruby
today = Date.new               # Ruby
```

The Groovy solution is short, precise, and more compact than normal Java. Groovy does not need to import the `java.util` package or specify the `Date` type; moreover, Groovy doesn't require semicolons when it can understand the code

without them. Despite being more compact, Groovy is fully comprehensible to a Java programmer.

The Ruby solution is listed to illustrate what Groovy avoids: a different packaging concept (`require`), a different comment syntax, and a different object-creation syntax. Although the Ruby way makes sense in itself (and may even be more consistent than Java), it does not align as nicely with the Java syntax and architecture as Groovy does.

Now you have an idea what Java friendliness means in terms of integration and syntax alignment. But how about feature richness?

1.1.3 Power in your code: a feature-rich language

Giving a list of Groovy features is a bit like giving a list of moves a dancer can perform. Although each feature is important in itself, it's how well they work together that makes Groovy shine. Groovy has three main types of features over and above those of Java: language features, libraries specific to Groovy, and additions to the existing Java standard classes (GDK). Figure 1.3 shows some of these features and how they fit together. The shaded circles indicate the way that the features use each other. For instance, many of the library features rely heavily on

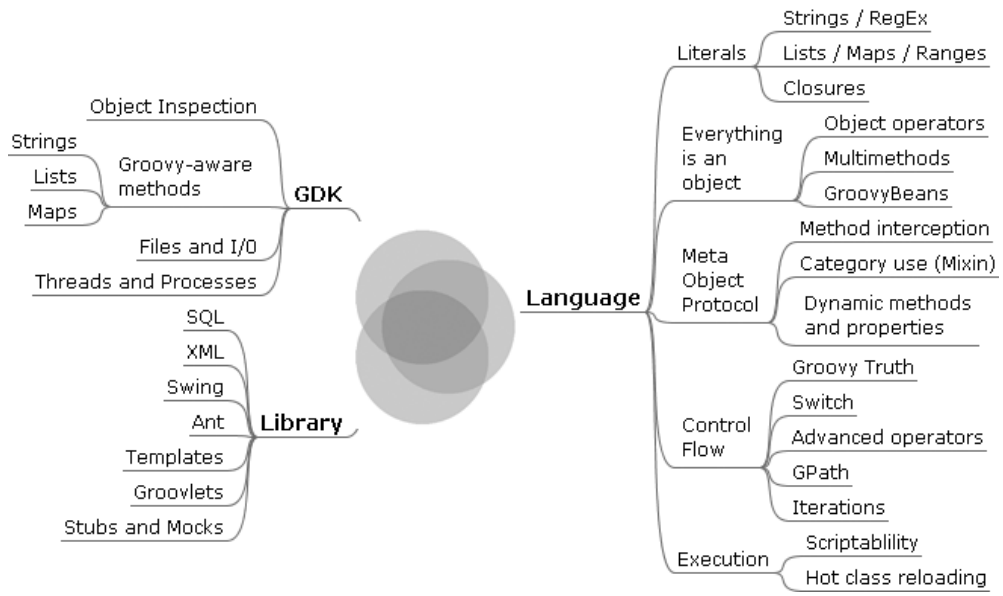


Figure 1.3 Many of the additional libraries and JDK enhancements in Groovy build on the new language features. The combination of the three forms a "sweet spot" for clear and powerful code.

language features. Idiomatic Groovy code rarely uses one feature in isolation—instead, it usually uses several of them together, like notes in a chord.

Unfortunately, many of the features can't be understood in just a few words. *Closures,* for example, are an invaluable language concept in Groovy, but the word on its own doesn't tell you anything. We won't go into all the details now, but here are a few examples to whet your appetite.

Listing a file: closures and I/O additions

Closures are blocks of code that can be treated as first-class objects: passed around as references, stored, executed at arbitrary times, and so on. Java's anonymous inner classes are often used this way, particularly with adapter classes, but the syntax of inner classes is ugly, and they're limited in terms of the data they can access and change.

File handling in Groovy is made significantly easier with the addition of various methods to classes in the `java.io` package. A great example is the `File.eachLine` method. How often have you needed to read a file, a line at a time, and perform the same action on each line, closing the file at the end? This is such a common task, it shouldn't be difficult—so in Groovy, it isn't.

Let's put the two features together and create a complete program that lists a file with line numbers:

```
def number=0
new File ('test.groovy').eachLine { line ->
    number++
    println "$number: $line"
}
```

The closure in curly braces gets executed for each line, and `File`'s new `eachLine` method makes this happen.

Printing a list: collection literals and simplified property access

`java.util.List` and `java.util.Map` are probably the most widely used interfaces in Java, but there is little language support for them. Groovy adds the ability to declare list and map literals just as easily as you would a string or numeric literal, and it adds many methods to the collection classes.

Similarly, the JavaBean conventions for properties are almost ubiquitous in Java, but the language makes no use of them. Groovy simplifies property access, allowing for far more readable code.

Here's an example using these two features to print the package for each of a list of classes. Note that the word *package* needs to be quoted because it's a keyword, but it can still be used for the property name. Although Java would allow a

similar first line to declare an array, we're using a real list here—elements could be added or removed with no extra work:

```
def classes = [String, List, File]
for (clazz in classes)
{
    println clazz.'package'.name
}
```

In Groovy, you can even avoid such commonplace `for` loops by applying property access to a list—the result is a list of the properties. Using this feature, an equivalent solution to the previous code is

```
println( [String, List, File].'package'.name )
```

to produce the output

```
["java.lang", "java.util", "java.io"]
```

Pretty cool, eh?

XML handling the Groovy way: GPath with dynamic properties

Whether you're reading it or writing it, working with XML in Java requires a considerable amount of work. Alternatives to the W3C DOM make life easier, but Java itself doesn't help you in language terms—it's unable to adapt to your needs. Groovy allows classes to act as if they have properties at runtime even if the names of those properties aren't known when the class is compiled. `GPath` was built on this feature, and it allows seamless XPath-like navigation of XML documents.

Suppose you have a file called customers.xml such as this:

```
<?xml version="1.0" ?>
<customers>
  <corporate>
    <customer name="Bill Gates"        company="Microsoft" />
    <customer name="Steve Jobs"        company="Apple" />
    <customer name="Jonathan Schwartz" company="Sun" />
  </corporate>

  <consumer>
    <customer name="John Doe" />
    <customer name="Jane Doe" />
  </consumer>
</customers>
```

You can print out all the corporate customers with their names and companies using just the following code. (Generating the file in the first place with Groovy using a `Builder` would be considerably easier than in Java, too.)

```
def customers = new XmlSlurper().parse(new File('customers.xml'))
for (customer in customers.corporate.customer)
{
    println "${customer.@name} works for ${customer.@company}"
}
```

Even trying to demonstrate just a few features of Groovy, you've seen other features in the preceding examples—string interpolation with `GString`, simpler `for` loops, optional typing, and optional statement terminators and parentheses, just for starters. The features work so well with each other and become second nature so quickly, you hardly notice you're using them.

Although being Java friendly and feature rich are the main driving forces for Groovy, there are more aspects worth considering. So far, we have focused on the hard technical facts about Groovy, but a language needs more than that to be successful. It needs to *attract* people. In the world of computer languages, building a better mousetrap doesn't guarantee that the world will beat a path to your door. It has to appeal to both developers and their managers, in different ways.

1.1.4 Community-driven but corporate-backed

For some people, it's comforting to know that their investment in a language is protected by its adoption as a standard. This is one of the distinctive promises of Groovy. Since the passage of JSR-241, Groovy is the second standard language for the Java platform (the first being the Java language).

The size of the user base is a second criterion. The larger the user base, the greater the chance of obtaining good support and sustainable development. Groovy's user base is reasonably sized. A good indication is the activity on the mailing lists and the number of related projects (see http://groovy.codehaus.org/Related+Projects).

Attraction is more than strategic considerations, however. Beyond what you can measure is a gut feeling that causes you to enjoy programming *or not*.

The developers of Groovy are aware of this feeling, and it is carefully considered when deciding upon language features. After all, there is a reason for the name of the language.

GROOVY "A situation or an activity that one enjoys or to which one is especially well suited (found his groove playing bass in a trio). A very pleasurable experience; enjoy oneself (just sitting around, grooving on the music). To be affected with pleasurable excitement. To react or interact harmoniously." [Leo]

Someone recently stated that Groovy was, "Java-stylish with a Ruby-esque feeling." We cannot think of a better description. Working with Groovy feels like a partnership between you and the language, rather than a battle to express what is clear in your mind in a way the computer can understand.

Of course, while it's nice to "feel the groove," you still need to pay your bills. In the next section, we'll look at some of the practical advantages Groovy will bring to your professional life.

1.2 What Groovy can do for you

Depending on your background and experience, you are probably interested in different features of Groovy. It is unlikely that anyone will require every aspect of Groovy in their day-to-day work, just as no one uses the whole of the mammoth framework provided by the Java standard libraries.

This section presents interesting Groovy features and areas of applicability for Java professionals, script programmers, and pragmatic, extreme, and agile programmers. We recognize that developers rarely have just one role within their jobs and may well have to take on each of these identities in turn. However, it is helpful to focus on how Groovy helps in the kinds of situations typically associated with each role.

1.2.1 Groovy for Java professionals

If you consider yourself a Java professional, you probably have years of experience in Java programming. You know all the important parts of the Java Runtime API and most likely the APIs of a lot of additional Java packages.

But—be honest—there are times when you cannot leverage this knowledge, such as when faced with an everyday task like recursively searching through all files below the current directory. If you're like us, programming such an ad-hoc task in Java is just too much effort.

But as you will learn in this book, with Groovy you can quickly open the console and type

```
groovy -e "new File('.').eachFileRecurse { println it }"
```

to print all filenames recursively.

Even if Java had an `eachFileRecurse` method and a matching `FileListener` interface, you would still need to explicitly create a class, declare a `main` method, save the code as a file, and compile it, and only then could you run it. For the sake of comparison, let's see what the Java code would look like, assuming the existence of an appropriate `eachFileRecurse` method:

```
public class ListFiles {                          // JAVA !!
    public static void main(String[] args) {
        new java.io.File(".").eachFileRecurse(
            new FileListener() {
                public void onFile (File file) {
                    System.out.println(file.toString());
                }
            }
        );
    }
}
```

Imagine Java had this

Notice how the intent of the code (printing each file) is obscured by the scaffolding code Java requires you to write in order to end up with a complete program.

Besides command-line availability and code beauty, Groovy allows you to bring dynamic behavior to Java applications, such as through expressing business rules, allowing smart configurations, or even implementing *domain specific languages*.

You have the options of using static or dynamic types and working with precompiled code or plain Groovy source code with on-demand compiling. As a developer, you can decide where and when you want to put your solution "in stone" and where it needs to be flexible. With Groovy, you have the choice.

This should give you enough safeguards to feel comfortable incorporating Groovy into your projects so you can benefit from its features.

1.2.2 Groovy for script programmers

As a script programmer, you may have worked in Perl, Ruby, Python, or other dynamic (non-scripting) languages such as Smalltalk, Lisp, or Dylan.

But the Java platform has an undeniable market share, and it's fairly common that folks like you work with the Java language to make a living. Corporate clients often run a Java standard platform (e.g. J2EE), allowing nothing but Java to be developed and deployed in production. You have no chance of getting your ultra-slick scripting solution in there, so you bite the bullet, roll up your sleeves, and dig through endless piles of Java code, thinking all day, "If I only had [*your language here*], I could replace this whole method with a single line!" We confess to having experienced this kind of frustration.

Groovy can give you relief and bring back the fun of programming by providing advanced language features where you need them: in your daily work. By allowing you to call methods on *anything*, pass blocks of code around for immediate or later execution, augment existing library code with your own specialized semantics, and use a host of other powerful features, Groovy lets you express yourself clearly and achieve miracles with little code.

Just sneak the groovy-all-*.jar file into your project's classpath, and you're there.

Today, software development is seldom a solitary activity, and your teammates (and your boss) need to know what you are doing with Groovy and what Groovy is about. This book aims to be a device you can pass along to others so they can learn, too. (Of course, if you can't bear the thought of parting with it, you can tell them to buy their own copies. We won't mind.)

1.2.3 Groovy for pragmatic programmers, extremos, and agilists

If you fall into this category, you probably already have an overloaded bookshelf, a board full of index cards with tasks, and an automated test suite that threatens to turn red at a moment's notice. The next iteration release is close, and there is anything but time to think about Groovy. Even uttering the word makes your pair-programming mate start questioning your state of mind.

One thing that we've learned about being pragmatic, extreme, or agile is that every now and then you have to step back, relax, and assess whether your tools are still *sharp* enough to cut smoothly. Despite the ever-pressing project schedules, you need to *sharpen the saw* regularly. In software terms, that means having the knowledge and resources needed and using the right methodology, tools, technologies, and languages for the task at hand.

Groovy will be an invaluable tool in your box for all automation tasks that you are likely to have in your projects. These range from simple build automation, continuous integration, and reporting, up to automated documentation, shipment, and installation. The Groovy automation support leverages the power of existing solutions such as Ant and Maven, while providing a simple and concise language means to control them. Groovy even helps with testing, both at the unit and functional levels, helping us test-driven folks feel right at home.

Hardly any school of programmers applies as much rigor and pays as much attention as we do when it comes to self-describing, intention-revealing code. We feel an almost physical need to remove duplication while striving for simpler solutions. This is where Groovy can help tremendously.

Before Groovy, I (Dierk) used other scripting languages (preferably Ruby) to sketch some design ideas, do a *spike*—a programming experiment to assess the feasibility of a task—and run a functional prototype. The downside was that I was never sure if what I was writing would *also* work in Java. Worse, in the end I had the work of porting it over or redoing it from scratch. With Groovy, I can do all the exploration work *directly* on my target platform.

EXAMPLE Recently, Guillaume and I did a spike on *prime number disassembly.*[2] We started with a small Groovy solution that did the job cleanly but not efficiently. Using Groovy's interception capabilities, we unit-tested the solution and counted the number of operations. Because the code was clean, it was a breeze to optimize the solution and decrease the operation count. It would have been much more difficult to recognize the optimization potential in Java code. The final result can be used from Java as it stands, and although we certainly still have the option of porting the optimized solution to plain Java, which would give us another performance gain, we can defer the decision until the need arises.

The seamless interplay of Groovy and Java opens two dimensions of optimizing code: using Java for code that needs to be optimized for runtime performance, and using Groovy for code that needs to be optimized for flexibility and readability.

Along with all these tangible benefits, there is value in learning Groovy for its own sake. It will open your mind to new solutions, helping you to perceive new concepts when developing software, whichever language you use.

No matter what kind of programmer you are, we hope you are now eager to get some Groovy code under your fingers. If you cannot hold back from looking at some real Groovy code, look at chapter 2.

1.3 Running Groovy

First, we need to introduce you to the tools you'll be using to run and optionally compile Groovy code. If you want to try these out as you read, you'll need to have Groovy installed, of course. Appendix A provides a guide for the installation process.

There are three commands to execute Groovy code and scripts, as shown in table 1.1. Each of the three different mechanisms of running Groovy is demonstrated in the following sections with examples and screenshots. Groovy can also be "run" like any ordinary Java program, as you will see in section 1.4.2, and there also is a special integration with Ant that is explained in section 1.4.3.

We will explore several options of integrating Groovy in Java programs in chapter 11.

[2] Every ordinal number N can be uniquely disassembled into factors that are prime numbers: N = p1*p2*p3. The disassembly problem is known to be "hard." Its complexity guards cryptographic algorithms like the popular Rivest-Shamir-Adleman (RSA) algorithm.

Table 1.1 Commands to execute Groovy

Command	What it does
`groovysh`	Starts the `groovysh` command-line shell, which is used to execute Groovy code interactively. By entering statements or whole scripts, line by line, into the shell and giving the `go` command, code is executed "on the fly."
`groovyConsole`	Starts a graphical interface that is used to execute Groovy code interactively; moreover, `groovyConsole` loads and runs Groovy script files.
`groovy`	Starts the interpreter that executes Groovy scripts. Single-line Groovy scripts can be specified as command-line arguments.

1.3.1 Using groovysh for "Hello World"

Let's look at `groovysh` first because it is a handy tool for running experiments with Groovy. It is easy to edit and run Groovy iteratively in this shell, and doing so facilitates seeing how Groovy works without creating and editing script files.

To start the shell, run `groovysh` (UNIX) or `groovysh.bat` (Windows) from the command line. You should then get a command prompt like this:

```
Lets get Groovy!
================
Version: 1.0-RC-01-SNAPSHOT JVM: 1.4.2_05-b04
Type 'exit' to terminate the shell
Type 'help' for command help
Type 'go' to execute the statements

groovy>
```

The traditional "Hello World!" program can be written in Groovy with one line and then executed in `groovysh` with the `go` command:

```
groovy> "Hello, World!"
groovy> go

===> Hello, World!
```

The `go` command is one of only a few commands the shell recognizes. The rest can be displayed by typing `help` on the command line:

```
groovy> help
Available commands (must be entered without extraneous characters):
exit/quit     - terminates processing
help          - displays this help text
```

```
discard         - discards the current statement
display         - displays the current statement
explain         - explains the parsing of the current statement (currently
                  disabled)
execute/go      - temporary command to cause statement execution
binding         - shows the binding used by this interactive shell
discardclasses - discards all former unbound class definitions
inspect         - opens ObjectBrowser on expression returned from
                  previous "go"
```

The `go` and `execute` commands are equivalent. The `discard` command tells Groovy to forget the last line typed, which is useful when you're typing in a long script, because the command facilitates clearing out the small sections of code rather than having to rewrite an entire script from the top. Let's look at the other commands.

Display command

The `display` command displays the last noncommand statement entered:

```
groovy> display
1> "Hello World!"
```

Binding command

The `binding` command displays variables utilized in a `groovysh` session. We haven't used any variables in our simple example, but, to demonstrate, we'll alter our "Hello World!" using the variable `greeting` to hold part of the message we print out:

```
groovy> greeting = "Hello"
groovy> "${greeting}, World!"
groovy> go

===> Hello, World!

groovy> binding
Available variables in the current binding
greeting = Hello
```

The `binding` command is useful when you're in the course of a longer `groovysh` session and you've lost track of the variables in use and their current values.

To clear the binding, exit the shell and start a new one.

Inspect command

The `inspect` command opens the *Groovy Object Browser* on the last evaluated expression. This browser is a Swing user interface that lets you browse through an object's native Java API and any additional features available to it via Groovy's

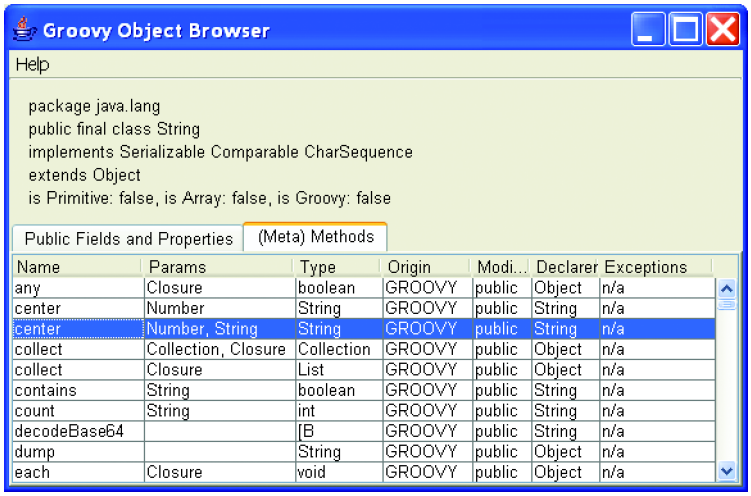


Figure 1.4 The Groovy Object Browser when opened on an object of type `String`, displaying the table of available methods in its bytecode and registered Meta methods

GDK. Figure 1.4 shows the Object Browser inspecting an instance of `String`. It contains information about the `String` class in the header and two tables showing available methods and fields.

Look at the second and third rows. A method with the name `center` is available on a `String` object. It takes a `Number` parameter (second row) and an optional `String` parameter (third row). The method's return type is a `String`. Groovy defined this new public method on the `String` class.

If you are anything like us, you cannot wait to try that new knowledge in the `groovysh` and type

```
groovy> 'test'.center 20, '-'
groovy> go

===> --------test--------
```

That's almost as good as IDE support!

For easy browsing, you can sort columns by clicking the headers and reverse the sort with a second click. You can sort by multiple criteria by clicking column headers in sequence, and rearrange the columns by dragging the column headers.

Future versions of the Groovy Object Browser may provide even more sophisticated features.

1.3.2 Using groovyConsole

The `groovyConsole` is a Swing interface that acts as a minimal Groovy interactive interpreter. It lacks support for the command-line options supported by `groovysh`; however, it has a File menu to allow Groovy scripts to be loaded, created, and saved. Interestingly, `groovyConsole` is written in Groovy. Its implementation is a good demonstration of Builders, which are discussed in chapter 7.

The `groovyConsole` takes no arguments and starts a two-paned Window like the one shown in figure 1.5. The console accepts keyboard input in the upper pane. To run a script, either key in Ctrl+R, Ctrl+Enter or use the Run command from the Action menu to run the script. When any part of the script code is selected, only the selected text is executed. This feature is useful for simple debugging or *single stepping* by successively selecting one or multiple lines.

The `groovyConsole`'s File menu has New, Open, Save, and Exit commands. New opens a new `groovyConsole` window. Open can be used to browse to a Groovy script on the file system and open it in the edit pane for editing and running. Save can be used to save the current text in the edit pane to a file. Exit quits the `groovyConsole`.

The Groovy Object Browser as shown in figure 1.4 is equally available in `groovyConsole` and also operates on the last evaluated expression. To open the browser, press Ctrl+I (for *inspect*) or choose Inspect from the Actions menu.

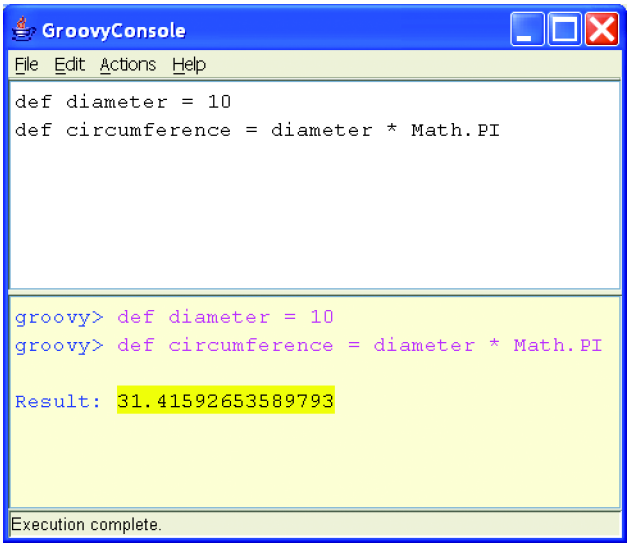


Figure 1.5
The `groovyConsole` with a simple script in the edit pane that calculates the circumference of a circle based on its diameter. The result is in the output pane.

That's it for `groovyConsole`. Whether you prefer working in `groovysh` or `groovyConsole` is a personal choice. Script programmers who perform their work in command shells tend to prefer the shell.

AUTHOR'S CHOICE I (Dierk) personally changed my habits to use the console more often for the sake of less typing through cut-and-paste in the edit pane.

Unless explicitly stated otherwise, you can put any code example in this book directly into `groovysh` or `groovyConsole` and run it there. The more often you do that, the earlier you will get a feeling for the language.

1.3.3 Using groovy

The `groovy` command is used to execute Groovy programs and scripts. For example, listing 1.1 shows the obligatory Fibonacci[3] number sequence Groovy program that prints the first 10 Fibonacci numbers. The Fibonacci number sequence is a pattern where the first two numbers are 1 and 1, and every subsequent number is the sum of the preceding two.

If you'd like to try this, copy the code into a file, and save it as Fibonacci.groovy. The file extension does not matter much as far as the `groovy` executable is concerned, but naming Groovy scripts with a .groovy extension is conventional. One benefit of using an extension of .groovy is that you can omit it on the command line when specifying the name of the script—instead of `groovy MyScript.groovy`, you can just run `groovy MyScript`.

Listing 1.1 Fibonacci.groovy

```
current = 1
next    = 1
10.times {          // loop 10 times
   print current + ' '
   newCurrent = next
   next = next + current
   current = newCurrent
}
println ''
```

[3] Leonardo Pisano (1170..1250), aka Fibonacci, was a mathematician from Pisa (now a town in Italy). He introduced this number sequence to describe the growth of an isolated rabbit population. Although this may be questionable from a biological point of view, his number sequence plays a role in many different areas of science and art. For more information, you can subscribe to the *Fibonacci Quarterly*.

Run this file as a Groovy program by passing the file name to the `groovy` command. You should see the following output:

```
> groovy Fibonacci
1 1 2 3 5 8 13 21 34 55
```

The `groovy` command has many additional options that are useful for command-line scripting. For example, expressions can be executed by typing `groovy -e "println 1+1"`, which prints `2` to the console. Section 12.3 will lead you through the full range of options, with numerous examples.

In this section, we have dealt with Groovy's support for simple ad-hoc scripting, but this is not the whole story. The next section expands on how Groovy fits into a code-compile-run cycle.

1.4 Compiling and running Groovy

So far, we have used Groovy in *direct* mode, where our code is directly executed without producing any executable files. In this section, you will see a second way of using Groovy: compiling it to Java bytecode and running it as regular Java application code within a Java Virtual Machine (JVM). This is called *precompiled* mode. Both ways execute Groovy inside a JVM eventually, and both ways compile the Groovy code to Java bytecode. The major difference is *when* that compilation occurs and whether the resulting classes are used in memory or stored on disk.

1.4.1 Compiling Groovy with groovyc

Compiling Groovy is straightforward, because Groovy comes with a compiler called `groovyc`. The `groovyc` compiler generates at least one class file for each Groovy source file compiled. As an example, we can compile Fibonacci.groovy from the previous section into normal Java bytecode by running `groovyc` on the script file like so:

```
> groovyc -d classes Fibonacci.groovy
```

In our case, the Groovy compiler outputs two Java class files to a directory named classes, which we told it to do with the `-d` flag. If the directory specified with `-d` does not exist, it is created. When you're running the compiler, the name of each generated class file is printed to the console.

For each script, `groovyc` generates a class that extends `groovy.lang.Script`, which contains a `main` method so that `java` can execute it. The name of the compiled class matches the name of the script being compiled.

More classes may be generated, depending on the script code; however, we don't really need to care about that because that is a Java platform topic. In essence, `groovyc` works the same way that `javac` compiles nested classes.

NOTE The Fibonacci script contains the `10.times{}` construct that causes `groovyc` to generate a class of type *closure,* which implements what is inside the curly braces. This class is nested inside the Fibonacci class. You will learn more about closures in chapter 5. If you find this confusing, you can safely ignore it for the time being.

The mapping of class files to implementations is shown in table 1.2, with the purpose of each explained.

Table 1.2 Classes generated by `groovyc` for the Fibonacci.groovy file

Class file	Is a subclass of ...	Purpose
`Fibonacci.class`	`groovy.lang.Script`	Contains a `main` method that can be run with the `java` command.
`Fibonacci$_run_closure1.class`	`groovy.lang.Closure`	Captures what has to be done 10 *times.* You can safely ignore it.

Now that we've got a compiled program, let's see how to run it.

1.4.2 Running a compiled Groovy script with Java

Running a compiled Groovy program is identical to running a compiled Java program, with the added requirement of having the embeddable groovy-all*.jar file in your JVM's classpath, which will ensure that all of Groovy's third-party dependencies will be resolved automatically at runtime. Make sure you add the directory in which your compiled program resides to the classpath, too. You then run the program in the same way you would run any other Java program, with the `java` command.[4]

```
> java -cp %GROOVY_HOME%/embeddable/groovy-all-1.0.jar;classes Fibonacci
1 1 2 3 5 8 13 21 34 55
```

Note that the .class file extension for the main class should not be specified when running with `java`.

[4] The command line as shown applies to Windows shells. The equivalent on Linux/Solaris/UNIX/Cygwin would be

```
java -cp $GROOVY_HOME/embeddable/groovy-all-1.0.jar:classes Fibonacci
```

All this may seem like a lot of work if you're used to building and running your Java code with Ant at the touch of a button. We agree, which is why the developers of Groovy have made sure you can do all of this easily in an Ant script.

1.4.3 Compiling and running with Ant

An Ant task is shipped with Groovy for running the `groovyc` compiler in an Ant build script. To use it, you need to have Ant installed.[5] We recommend version 1.6.2 or higher.

Listing 1.2 shows an Ant build script, which compiles and runs the Fibonacci.groovy script as Java bytecode.

Listing 1.2 build.xml for compiling and running a Groovy program as Java bytecode

```
<project name="fibonacci-build" default="run">

  <property environment="env"/>

  <path id="groovy.classpath">
    <fileset dir="${env.GROOVY_HOME}/embeddable/"/>
  </path>

  <taskdef name ="groovyc"
    classname   ="org.codehaus.groovy.ant.Groovyc"
    classpathref="groovy.classpath"/>

  <target name="compile"
    description="compile groovy to bytecode">
    <mkdir  dir="classes"/>
    <groovyc
      destdir="classes"
      srcdir="."
      includes="Fibonacci.groovy"
      classpathref="groovy.classpath">
    </groovyc>
  </target>

  <target name="run" depends="compile"
    description="run the compiled class">
    <java classname="Fibonacci">
      <classpath refid="groovy.classpath"/>
      <classpath location="classes"/>
    </java>
  </target>
</project>
```

1 Path definition

2 taskdef

3 compile target

4 run target

[5] Groovy ships with its own copy of the Ant jar files that could also be used for this purpose, but it is easier to explain with a standalone installation of Ant.

Store this file as build.xml in your current directory, which should also contain the Fibonacci.groovy script, and type `ant` at the command prompt.

The build will start at the ❹ `run` target, which depends on the ❸ `compile` target and therefore calls that one first. The `compile` target is the one that uses the `groovyc` task. In order to make this task known to Ant, the ❷ `taskdef` is used. It finds the implementation of the `groovyc` task by referring to the `groovy.classpath` in the ❶ path definition.

When everything compiles successfully in the ❸ `compile` target, the ❹ `run` target calls the `java` task on the compiled classes.

You will see output like this:

```
> ant
Buildfile: build.xml

compile:
    [mkdir] Created dir: …\classes
  [groovyc] Compiling 1 source file to …\classes
run:
     [java] 1 1 2 3 5 8 13 21 34 55

  BUILD SUCCESSFUL
Total time: 2 seconds
```

Executing `ant` a second time shows no compile output, because the `groovyc` task is smart enough to compile *only when necessary*. For a *clean* compile, you have to delete the destination directory before compiling.

The `groovyc` Ant task has a lot of options, most of which are similar to those in the `javac` Ant task. The `srcdir` and `destdir` options are mandatory.

Using `groovyc` for compilation can be handy when you're integrating Groovy in Java projects that use Ant (or Maven) for build automation. More information about integrating Groovy with Ant and Maven will be given in chapter 14.

1.5 Groovy IDE and editor support

If you plan to code in Groovy often, you should look for Groovy support for your IDE or editor of choice. Some editors only support syntax highlighting for Groovy at this stage, but even that can be useful and can make Groovy code more convenient to work with. Some commonly used IDEs and text editors for Groovy are listed in the following sections.

This section is likely to be out of date as soon as it is printed. Stay tuned for updates for your favorite IDE, because improved support for Groovy in the major Java IDEs is expected in the near future. Sun Microsystems recently announced

Groovy support for its NetBeans *coyote* project (https://coyote.dev.java.net/), which is particularly interesting because it is the first IDE support for Groovy that is managed by the IDE's own vendor itself.

1.5.1 IntelliJ IDEA plug-in

Within the Groovy community, work is ongoing to develop an open-source plug-in called GroovyJ. With the help of this plug-in and IDEA's built-in features, a Groovy programmer can benefit from the following:

- Simple syntax highlighting based on user preferences: GroovyJ currently uses Java 5's syntax highlighter, which covers a large proportion of the Groovy syntax. Version 1.0 will recognize the full Groovy syntax and allow customization of the highlighting through the Colors & Fonts panel, just as it is possible with the Java syntax.
- Code completion: To date, code completion is limited to word completion, leveraging IDEA's word completion based on an on-the-fly dictionary for the current editor only.
- Tight integration with IDEA's *compile*, *run*, *build*, and *make* configuration as well as output views.
- Lots of advanced editor actions that can be used as in Java.
- Efficient lookup for all related Java classes in the project or dependent libraries.
- Efficient navigation between files, including .groovy files.
- A Groovy file-type icon.

GroovyJ has a promising future, which is greatly dependent on its implementation of IDEA's *Program Structure Interface (PSI)* for the Groovy language. It will do so by specializing the Groovy grammar file and generating a specialized parser for this purpose. Because IDEA bases all its advanced features (such as refactoring support, inspections, navigation, intentions, and so forth) on the PSI, it seems to be only a matter of time before we will see these features for Groovy.

GroovyJ is an interesting project, mindfully led by Franck Rasolo. This plug-in is one of the most advanced ones available to Groovy at this point. For more information, see http://groovy.codehaus.org/GroovyJ+Status.

1.5.2 Eclipse plug-in

The Groovy plug-in for Eclipse requires Eclipse 3.1.1 or newer. The plug-in will also run in Eclipse 3.x-derived tools such as IBM Rational's Rational Application Developer and Rational Software Architect. As of this writing, the Groovy Eclipse plug-in supports the following features:

- Syntax highlighting for Groovy files
- A Groovy file decorator (icon) for Groovy files in the Package Explorer and Resources views
- Running Groovy scripts from within the IDE
- Auto-build of Groovy files
- Debugger integration

The Groovy Eclipse plug-in is available for download at http://groovy.codehaus.org/Eclipse+Plugin.

1.5.3 Groovy support in other editors

Although they don't claim to be full-featured development environments, a lot of all-purpose editors provide support for programming languages in general and Groovy in particular.

UltraEdit can easily be customized to provide syntax highlighting for Groovy and to start or compile scripts from within the editor. Any output goes to an integrated output window. A small sidebar lets you jump to class and method declarations in the file. It supports smart indentation and brace matching for Groovy. Besides the Groovy support, it is a feature-rich, quick-starting, all-purpose editor. Find more details at http://groovy.codehaus.org/UltraEdit+Plugin.

The *JEdit* plug-in for Groovy supports executing Groovy scripts and code snippets from within the editor. A syntax-highlighting configuration is available separately. More details are available here: http://groovy.codehaus.org/JEdit+Plugin.

Syntax highlighting configuration files for TextPad, Emacs, Vim, and several other text editors can be found on the Groovy web site at http://groovy.codehaus.org/Other+Plugins.

AUTHOR'S CHOICE When programming small ad-hoc Groovy scripts, I (Dierk) personally use UltraEdit on Windows and Vim on Linux. For any project of some size, I use IntelliJ IDEA with the GroovyJ plug-in.

As Groovy matures and is adopted among Java programmers, it will continue to gain support in Java IDEs with features such as debugging, unit testing, and dynamic code-completion.

1.6 Summary

We hope that by now we've convinced you that you really want Groovy in your life. As a modern language built on the solid foundation of Java and with support from Sun, Groovy has something to offer for everyone, in whatever way they interact with the Java platform.

With a clear idea of why Groovy was developed and what drives its design, you should be able to see where features fit into the bigger picture as each is introduced in the coming chapters. Keep in mind the principles of Java integration and feature richness, making common tasks simpler and your code more expressive.

Once you have Groovy installed, you can run it both directly as a script and after compilation into classes. If you have been feeling energetic, you may even have installed a Groovy plug-in for your favorite IDE. With this preparatory work complete, you are ready to see (and try!) more of the language itself. In the next chapter, we will take you on a whistle-stop tour of Groovy's features to give you a better feeling for the shape of the language, before we examine each element in detail for the remainder of part 1.

JAVA

"... a clear and detailed exposition of what is groovy about Groovy. I'm glad to have it on my bookshelf."
—from the *Foreword* by James Gosling

Groovy IN ACTION

Dierk König, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet

Combine the easy-to-use dynamic features of Ruby or Python with the power and stability of the Java platform and the result is, well, Groovy. Groovy is a dynamic language that looks like Java down to its curly brackets. It compiles straight to bytecode, and works seamlessly with existing Java objects and libraries. With this single tool developers can both execute command-line scripts and build commercial-grade applications.

Groovy in Action is a fast-paced tutorial covering the Groovy language and how and when to apply it. Java developers will master Groovy's enhancements to Java such as builders, template engines, and support for regular expressions and database programming. The book includes dozens of practical examples. It provides tips & tricks for daily work, unit testing, build support, and even scripting Windows.

What's Inside

- A comprehensive Groovy language tutorial
- Explore the benefits of dynamic programming
- Tackle day-to-day tasks like shell scripting and build support
- Grails, the Groovy Web Development framework
- Dozens of reusable examples

Dierk König has been a committer on the Groovy project since 2004. **Guillaume Laforge** is the official Groovy project manager and leader of the Expert Group standardizing Groovy. Technology strategist **Andrew Glover**, speaker and consultant **Dr. Paul King**, and **Jon Skeet**, a Java and C# expert, round out the authoring team.

"Excellent code samples ... very readable."
—Scott Shaw, ThoughtWorks

"Top of my list."
—Samuel Pullara, VP Technology Strategy, Yahoo, Inc.

"Collects in one place details of the language and its libraries—a valuable resource."
—John Wilson
The Wilson Partnership

"Great, logical focus on language features."
—Norman Richards
JBoss Developer, author of *XDoclet in Action*

"Destined to be the definitive guide. First rate!"
—Glen Smith, Bytecode Pty Ltd

"You want to learn Groovy? This book has all you need."
—Stuart Caborn, ThoughtWorks

www.manning.com/koenig

ISBN 1-932394-84-2

$49.99 US/$64.99 Canada