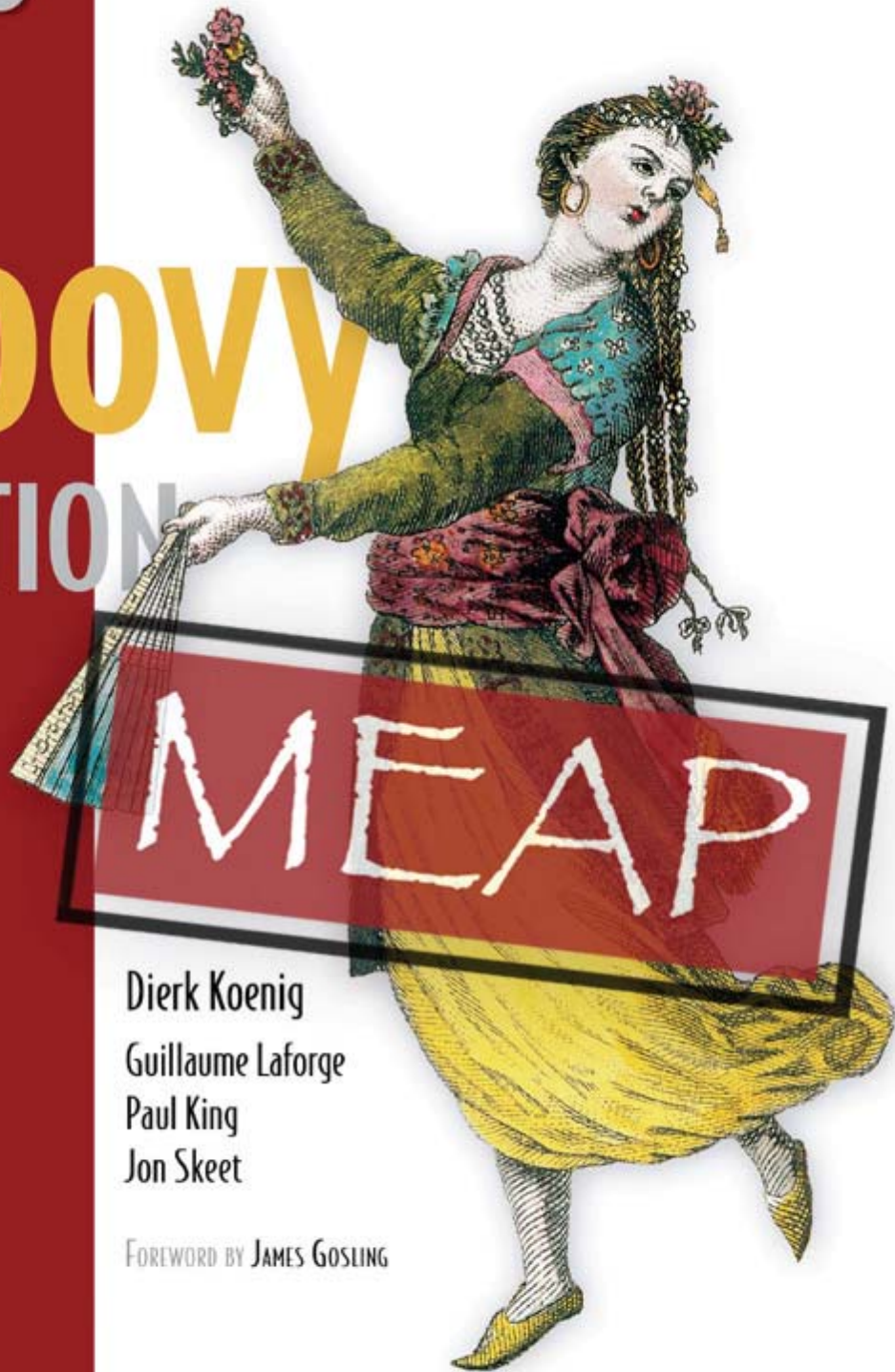


SECOND EDITION

Groovy IN ACTION

COVERS GROOVY 1.7



Dierk Koenig
Guillaume Laforge
Paul King
Jon Skeet

FOREWORD BY JAMES GOSLING

 MANNING



MEAP Edition
Manning Early Access Program

Copyright 2011 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Part I: The Groovy language

- 1 Your way to Groovy
- 2 Overture: The Groovy basics
- 3 The simple Groovy datatypes
- 4 The collective Groovy datatypes
- 5 Working with closures
- 6 Groovy control structures
- 7 Object orientation, Groovy style
- 8 Dynamic programming with Groovy
- 9 Compile-time meta programming and AST Transformations

Part II: Around the Groovy library

- 10 Working with builders
- 11 Working with the GDK
- 12 Database programming with Groovy
- 13 Integrating Groovy
- 14 Working with XML

Part III: Everyday Groovy

- 15 Tips and tricks
- 16 Unit testing with Groovy
- 17 Concurrent Groovy with GPar
- 18 Domain Specific Languages (DSLs)
- 19 Groovy modules and frameworks
- 20 Groovy best practice patterns

Your way to Groovy



A smooth introduction to Groovy

- What Groovy is all about
- How it makes your programming life easier
- How to start

Seek simplicity, and distrust it.

-- Alfred North Whitehead

You've heard of Groovy, maybe even installed the distribution and tried some snippets from the online tutorials. Perhaps your project has adopted Groovy as a dynamic extension to Java and you now seek information about what you can do with it. You may have been acquainted with Groovy from using the Grails web application platform, the Griffon desktop application framework, the Gradle build system or the the Spock testing facility and now look for background information about the language that these tools are built upon. This book delivers to that purpose but you can expect even more from learning Groovy.

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 a pleasure to use.

Learning Groovy is a wise investment. Groovy brings the power of advanced language features such as closures, dynamic methods, 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.

Groovy follows a pragmatic “no drama”¹ approach: it obeys the Java object model and always keeps the perspective of a Java programmer. It doesn't force you into any new programming paradigm but offers you those advanced capabilities that you legitimately expect from a “top of stack” language.

Footnote 1 thanks to Mac Liaw for this wording

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.

He and his wife were waiting for a late plane. While she went shopping, he visited an Internet cafe 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.² 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.

Footnote 2 <http://www.robert-tolksdorf.de/vmlanguages.html> lists about 240 (!) languages targeting the Java Virtual Machine.

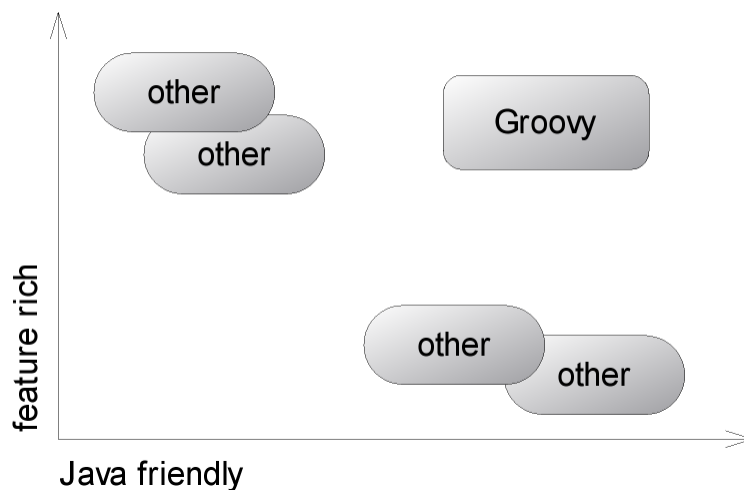


Figure 1.1 The landscape of JVM-based languages. Groovy is feature rich and Java friendly--it excels at both sides instead of sacrificing one for the sake of the other.

In the early days of Groovy, we were mainly asked how it would compare to Java, Beanshell, Pnuts, and embedded expression languages. The focus was clearly on Java-friendliness. Then the focus shifted to dynamic capabilities and the debate went on putting Groovy, JavaScript (Rhino), Jython, and JRuby side by side. Since recently, we see more comparison with JavaFX, Clojure, Scala, Fan, Nice, Newspeak and Jaskell. Most of them introduce the functional programming paradigm to the Java platform, which makes a comparison on the feature dimension rather difficult. They are simply different. Some other JVM languages like Alice and Fortress are even totally unrelated. By the time you read this, some new kids are likely to have appeared on the block and the pendulum may have swung in a totally different direction. But with the landscape picture above you are able to also position upcoming languages.

Some languages may offer more advanced features than Groovy. Not so many

languages may claim to fit equally well to the Java language. None can currently touch Groovy when you consider both aspects together: Nothing provides a better combination of Java friendliness and a complete feature set.

With Groovy being in this position, what are its main characteristics, then?

1.1.1 *What is Groovy?*

Groovy is an optionally typed, 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. Unlike other alternative languages, it is designed as a *companion*, not a replacement for Java.

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, 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 wasn't designed for.

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`. You can call all methods on it that you know are available for a `Date` and you can pass it as an argument to any method that expects a `Date`.

Calling into Java is an easy exercise. It is something that all JVM languages offer--at least the ones worth speaking of. They all make it possible, some by staying inside their own non-Java abstractions, some by providing a gateway. Groovy is one of the few that does it its own way *and* the Java way at the same time, since there is no difference.

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

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

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. This becomes particularly important when integrating with Java frameworks that call your class where you have no control over how that call is effected.

The “interoperability” in this direction is a bit more involved for alternative JVM languages. Yes, they may “compile to bytecode” but that does not mean much for itself, since one can produce valid bytecode that is totally incomprehensible for a Java caller. A language may not even be object-oriented and provide classes and methods. And even if it does, it may assign totally different semantics to those abstractions. Groovy in contrast fully stays inside the Java object model. Actually, compiling to class files is only one of many ways to integrate Groovy into your Java project. The integration chapter describes the full range of options. The integration ladder in figure 1.3 arranges the integration criteria by their significance.

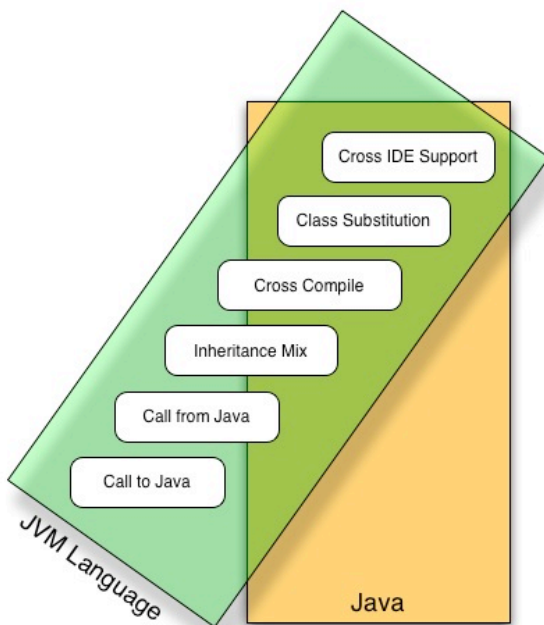


Figure 1.3 The integration ladder shows increasing cross-language support from simple calls for interoperability up to seamless tool integration.

One step up on the integration ladder and we meet the issue of references. A Groovy class may reference a Java class (that goes without saying) and a Java class may reference a Groovy class, as we have seen above. We can even have circular references and `groovyc` compiles them all transparently. Even better, the leading IDEs provide cross-language compile, navigation, and refactoring such that you

hardly ever need to care about the project build setup. You are free to choose Java or Groovy when implementing any class for that matter. Such a tight build-time integration is a challenge for every other language.

Overloaded methods is the next rung where candidates slip off. Imagine you set out to implement the Java interface `java.io.Writer` in any non-Java language. It comes with three versions of “write” that take one parameter: `write(int c)`, `write(String str)`, and `write(char[] buf)`. Implementing this in Groovy is trivial, it's *exactly* like in Java. The formal parameter types distinguish which methods you override. That's one of many merits of optional typing. Languages that are solely dynamically typed have no way of doing this.

But the buck doesn't stop here. The Java/Groovy mix allows annotations and interfaces being defined in either language and implemented and used in the other. You can subclass in any combination even with abstract classes and “sandwich” inheritance like Java - Groovy - Java or Groovy - Java - Groovy in arbitrary depth. It may look exotic at first sight but we actually needed this feature in customer projects. We'll come back to that. Of course, this integration presupposes that your language knows about annotations and interfaces like Groovy does.

True seamless integration means that you can take *any* Java class from a given Java codebase and replace it with a Groovy class. Likewise, you can take any Groovy class and rewrite it in Java both without touching any other class in the codebase. That's what we call a *drop-in replacement*, which imposes further consideration about annotations, static members, and accessibility of the used libraries from Java.

Finally, generated bytecode can be more or less Java-tool-friendly. There are more and more tools on the market that directly augment your bytecode, be it for gathering test coverage information or “weaving aspects” in. These tools do not only expect bytecode to be valid but also to find well-known patterns in it such as the Java and Groovy compiler provide. Bytecode generated by other languages is often not digestable for such tools.

Alternative JVM languages are often attributed as working “seamlessly” with Java. With the integration ladder above, you can check to what degree this applies: calls into Java, calls from Java, bidirectional compilation, inheritance intermix, mutual class substitutability, and tool support. We didn't even consider security, profiling, debugging and other Java “architectures”. So much for the *platform* integration, now onto the syntax.

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 various languages in order to demonstrate what alignment *should* mean:

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

today = new Date()           // Groovy

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

import java.util._           // Scala
var today = new Date         // Scala

(import '(java.util Date)) ; Clojure
(def today (new Date))      ; Clojure
(def today (Date.))        ; Clojure alternative
```

The Groovy solution is short, precise, and more compact than regular Java. Groovy does not need to import the `java.util` package or specify the `Date` type. This is very handy when using Groovy to evaluate user input. In those cases, one cannot assume that the user is proficient in Java package structures or willing to write more code than necessary. Additionally, 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. Scala introduces a new wildcard syntax with underscores and has its own way of declaring whether a reference is supposed to be (in Java terms) “final” or not (`var` vs. `val`). The user has to provide one or the other. Clojure doesn't support wildcard imports as of now and shows two alternative ways of instantiating a Java class, both of which differ syntactically from Java.

Although all the alternative notations make sense in themselves and may even be more consistent than Java, they do not align as nicely with the Java syntax and architecture as Groovy does. Throw into the mix that Groovy is the only language besides Java that fully supports the Java notation of generics and annotations and you easily retrace why we position the Groovy syntax as being perfectly aligned with Java.

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

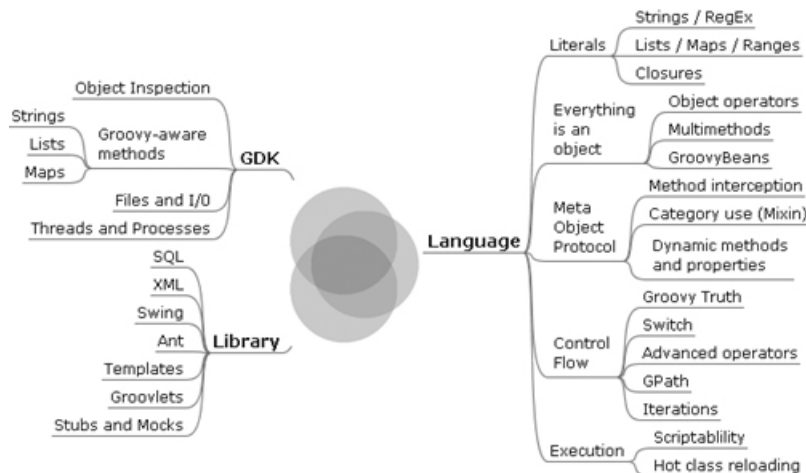


Figure 1.4 Many of the additional libraries and JDK enhancements in Groovy build on the new language features. The combination of the three forms a “sweet spot” for clear and powerful code.

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('data.txt').eachLine { line ->
    number++
    println "$number: $line"
}
```

which prints

```
1: first line
2: second line
```

The curly braces enclose the closure. It is passed as an argument to `File`'s new `eachLine` method which in turn calls back the closure for each line that it reads, passing the current line as an argument.

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 *clazz* is not *class* because that would be a Groovy keyword--exactly like in Java. 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
}
```

which prints

```
java.lang
java.util
java.io
```

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? The star character is optional in the above code. We add it to emphasize that the access to `package` and `name` is *spread* over the list and thus applied to every item in it.

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 had 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.

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

which prints

```
Bill Gates works for Microsoft
Steve Jobs works for Apple
Jonathan Schwartz works for Sun
```

Note that Groovy cannot possibly know anything in advance about the elements and attributes that are available in the XML file. It happily compiles anyway. That's one capability that distinguishes a *dynamic* language.

SCRIPTING THE WEB

For closing up we show a little trick that Scott Davis presented at JavaOne 2009: fetching a rhyme from a REST web service and evaluating the result as if it was Groovy code. This code will print all rhymes to *movie*. Expect your favorite programming language to be included!

```
def text = "http://azarask.in/services/rhyme/?q=movie".toURL().text
for (rhyme in evaluate(text)) println rhyme
```

The term *scripting* refers to the ability to take a string of program code and evaluate it at runtime. That string may be given as user input, read from a database, or fetched from the web like above. The text we fetch happens to be so simple³ that we can treat it as valid Groovy code that denotes a list of Strings. We don't need to write a parser. The Groovy parser does all the work.

Footnote 3 It is actually JavaScript Object Notation (JSON) format.

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 language under standardization 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 has grown beyond all expectations. Recent polls suggest that Groovy is used in the majority of all organizations that develop professionally with Java, much higher than any alternative language. Groovy is regularly covered in Java conferences and publications, and virtually any Java open-source project that allows scripting extensions supports Groovy. Groovy and Grails mailinglists are the most busy ones at codehaus. Groovy has become an important item in many developers CVs and job descriptions.

Many corporations support Groovy in various ways. Sun Microsystems, Inc. integrates Groovy support in their NetBeans IDE tool suite, presents Groovy at JavaOne, and pushes forward the idea of multiple language on the JVM like in the JSRs 241 (Groovy), 223 (Scripting Integration), and 292 (InvokeDynamic). Oracle Corporation has a long-standing tradition of using Groovy in a number of products just like other big players including IBM and SAP. While the development of Groovy has always been driven by its community, it also profited from financial backing. Sustainability of the Groovy development was first sponsored by Big Sky Technology, then by G2One and recently taken over by SpringSource. Big thanks to all that made this development possible!

Commercial support is also available if needed. Many companies offer training, consulting and engineering for Groovy, including the ones that we authors work for (alphabetically): ASERT, Canoo, and SpringSource.

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.

NOTE**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.” (<http://dict.leo.org>)

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:

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

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 that can be maintained while the application is running, 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 ultraslick 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 your *house elf* 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.

NOTE

Example

Recently, Guillaume and I did a spike on *prime number disassembly*.⁴ 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.

Footnote 4 Every ordinal number N can be uniquely disassembled into factors that are prime numbers: $N = p_1 * p_2 * p_3$. The disassembly problem is known to be "hard". Its complexity guards cryptographic algorithms like the popular Rivest-Shamir-Adleman (RSA) algorithm.

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.

TIP

The Groovy Web Console

You can execute Groovy code--and most examples in this book--even without installing anything! Point your browser to <http://groovyconsole.appspot.com/>. This console is hosted on the Google app engine and is thankfully provided by Guillaume Laforge. Share and enjoy!

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.

Table 1.1 Commands to execute Groovy

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

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

1.3.1 Using *groovysh* for a welcome message

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 below where you can enter some Groovy code to receive a warm welcome:

```
Groovy Shell (1.7, JVM: 1.5.0_19)
Type 'help' or 'h' for help.
-----
groovy:000> "Welcome, " + System.properties."user.name"
==> Welcome, Dierk
groovy:000>
```

The shell is a good companion when you work on a remote server with only a text terminal being available. For the more common case that you work on a desktop or laptop machine, there are more comfortable options as we will see in a minute.

The shell can be started with a number of different command-line options that are well explained in the online documentation (<http://groovy.codehaus.org/Groovy+Shell>). It also understands some useful commands, most notably *help*, which spares us listing all commands here. One explanation, though: the shell comes with the notion of an “editing buffer” that comes into play when a statement or expression spans over more multiple lines. Class and method definitions are typical cases. The shell then keeps track of the line numbers and allows various commands on the buffer like editing it in your system's text editor.

1.3.2 Using *groovyConsole*

The *groovyConsole* is a Swing interface that acts as a minimal Groovy development editor. 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.

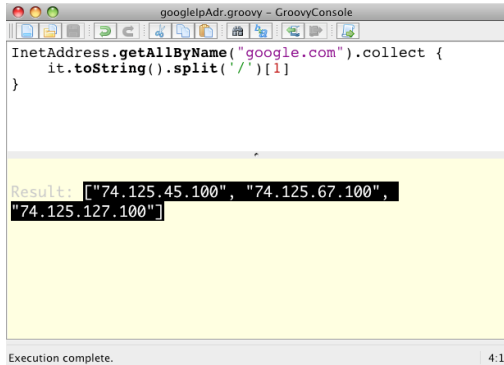


Figure 1.5 The `groovyConsole` with a script in the edit pane that finds the ip addresses of `google.com`. The output pane captures the result.

The `groovyConsole` comes with all the user interface goodness that you can expect from a Swing application.⁵ Walk through the menus or read the documentation under <http://groovy.codehaus.org/Groovy+Console> (you got the pattern by now, right?). The console comes with some pleasant surprises. For good reasons, we made it very “demo friendly”. Ctrl-Shift-L and Ctrl-Shift-S will make the code appear larger or smaller such that the audience can better see the code. You can also drag and drop Groovy files from your filesystem right into the editor. But that's not all!

Footnote 5 Thanks to Romain Guy, the user interface expert and co-author of *Filthy Rich Clients* who supported the Groovy team here.

Figure 1.6 shows the Object Browser inspecting the returned list of ip addresses. It contains information about the `ArrayList` class in the header and tabbed tables showing available variables, methods, and fields.

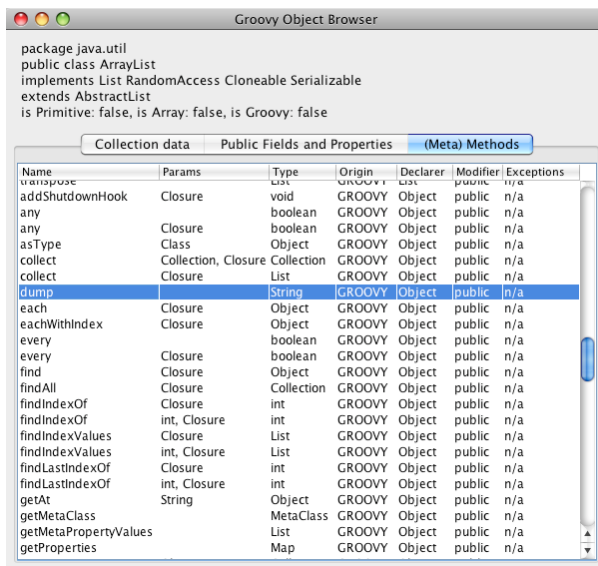


Figure 1.6 The Groovy Object Browser when opened on an object of type `ArrayList`, displaying the table of available methods in its bytecode and registered Meta methods

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.

By this means, you can easily find out, what methods you can call on the object you are currently working on (same intent as code completion in IDEs), which type declared that method and whether it comes from Groovy or Java. Let's try: click on the “Name” header to sort by method names, then on “Declarer”, then on “Origin”. Now scroll down the list until you see “Object” as declarer. Now you should see the same as in Figure 1.6: the list of all methods including parameter types and return type that Groovy adds to `java.lang.Object`. We will learn more about these methods in the GDK chapter 9.

Highlighted is the method `dump()` that Groovy adds to all objects. Try it! Put it in the the input field of the console. You'll see that it is like `toString()` but including the internal state of the object. Very useful, that.

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 calculates the *golden ratio* that intersects a line into a smaller and bigger part such that the total line length relates to the bigger part like the bigger part relates to the smaller one. Composing paintings, photos, or *user interfaces* with the help of the golden ratio is considered pleasing to the human eye and has a long tradition in classic art. The pentagram that underlies the Groovy logo is composed of golden ratios.⁶

Footnote 6 http://en.wikipedia.org/wiki/Golden_ratio#Pentagram

We calculate the golden ratio by narrowing down on the ratio of adjacent Fibonacci⁷ 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. The ratio between `fibonacci(n)` and `fibonacci(n-1)` comes closer and closer to the golden ratio for increasing values of `n`.

Footnote 7 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.

We don't go into the details of the implementation right now. Think about it as arbitrary Groovy code, which for the beginning isn't quite as “Groovy idomatic” as it could be. One little explanation anyway: `[-1]` refers to the last element in a list, `[-2]` to the last-but-one.

If you'd like to try this, copy the code into a file, and save it as `Gold.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 this extension is that you can omit it on the command line when specifying the name of the script--instead of `groovy Gold.groovy`, you can just run `groovy Gold`.

Listing 1.1 `Gold.groovy` calculates the golden ratio by comparing adjacent fibonacci numbers until the golden rule is sufficiently satisfied.

```
List fibo = [1, 1]           // list of fibonacci numbers
List gold = [1, 2]         // list of golden ratio candidates

while ( ! isGolden( gold[-1] ) ) {           // last golden candidate
    fibo.add( fibo[-1] + fibo[-2] )         // next fibo number
    gold.add( fibo[-1] / fibo[-2] )         // next golden candidate
}
```

```

}

println "found golden ratio with fibo(${ fibo.size-1 }) as"
println fibo[-1] + " / " + fibo[-2] + " = " + gold[-1]
println "_" * 10 + " |" + "_" * (10 * gold[-1])

def isGolden(candidate) {      // candidate satisfies golden rule
    def small = 1              // smaller section
    def big = small * candidate // bigger section
    return isCloseEnough( (small+big)/big, big/small)
}

def isCloseEnough(a,b) { return (a-b).abs() < 1.0e-9 }

```

Run this file as a Groovy program by passing the file name to the `groovy` command. You should see the following output that prints the value, the last step of the calculation, and a visual indication of where the golden ratio intersects a given line.

```

found golden ratio with fibo(23) as
46368 / 28657 = 1.6180339882
_____ | _____

```

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 Math.PI"`, which prints `3.141592653589793` 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*⁸ 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.

Footnote 8 We avoid the term “interpreted” to make clear that Groovy code is *never* interpreted in the sense of traditional Perl/Python/Ruby/Bash scripts. It is *always fully compiled* into proper classes--even if that happens transparently.

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 `Gold.groovy` from the previous section into normal Java bytecode by running `groovyc` on the script file like so:

```
groovyc -d classes Gold.groovy
```

In our case, the Groovy compiler outputs a 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.

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.⁹

Footnote 9 The command line as shown applies to Windows shells. The equivalent on Mac/Linux/Solaris/UNIX/Cygwin would be `java -cp $GROOVY_HOME/embeddable/groovy-all-1.7.jar:classes Gold`

```
j a v a - c p
%GROOVY_HOME%/embeddable/groovy-all-1.7.jar;classes
Gold
```

```
found golden ratio with fibo(23) as
46368 / 28657 = 1.6180339882
```

Note that the `.class` file extension for the main class should not be specified

when running with `java`.

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.

Groovy comes with a `groovyc` Ant tasks that works pretty much like the `javac` task. See the details under <http://groovy.codehaus.org/The+groovyc+Ant+Task>. But there is more: the `groovy` Ant task allows you to hook into the Ant build with whatever Groovy code you like. We will come back to this with more details in XREF ant.

When it comes to integrating Groovy into a larger project setup, there are even more options. One is using the Groovy Maven integration. Check out the details under <http://groovy.codehaus.org/GMaven>. A second option is to rely on the Groovy-based Gradle build system that we introduce in XREF gradle. A very lightweight option for dependency resolution is using Groovy's `@Grab` annotation as covered in XREF grape. Finally, Groovy projects of any size are developed with IDE help anyway and they all support transparent cross-compile of Groovy and Java sources as we will see next.

1.5 Groovy IDE and editor support

Depending on how you use Groovy--from command-line scripts through medium sized all-Groovy applications up to multi-language enterprise projects--you face very different needs for development support. On the small scale, a decent text editor is fine, on the large scale, you need the full story including integrated cross-language unit testing, refactoring, debugging and profiling support like all leading IDEs provide. This applies to literally all languages but for Groovy, there is an additional consideration.

The Groovy compiler is very lenient when it comes to compile-time checking of code. It must be, because in a dynamic language, new methods¹⁰ may become available at runtime that the compiler cannot foresee. Therefore, it cannot shield you from mistyped method names. But the IDE can warn you. It can highlight unknown method names and even apply so-called type inference to give even better warnings and type-inferred code completion.

Footnote 10 This applies to more than just method names but we keep it short for the beginning.

That's why IDE support is even more valuable for Groovy as it is for other programming languages. Some commonly used IDEs and text editors for Groovy are listed in the following sections. However, this information is likely to be out of

date as soon as it is printed. Stay tuned for updates for your favorite IDE.

1.5.1 IntelliJ IDEA plug-in

JetBrains, the company behind IntelliJ IDEA, was the first to provide a compelling Groovy plugin for their commercial IDE under the name JetGroovy that today is bundled by default with their distribution (since version 8). Interestingly, this plugin is open-source and you can join the effort. The development of this plugin led to the first cross-language compiler for Groovy that made bidirectional Java-Groovy compilation possible. JetBrains thankfully donated this compiler to the Groovy project and it has heavily influenced the Groovy compiler that we have today.

Listing all the features of JetGroovy would be a silly attempt. I wouldn't even know where to start. It may be enough to say that any Groovy code is so tightly integrated that the lines with Java begin to blur. The screenshot in figure 1.7 shows a Groovy script that produces this book from docbook format to PDF. Note that the method `getRepls()` has no return type and is thus dynamically typed. It returns a map where both keys and values are strings. Now see how in the structure pane (left bottom) the return type is listed as `Map<String, String>`.

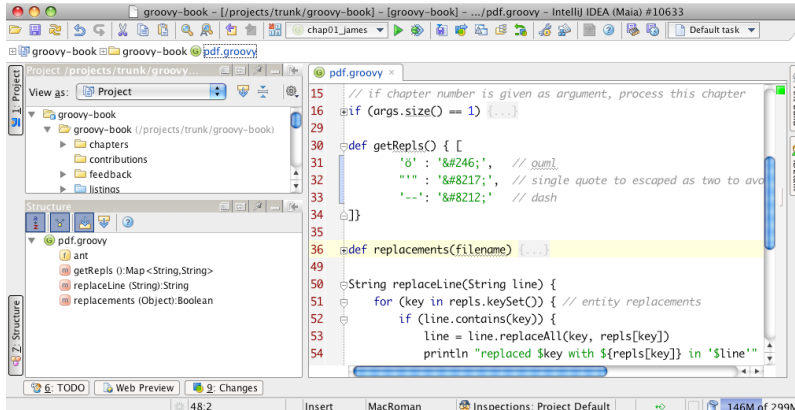


Figure 1.7 The special Groovy support in IntelliJ IDEA uses type inference to provide type safety where the compiler can't.

This is type inference in action and it controls how code completion works in the trailing code and even how method calls on keys and values of that `Map` are known to be of type `String`. As an example, in `line.contains(key)` the `key` must be a `String` and since IntelliJ infers that it is, there is no warning marker.

Note that in contrast the first line shows `args.size()` with `size` underlined. Since the type of `args` is not known, the IDE cannot guarantee that

the `size` method will be available at runtime. It is left to the developer's responsibility.

Beyond the native language support, IntelliJ offers additional goodies for various Groovy-based frameworks like Grails, Griffon, Gant, and by the time you are reading this, probably even more.

1.5.2 NetBeans IDE plug-in

NetBeans IDE, the open-source IDE developed by Sun Microsystems, has recently enjoyed a major uplift in the market. Lots of resources have been granted to the project and Groovy support has become a main focus since version 6.5. Since then, Groovy is part of the standard “Java” distribution of NetBeans IDE.

One of the compelling features of NetBeans IDE is the cross-language support for multiple languages such that one can easily combine Java, Groovy, JavaFx, and others in the same project. Furthermore, NetBeans IDE is always at the forefront of providing value-added services for the Groovy frameworks Grails and Griffon. The online documentation gives a good overview of the features. Also check out Geertjan Wielanga's¹¹ blog and the quick-start guide¹².

Footnote 11 <http://blogs.sun.com/geertjan>

Footnote 12 <http://www.netbeans.org/kb/docs/java/groovy-quickstart.html>

1.5.3 Eclipse plug-in

The Groovy plug-in for Eclipse has a long tradition in which it has gone through a number of changes. Since recently, the effort is led by SpringSource following the approach of coercing the Groovy compiler into contributing to the Java model used by Java Development Toolkit (JDT) to populate the workbench. This is going to result in a fully integrated developer experience for the eclipse user.

More features like advanced Grails support and integration into the SpringSource tool suite (STS) are on the roadmap and likely to be available by the time you read this.

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

1.5.4 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.

The cross-platform *JEdit* editor comes with a plug-in for Groovy that supports

executing Groovy scripts and code snippets. A syntax-highlighting configuration is available separately. More details are available here: <http://groovy.codehaus.org/JEdit+Plugin>.

For Mac users, there is the popular *TextMate* editor with its Windows equivalent simply called *E*. It comes with a Groovy and Grails bundle that you can install from MacroMate's bundle repository.

UltraEdit (Windows only) 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>.

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>.

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 community support and corporate backing, 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.

Part 1

The Groovy language

Learning a new programming language is comparable to learning to speak a foreign language. You have to deal with new vocabulary, grammar, and language idioms. This initial effort pays off multiple times, however. With the new language, you find unique ways to express yourself, you are exposed to new concepts and styles that add to your personal abilities, and you may even explore new perspectives on your world. This is what Groovy did for us, and we hope Groovy will do it for you, too.

The first part of this book introduces you to the language basics: the Groovy syntax, grammar, and typical idioms. We present the language *by example* as opposed to using an academic style.

You may want to skim this part initially and revisit it later when you're getting read to for serious development with Groovy. If you decide to skim, please make sure you visit chapter 2 and its examples. They are cross-linked to the in-depth chapters so you can easily look up details about any topic that interests you.

One of the difficulties of explaining a programming language by example is that you have to start somewhere. No matter where you start, you end up needing to use some concept or feature that you haven't explained yet for your examples. Section 2.3 serves to resolve this perceived deadlock by providing a collection of self-explanatory warm-up examples.

We explain the main portion of the language using its built-in datatypes and introduce expressions, operators, and keywords as we go along. By starting with some of the most familiar aspects of the language and building up your knowledge in stages, we hope you'll always feel confident when exploring new territory.

Chapter 3 introduces Groovy's typing policy and walks through the text and numeric datatypes that Groovy supports at the language level.

Chapter 4 continues looking at Groovy's rich set of built-in types, examining those with a collection-like nature: ranges, lists, and maps.

Chapter 5 builds on the preceding sections and provides an in-depth description of the *closure* concept.

Chapter 6 touches on logical branching, looping, and shortcutting program execution flow.

Finally, chapter 7 sheds light on the way Groovy builds on Java's object-oriented features and takes them to a new level of dynamic execution.

At the end of part 1, you'll have a "big picture" view of the Groovy language. This is the basis for getting the most out of part 2, which explores the Groovy library: the classes and methods that Groovy adds to the Java platform. Part 3, "Everyday Groovy," will apply the knowledge obtained in parts 1 and 2 to the daily tasks of your programming business.