Extracted from:

Programming Groovy 2

Dynamic Productivity for the Java Developer

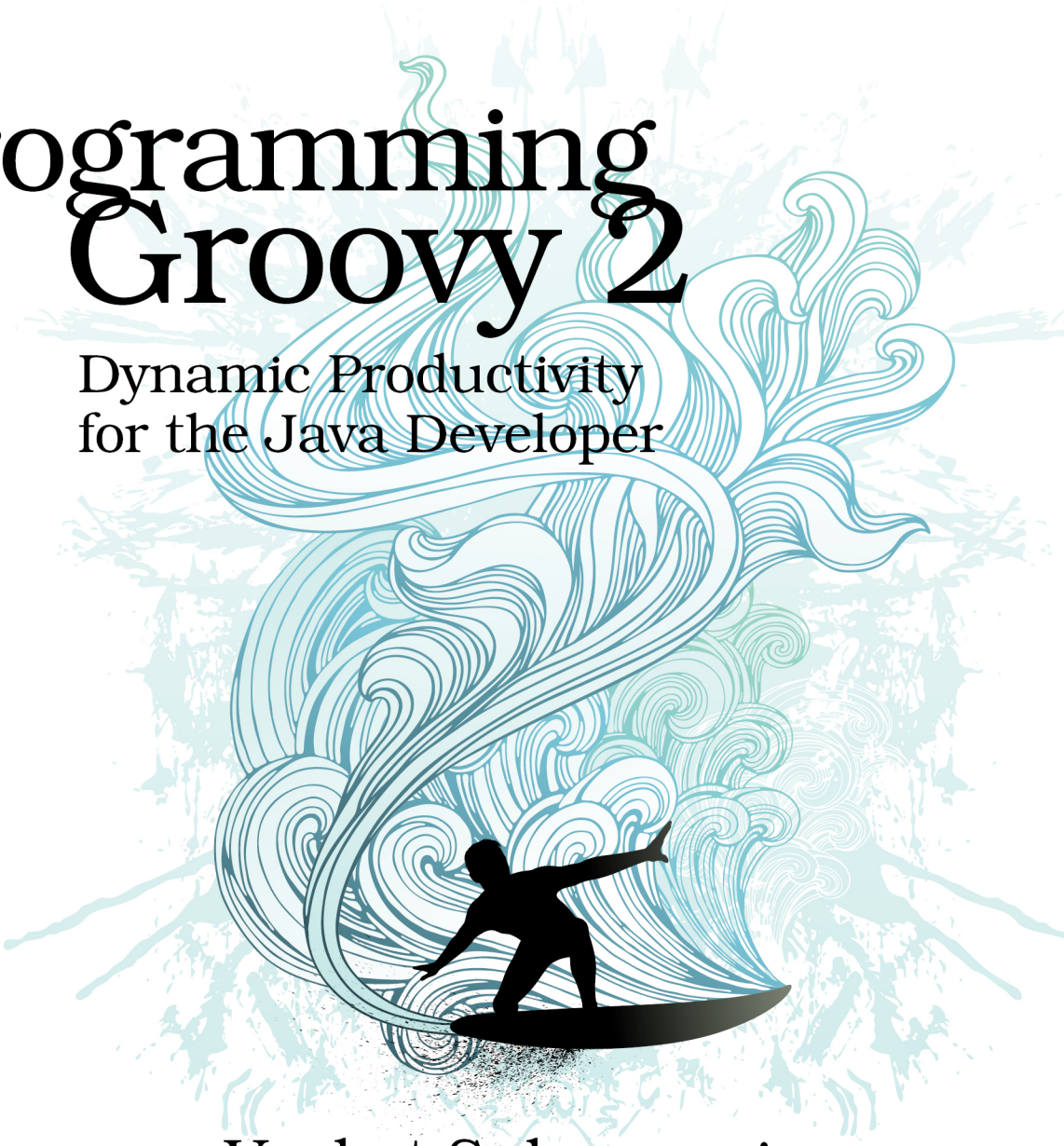
The Pragmatic Bookshelf

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

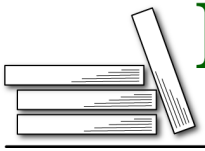
Programming Groovy 2

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.

Introduction

The Java platform is arguably one of the most powerful and widely adopted ecosystems today. It has three significant pieces:

- The Java Virtual Machine (JVM), which has become increasingly powerful and more performant over the years
- The Java Development Kit (JDK), the rich set of third-party libraries and frameworks that help us effectively leverage the power of the platform
- The set of languages on the JVM—the Java language being the first—that help us program the platform

Languages are like vehicles that let us navigate the platform. They let us reach into various parts of this landscape with ease. The Java language has come a long way; its libraries have been refactored and expanded. It's gotten us this far, but we need to look beyond the Java language to languages that are lightweight and that can make us more productive. When used correctly, dynamic languages, the functional style of programming, and metaprogramming capabilities can help us navigate the landscape much faster. When viewed as vehicles, these newer languages aren't faster cars; they're flying machines, giving us the capability to be several orders of magnitude more productive.

The Java language has been flirting with metaprogramming and the functional style of programming for a while and will support some of these features to various degrees in future versions. We don't have to wait for that day, however. We can build performant JVM applications with all the dynamic capabilities today, right now, using Groovy.

What's Groovy?

Merriam-Webster defines *groovy* as "marvelous, wonderful, excellent, hip, trendy." The Groovy language is all of that—it's lightweight, low-ceremony, dynamic, object-oriented, and runs on the JVM. Groovy is open sourced under

the Apache License, version 2.0. It derives strength from various 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 1.5 and newer. For deployment, all we need is a Groovy Java archive (JAR) in addition to the regular Java stuff, and we're all set.

Groovy is a "language that has been reborn several times."[1] James Strachan and Bob McWhirter started it in 2003, and it was commissioned into Java Specification Request (JSR) 241 in March 2004. Soon afterward, it was almost abandoned because of 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 for a while. A number of people, including committers and users, simply gave up on the language. Finally, a group of smart and enthusiastic developers joined forces with Guillaume and Jeremy, and a vibrant developer community emerged.

The release of Groovy version 1.0 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. Organizations and developers are beginning to use Groovy at various levels on their projects, and the time is ripe for major Groovy adoption in the industry. Groovy version 2.0 was released in mid 2012.

Groovy shines in tools and frameworks like Grails, CodeNarc, easyb, Gradle, and Spock. Grails, a dynamic web-development framework based on "coding by convention," exploits Groovy metaprogramming.[2] Using Grails, we can quickly build web applications on the JVM using Groovy, Spring, Hibernate, and other Java frameworks.

Why Dynamic Languages?

Dynamic languages have the ability to extend a program at runtime, including changing types, behaviors, and object structures. With these languages, we can do things at runtime that static languages do at compile time; we can even execute program statements that are created on the fly at runtime.

For example, if we want to compute a five percent raise on an $80,000 salary, we could simply write the following:

1. See "A bit of Groovy history," a blog post by Guillaume Laforge at http://glaforge.free.fr/weblog/index.php?itemid=99.
2. http://grails.org

```
5.percentRaise(80000)
```

Yes, that's the friendly java.lang.Integer responding to our own dynamic method, which we can add quite easily, like so:

```
Integer.metaClass.percentRaise = { amount -> amount * (1 + delegate / 100.0) }
```

As we see here, it's easy to add dynamic methods to classes in Groovy. The dynamic method we added to the Integer instance, referred using the delegate variable, returns the dollar amount increased by the appropriate percentage.

The flexibility of dynamic languages gives us the advantage of evolving programs as the applications execute. This goes far beyond code generation. We should consider code generation to be soooo twentieth century. In fact, generated code is like an incessant itch; if we keep scratching it, it turns into a sore. With dynamic languages, there are better ways. Dynamic languages make it easier to prefer *code synthesis*, which is in-memory code-creation at runtime. The code is synthesized based on the flow of logic through the application and becomes active *just in time*.

By carefully applying dynamic languages' capabilities, we can be more productive as application developers. This greater productivity means we can easily create higher levels of abstractions in shorter amounts of time. We can also use a smaller yet more capable set of developers to create applications. In addition, greater productivity means we can create parts of our application quickly and get feedback from our fellow developers, testers, domain experts, and customer representatives. And all this leads to greater agility. 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 to web development in "Why Scripting Languages Matter" (see Appendix 1, *Web Resources*, on page ?).

Dynamic languages have been around for a long time, so why is now a great time to get excited about them? There are at least four reasons:

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

Let's start by talking about machine speed. Doing at runtime what other languages do at compile time raises a concern about dynamic languages' speed. 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 power and memory today than large computers had decades ago. Tasks that were quite unimaginable using a 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 and JVM support for dynamic languages.

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. The Groovy users mailing list is very active, with constant discussions from passionate users expressing opinions of, ideas about, and criticisms of current and future features.[3] This is leading to greater experimentation, learning, and adaptation of languages than in the past.

Next let's look at 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 our program can be extended at runtime, we can't rely upon coding-time verification alone. From the testing point of view, writing code in dynamic languages requires greater discipline than writing in statically typed languages. Over the past few years, we've seen increased 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.

Finally, 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—compelling stories to share with our developers and managers. That tipping point, for Ruby in particular and for dynamic languages in general, came in the form of Rails.[4] It showed struggling web developers how they could quickly develop applications using Ruby's dynamic capabilities. In the same vein came Grails, a web framework written in Groovy and Java that offers the same productivity and ease.[5]

3. Visit http://groovy.codehaus.org/Mailing+Lists and http://groovy.markmail.org to see.
4. http://rubyonrails.org
5. http://grails.org

These frameworks have caused enough stir in the development community to make the industrywide adoption of dynamic languages highly probable.

Dynamic languages, along with metaprogramming capabilities, make simple things simpler and hard things manageable. We still have to deal with the inherent complexity of our application, but dynamic languages let us focus our 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: use the exceedingly complex and not-so-flexible Java along with heavyweight tools, or move on to using dynamic languages such as Ruby that are object-oriented and have metaprogramming capabilities built in. (For instance, it takes only a couple of lines of code to do aspect-oriented programming—AOP—in Ruby and Groovy.) A few years ago, taking advantage of dynamic capabilities and metaprogramming while being productive meant leaving behind the Java platform. (After all, we use these features to be productive and can't let them slow us down, right?) That is not the case anymore. Languages such as Groovy, JRuby, and Clojure are dynamic and run on the JVM. Using these languages, we can take full advantage of both the rich Java platform and dynamic-language capabilities.

Why Groovy?

As Java programmers, we don't have to switch completely to a different language. Groovy feels like the Java language we already know, with a few augmentations.

Dozens of scripting languages can run on the JVM—Groovy, JRuby, BeanShell, Scheme, Jaskell, Jython, JavaScript, and others. The list could go on and on. Our language choice should depend on a number of criteria: our needs, our preferences, our background, the projects we work with, our corporate technical environment, and so on. In this section, we discuss when Groovy is the *right* language to use.

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.

Let's explore these in detail. First, we can run almost any Java code as Groovy (see Section 2.11, *Gotchas*, on page ? for known problem areas), which means a flat learning curve. We can start writing code in Groovy and, if we're stuck, simply switch gears and write the Java code we're familiar with. We can later refactor that code and make it groovier.

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

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

As we learn Groovy, we 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, we're just getting started, and very soon you'll be a pro at it):

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

Second, when programming in Groovy we can expect almost everything we expect in Java. Groovy classes extend the same good old java.lang.Object—Groovy classes are Java classes. The object-oriented paradigm and Java semantics are preserved, so when we write expressions and statements in Groovy, we already know what those mean to us as Java programmers.

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

Introduction/UseGroovyClass.groovy

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

If we run groovy UseGroovyClass, we'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 we've enjoyed the benefits of other dynamically typed languages, such as Smalltalk, Python, JavaScript, and Ruby, we can also enjoy those in Groovy. For instance, if we want to add the method isPalindrome() to String—a method that tells whether a word is spelled the same forward and backward—we can add that easily with only a couple of lines of code (again, don't try to figure out all the details of how this works right now; we have the rest of the book for that):

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()}"
```

Let's look at the output to see how the previous code works:

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

That's how easy it is to extend a class—even the sacred java.lang.String class—with convenient methods, without intruding into its source code.

Finally, as Java programmers, we rely heavily on the JDK and the API to get our work done. These are available in Groovy. In addition, Groovy extends the JDK with convenience methods and closure support through the Groovy JDK (GDK). Here's a quick example of a GDK extension to the java.util.ArrayList class:

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

From the output of the previous code, we can confirm that the JDK is being used, but in addition we're able to use the Groovy-added join() method to concatenate the elements in the ArrayList:

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

Groovy augments the Java we know. If a project team is familiar with Java, is using it for most of the organization's projects, and has a lot of Java code to integrate and work with, then Groovy is a nice path toward productivity gains.

What's in This Book?

This book is about programming with Groovy; it is aimed at Java programmers who already know the JDK well but are interested in learning the Groovy language and its dynamic capabilities. Throughout this book we'll explore the Groovy language's features with many practical examples. The objective is to make programmers quickly productive with this interesting and powerful language.

The rest of this book is organized into four parts, as follows:

In the chapters in Part I, "Beginning Groovy," we focus on the whys and whats of Groovy—the fundamentals that'll help us get comfortable with general programming in Groovy. This book is for experienced Java programmers, so we won't spend any time with programming basics, like what an if statement is or how to write it. Instead, we directly dive into the similarities of Groovy and Java, and topics that are specific to Groovy.

In Part II, "Using Groovy," we'll see how to use Groovy for everyday coding—working with XML, accessing databases, and working with multiple Java/Groovy classes and scripts—so we can put Groovy to use right away for the day-to-day tasks. We'll also discuss the Groovy extensions and additions to the JDK so we can take advantage of both the power of Groovy and the JDK at the same time.

In Part III, "MOPping Groovy," we dive into Groovy's metaprogramming capabilities. We'll see Groovy really shine in these chapters and you'll learn how to take advantage of its dynamic nature. We'll start with the fundamentals of the metaobject protocol (MOP), cover how to do AOP-like operations in Groovy, and discuss dynamic method/property discovery and dispatching. We will also explore the compile-time metaprogramming capability and see how it can help extend and transform code during the compilation phase.

In the last part, "Using Metaprogramming," we'll apply Groovy metaprogramming right away to create and use builders and domain-specific languages (DSLs). Unit testing is not only necessary in Groovy because of its dynamic nature, but it's also easy to do—we can use Groovy to unit-test Java and Groovy code, as you'll see in this part of the book.

You're reading the introduction now, of course. Here's what's in the rest of the book:

In Chapter 1, *Getting Started*, on page ?, we'll download and install Groovy and take it for a test-drive using groovysh and groovyConsole. We'll also see how to run Groovy without these tools—from the command line and within an integrated development environment.

In Chapter 2, *Groovy for Java Eyes*, on page ?, we'll start with familiar Java code and refactor that to Groovy. After a quick tour of Groovy features that improve our everyday Java coding, we'll talk about Groovy's support for Java 5 features. Groovy follows Java semantics, except in places it does not—we'll also discuss gotchas that'll help avoid surprises.

In Chapter 3, *Dynamic Typing*, on page ?, we'll see how Groovy's typing is similar to and different from Java's typing, what Groovy really does with the type information we provide, and when to take advantage of dynamic typing versus optional typing. We'll also cover how to take advantage of Groovy's dynamic typing, design by capability, and multimethods. For tasks that need better performance than we can get from dynamic typing, we'll see how we can instruct Groovy to statically type parts of code.

In Chapter 4, *Using Closures*, on page ?, 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. Groovy closures go beyond simple lambda expressions; they facilitate trampoline calls and memoization.

In Chapter 5, *Working with Strings*, on page ?, we'll talk about Groovy strings, working with multiline strings, and Groovy's support for regular expressions.

In Chapter 6, *Working with Collections*, on page ?, we'll explore Groovy's support for Java collections—lists and maps. We'll explore various convenience methods on collections, and we'll never again want to use collections the old way.

Groovy embraces and extends the JDK. We'll explore the GDK and see the extensions to `Object` and other Java classes in Chapter 7, *Exploring the GDK*, on page ?.

Groovy has pretty good support for working with XML, including parsing and creating XML documents, as we'll see in Chapter 8, *Working with XML*, on page ?.

Chapter 9, *Working with Databases*, on page ?, presents Groovy's SQL support, which will make our database-related programming easy and fun. In this chapter, we'll cover iterators, data sets, and how to perform regular database operations using simpler syntax and closures. We'll also see how to get data from Microsoft Excel documents.

One of Groovy's key strengths is its integration with Java. In Chapter 10, *Working with Scripts and Classes*, on page ?, we'll investigate ways to closely interact with multiple Groovy scripts, Groovy classes, and Java classes from within our Groovy and Java code.

Metaprogramming is one of the biggest benefits of dynamic languages in general, and Groovy in particular; with this feature we can inspect classes at runtime and dynamically dispatch method calls. We'll explore Groovy's support for metaprogramming in Chapter 11, *Exploring Metaobject Protocol (MOP)*, on

page ?, beginning with the fundamentals of how Groovy handles method calls to Groovy objects and Java objects.

With Groovy we can perform AOP-like method interceptions using GroovyInterceptable and ExpandoMetaClass, as we'll see in Chapter 12, *Intercepting Methods Using MOP*, on page ?.

In Chapter 13, *MOP Method Injection*, on page ?, we'll dive into Groovy metaprogramming capabilities and learn how to inject methods at runtime.

In Chapter 14, *MOP Method Synthesis*, on page ?, we'll go through how to synthesize or generate dynamic methods at runtime.

Chapter 15, *MOPping Up*, on page ?, covers how to synthesize classes dynamically, how to use metaprogramming to delegate method calls, and how to choose between the metaprogramming techniques from the previous three chapters.

Groovy goodness does not end with runtime metaprogramming. Groovy now offers some of the same benefits at compile time, using abstract syntax tree (AST) transformation techniques, as we'll see in Chapter 16, *Applying Compile-Time Metaprogramming*, on page ?.

Groovy builders are specialized classes that help create fluent interfaces for a nested hierarchy. We discuss how to use them and how to create our own builders in Chapter 17, *Groovy Builders*, on page ?.

Unit testing is not a luxury or an "if we have time" practice in Groovy. Groovy's dynamic nature requires unit testing. Fortunately, Groovy facilitates writing tests and creating mock objects, as we'll cover in Chapter 18, *Unit Testing and Mocking*, on page ?. We will play with techniques that will help us use Groovy to unit-test our Java code and our Groovy code.

We can apply Groovy's metaprogramming capabilities to build internal DSLs using the techniques in Chapter 19, *Creating DSLs in Groovy*, on page ?. We'll start with the basics of DSLs, including their characteristics, and quickly jump into building them in Groovy.

Finally, in Appendix 1, *Web Resources*, on page ?, and Appendix 2, *Bibliography*, on page ?, you'll find all the references to web articles and books cited throughout this book.

Changes Since This Book's First Edition

This book's first edition covered Groovy version 1.5. Groovy has come a long way since then. This second edition is up to date with Groovy 2.1. Here's how the updates in this edition will help you:

- You'll learn Groovy 2.x features.
- You'll learn about Groovy code-generation transformations like `@Delegate`, `@Immutable`, and so on.
- You'll learn the benefits of the new Groovy 2.x static type-checking and static compilation facilities.
- You will pick up tips for creating your own extension methods with the new support for extension modules in Groovy 2.x.
- Closures in Groovy are quite exceptional, and you'll learn about their new support for tail-call optimization and memoization.
- You'll learn how to integrate Java and Groovy effectively, pass Groovy closures from Java, and even invoke dynamic Groovy methods from Java.
- You'll find new examples to learn about the enhancements to the metaprogramming API.
- You'll learn how to use `Mixins` and implement some elegant patterns with them.
- In addition to runtime metaprogramming, you can grasp compile-time metaprogramming and abstract syntax tree (AST) transformations.
- You'll see the details for building and reading JSON data.
- Additionally, you'll learn the Groovy syntax that facilitates fluent creation of DSLs.

Who Is This Book For?

This book is for developers working on the Java platform. It is best suited to programmers (and testers) who understand the Java language fairly well. 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. For example, *Effective Java [Blo08]* and *Thinking in Java [Eck06]* are good resources for Java.

Programmers who are somewhat familiar with Groovy can use this book to learn some tips and tricks that they may not have the opportunity to discover

otherwise. Finally, those already familiar with Groovy may find this book useful for training or coaching fellow developers in their organizations.

Online Resources

Web resources referenced throughout the book are collected in Appendix 1, *Web Resources*, on page ?. Here are two that will help you get started:

- The Groovy website for downloading the version of Groovy used in this book: http://groovy.codehaus.org.
- The official homepage for this book at the Pragmatic Bookshelf website: http://www.pragprog.com/titles/vslg2. From there you can download all the example source code for this book. You can also offer feedback by submitting errata entries or posting your comments and questions in the forum for the book.

If you're reading the book in ebook form, you can click on the link above a code listing to view or download the specific example.

Acknowledgments

It's been a real pleasure watching the Groovy ecosystem grow over the past four years. I thank the Groovy committers for creating a language and a set of tools that help programmers to be productive and have fun at the same time.

I'd like to thank everyone who read the first edition of this book. Special thanks to Norbert Beckers, Giacomo Cosenza, Jeremy Flowers, Ioan Le Gué, Fred Janon, Christopher M. Judd, Will Krespan, Jorge Lee, Rick Manocchi, Andy O'Brien, Tim Orr, Enio Pereira, David Potts, Srivaths Sankaran, Justin Spradlin, Fabian Topfstedt, Bryan Young, and Steve Zhang for taking the time to report errors on the book's errata page.

My sincere thanks and appreciation go to the technical reviewers of the second edition of this book. They were kind enough to give their time and attention to read through the concepts, try out the examples, and provide me valuable feedback, corrections, and encouragements along the way. Thank you, Tim Berglund, Mike Brady, Hamlet D'arcy, Scott Davis, Jeff Holland, Michael Kimsal, Scott Leberknight, Joe McTee, Al Scherer, and Eitan Suez.

A few more people deserve to be called out. I thank Guillaume Laforge for his encouragement and for taking the time to write the foreword. Cédric Champeau and Chris Reigrut were generous to quickly read through the beta of the second edition and provide valuable feedback. I am indebted to you; thank

you. I also thank Thilo Maier for reporting errors on the errata page for the second edition.

Special thanks to Brian Hogan, editor for the second edition, for his reviews, comments, suggestions, and encouragement. He provided much-needed guidance throughout the creation of this edition.

Thanks to the entire Pragmatic Programmers team for taking up this edition and for their support throughout the production process.