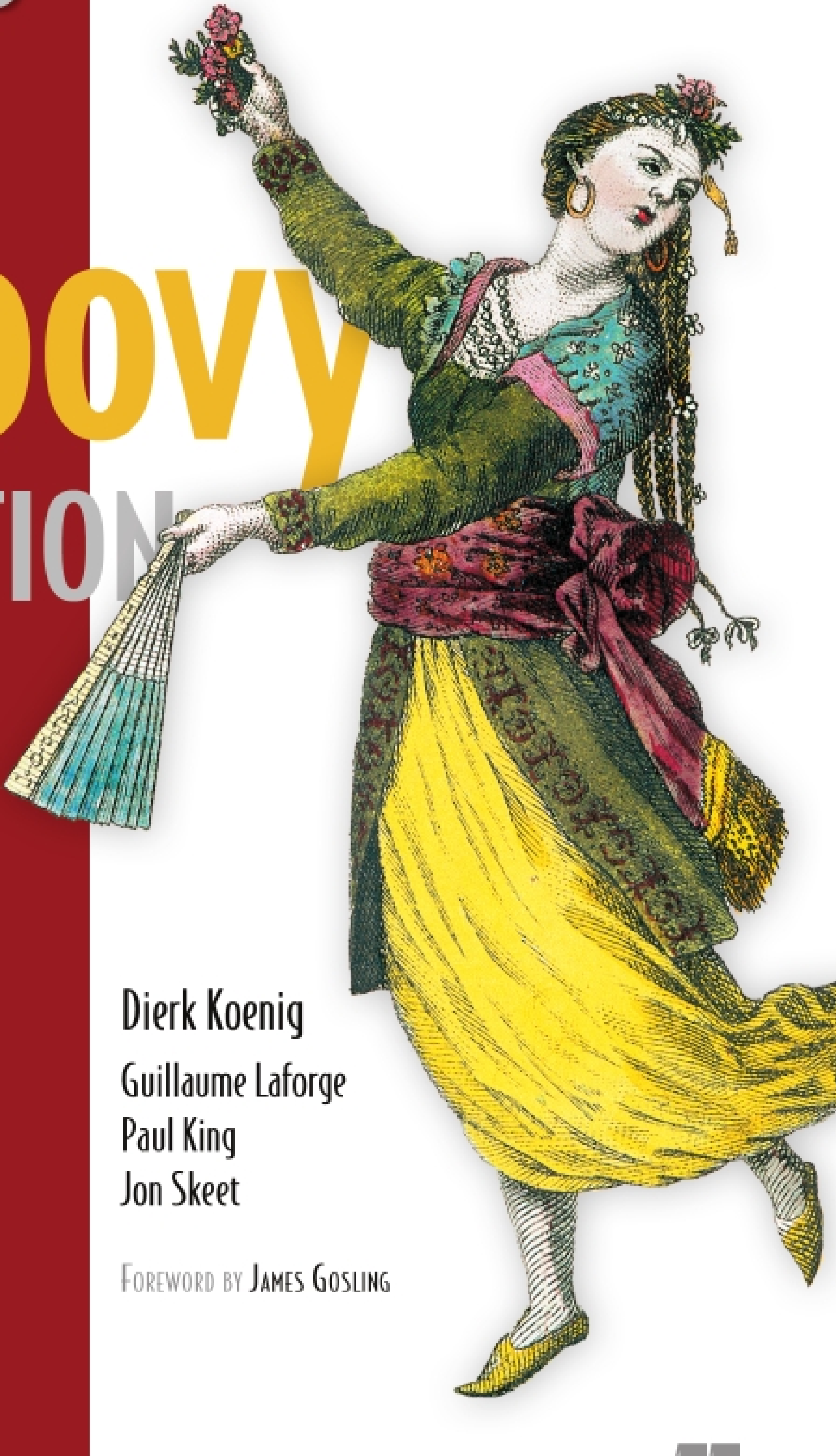
SECOND EDITION

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

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

Table of Contents

1 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”[3] 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 3. 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.[4] 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 4. http://www.robert-tolksdorf.de/vmlanguages.html lists about 240 (!) languages targeting the Java Virtual Machine.

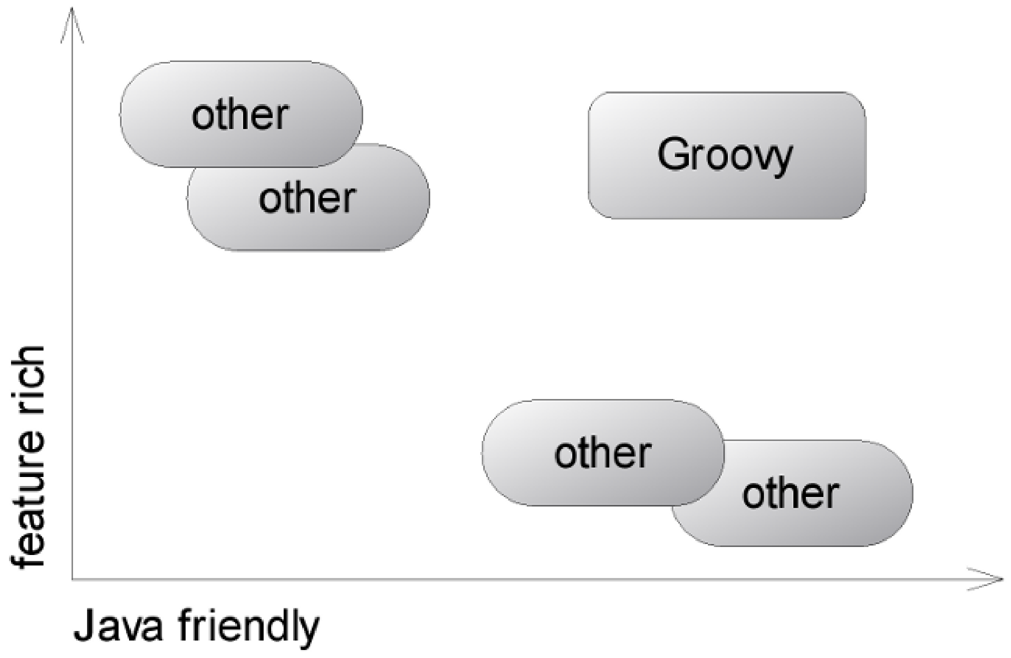


Figure 1.1. The landscape of JVM-based languages. Groovy is feature rich and Java friendly--it excels at both sides instead of sacrificing one for the sake of the other.

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.

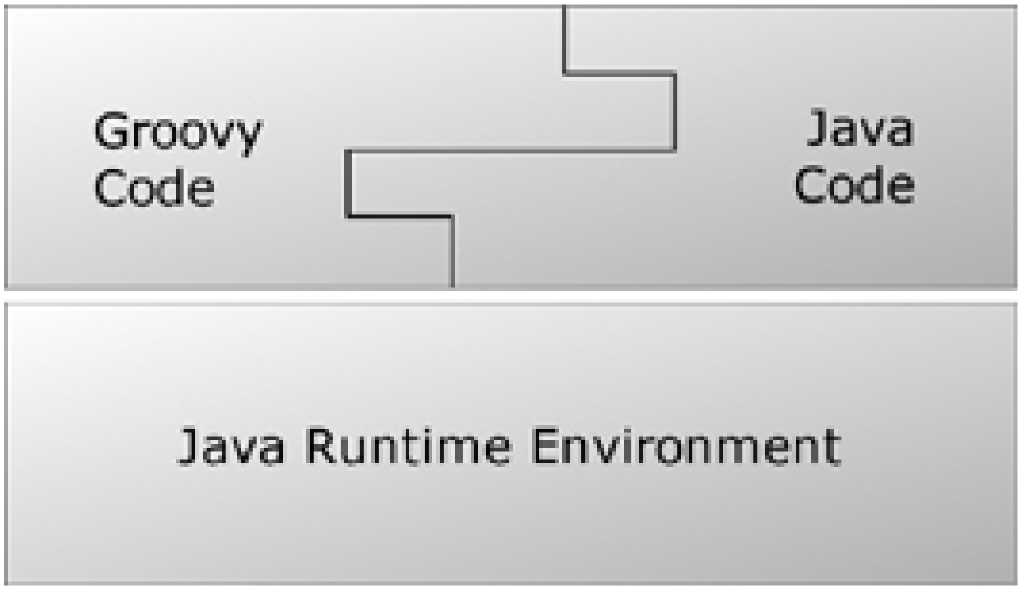


Figure 1.2. Groovy and Java join together in a tongue-and-groove fashion.

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.

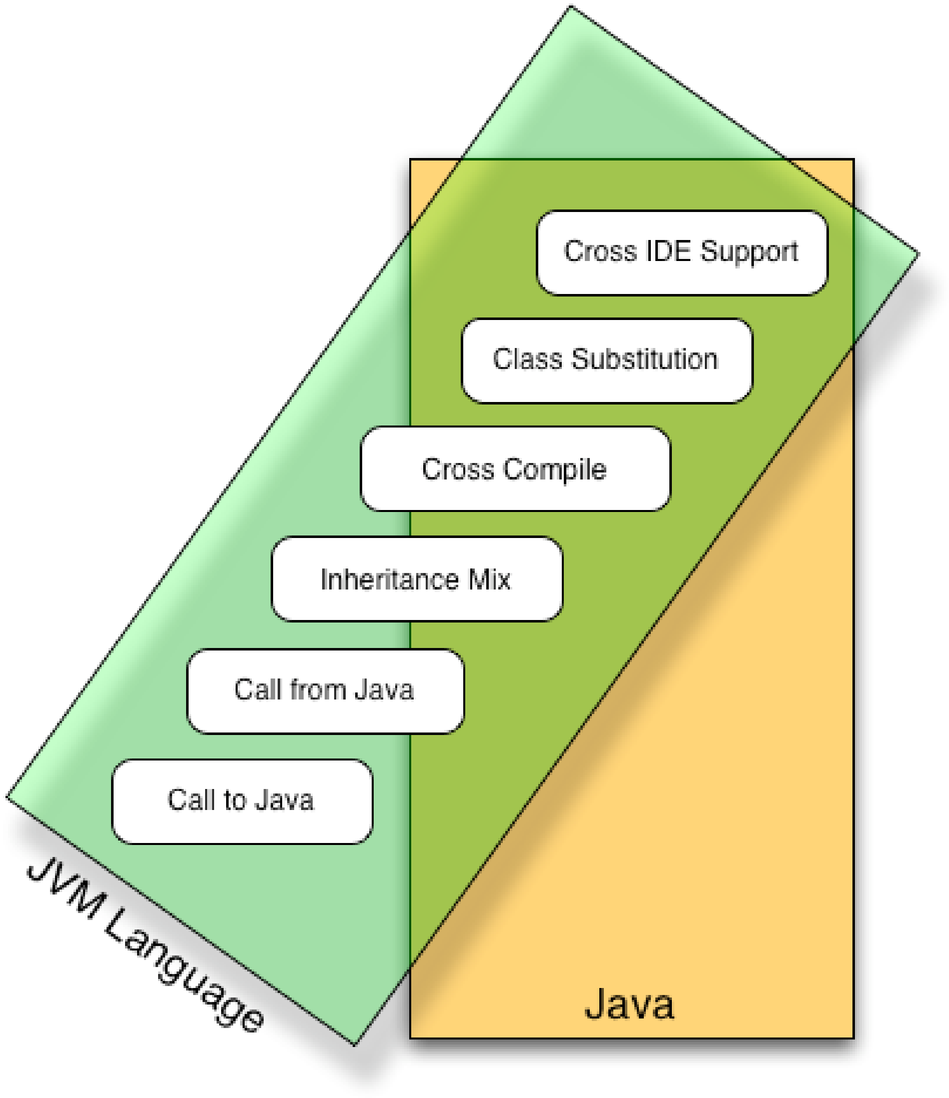


Figure 1.3. The integration ladder shows increasing cross-language support from simple calls for interoperability up to seamless tool integration.

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.

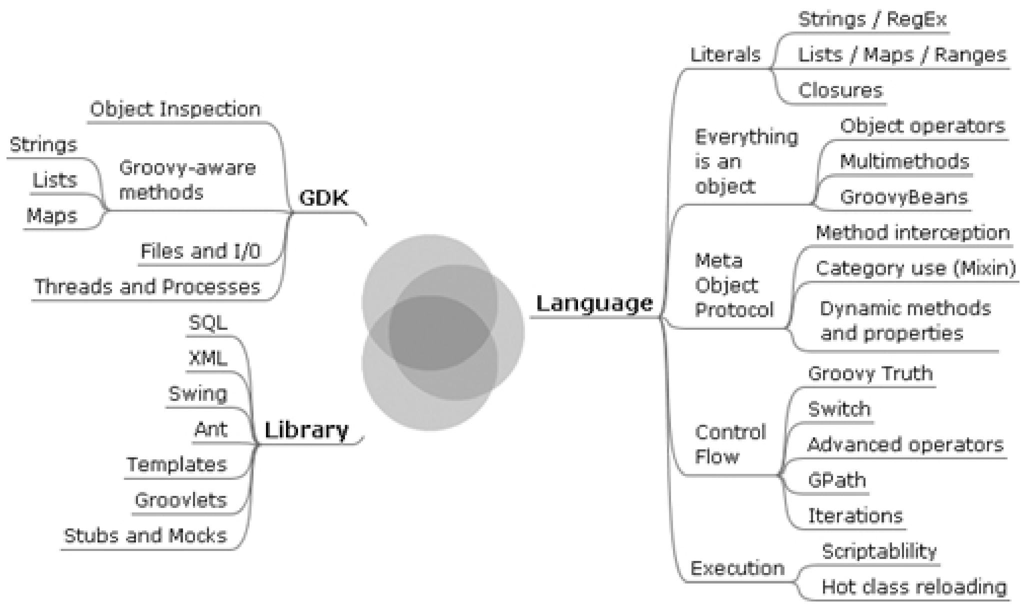


Figure 1.4. Many of the additional libraries and JDK enhancements in Groovy build on the new language features. The combination of the three forms a "sweet spot" for clear and powerful code.

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 fo
```

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 simple[5] that we can treat it is 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 5. 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.

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
            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.

Example

Recently, Guillaume and I did a spike on *prime number disassembly.* [6] 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 6. 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.

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.

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.

Commands to execute Groovy

Command	What it does
`groovy`	Starts the processor that executes Groovy scripts. Single-line Groovy scripts can be specified as command-line arguments.
`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 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.

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>
```

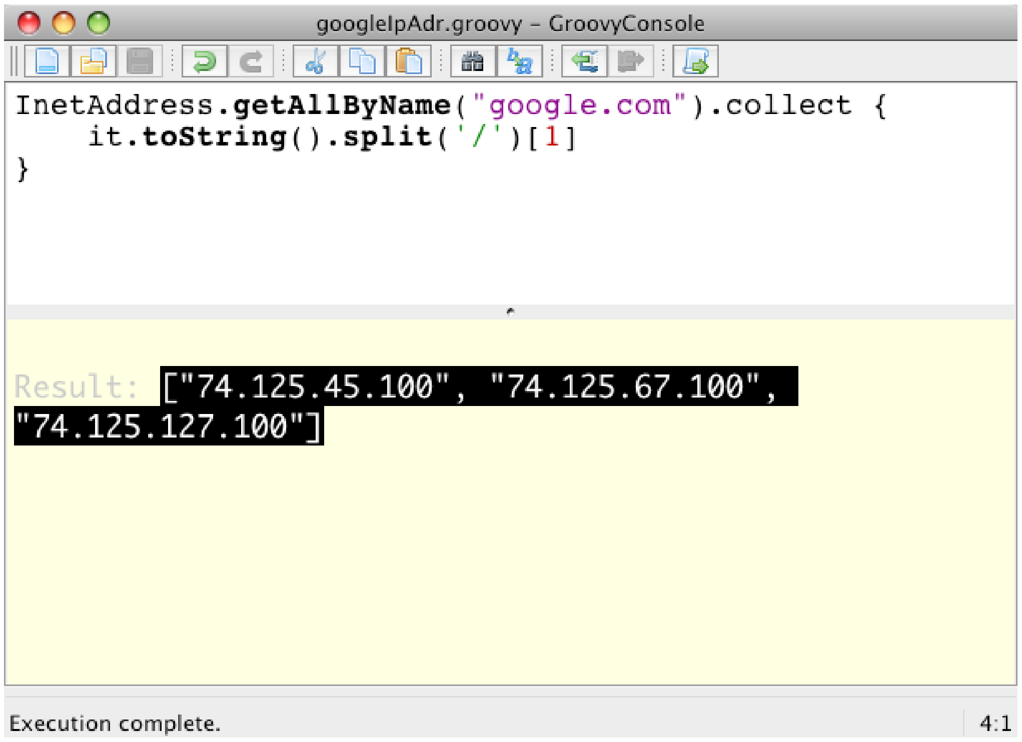
The shell is a good companion when you work on a remote server with only a text terminal being available. For the more common case that you work on a desktop or laptop machine, there are more comfortable options as we will see in a minute.

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.



```
InetAddress.getAllByName("google.com").collect {
    it.toString().split('/')[1]
}
```

```
Result: ["74.125.45.100", "74.125.67.100",
"74.125.127.100"]
```

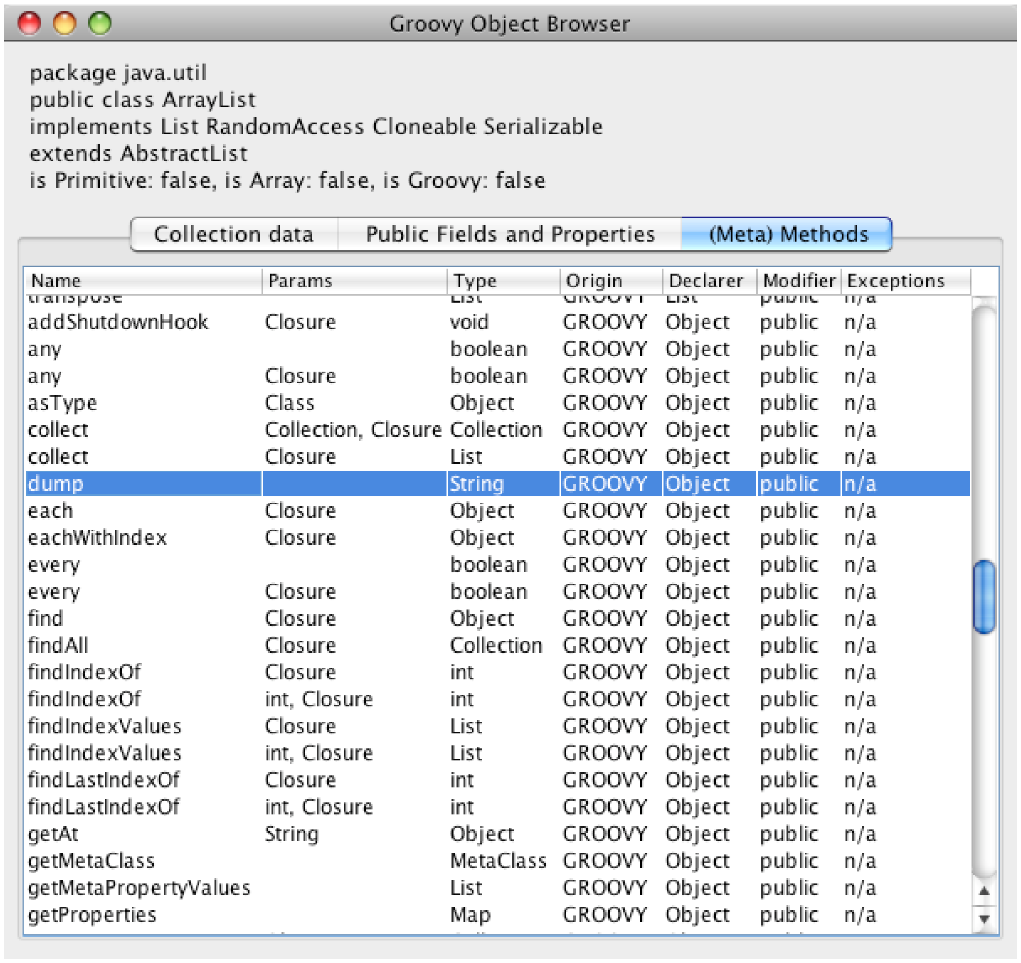
```
Execution complete.                                    4:1
```

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.[7] Walk through the menues 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 7. 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.



package java.util
public class ArrayList
implements List RandomAccess Cloneable Serializable
extends AbstractList
is Primitive: false, is Array: false, is Groovy: false

Name	Params	Type	Origin	Declarer	Modifier	Exceptions
addShutdownHook	Closure	void	GROOVY	Object	public	n/a
any		boolean	GROOVY	Object	public	n/a
any	Closure	boolean	GROOVY	Object	public	n/a
asType	Class	Object	GROOVY	Object	public	n/a
collect	Collection, Closure	Collection	GROOVY	Object	public	n/a
collect	Closure	List	GROOVY	Object	public	n/a
dump		String	GROOVY	Object	public	n/a
each	Closure	Object	GROOVY	Object	public	n/a
eachWithIndex	Closure	Object	GROOVY	Object	public	n/a
every		boolean	GROOVY	Object	public	n/a
every	Closure	boolean	GROOVY	Object	public	n/a
find	Closure	Object	GROOVY	Object	public	n/a
findAll	Closure	Collection	GROOVY	Object	public	n/a
findIndexOf	Closure	int	GROOVY	Object	public	n/a
findIndexOf	int, Closure	int	GROOVY	Object	public	n/a
findIndexValues	Closure	List	GROOVY	Object	public	n/a
findIndexValues	int, Closure	List	GROOVY	Object	public	n/a
findLastIndexOf	Closure	int	GROOVY	Object	public	n/a
findLastIndexOf	int, Closure	int	GROOVY	Object	public	n/a
getAt	String	Object	GROOVY	Object	public	n/a
getMetaClass		MetaClass	GROOVY	Object	public	n/a
getMetaPropertyValues		List	GROOVY	Object	public	n/a
getProperties		Map	GROOVY	Object	public	n/a

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 pentagramm that underlies the Groovy logo is composed of golden ratios.[8]

Footnote 8. http://en.wikipedia.org/wiki/Golden_ratio#Pentagram

We calculate the golden ratio by narrowing down on the ratio of adjacent Fibonacci [9] 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 `fibo(n)` and `fibo(n-1)` comes closer and closer to the golden ratio for increasing values of `n`.

Footnote 9. 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*[10] 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 10. 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. [11]

Footnote 11. 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`

```
java -cp %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 one 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 methods[12] 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 12. This applies to more than just method names but we keep it short for the beginning.

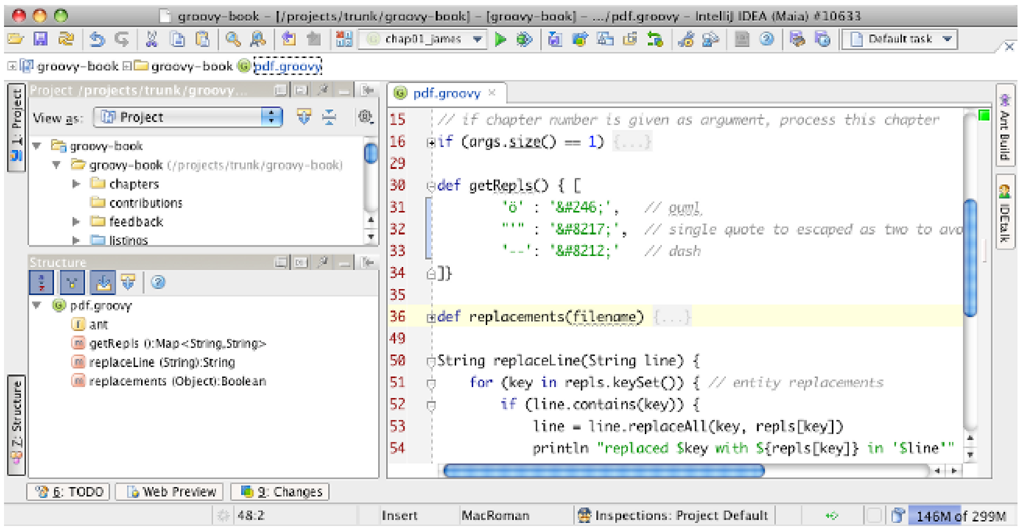
That's why IDE support is even more valuable for Groovy as it is for other programming languages. Some commonly used IDEs and text editors for Groovy are listed in the following sections. However, this information is likely to be out of date as soon as it is printed. Stay tuned for updates for your favorite 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 `getRepl()` 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>`.



groovy-book - [/projects/trunk/groovy-book] - [groovy-book] - .../pdf.groovy - IntelliJ IDEA (Maia) #10633
chap01_james
Default task
groovy-book
groovy-book
pdf.groovy
Project /projects/trunk/groovy...
1: Project
View as:
Project
groovy-book
groovy-book (/projects/trunk/groovy-book)
chapters
contributions
feedback
listings
Structure
7: Structure
pdf.groovy
ant
getRepls ():Map<String,String>
replaceLine (String):String
replacements (Object):Boolean
pdf.groovy
Ant Build
IDEtalk
6: TODO
Web Preview
9: Changes
48:2
Insert
MacRoman
Inspections: Project Default
146M of 299M

```
// if chapter number is given as argument, process this chapter
if (args.size() == 1) {...}

def getRepls() { [
        'ö' : '&#246;',    // ouml
        "'" : '&#8217;',   // single quote to escaped as two to avo
        '--': '&#8212;'    // dash
]}

def replacements(filename) {...}

String replaceLine(String line) {
    for (key in repls.keySet()) { // entity replacements
        if (line.contains(key)) {
            line = line.replaceAll(key, repls[key])
            println "replaced $key with ${repls[key]} in '$line'"
```

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's[13] blog and the quick-start guide[14].

Footnote 13. http://blogs.sun.com/geertjan

Footnote 14. 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.

2 Overture: The Groovy basics

A whistle-stop tour through Groovy

- What Groovy code looks like
- Quickstart examples
- Groovy's dynamic nature

Do what you think is interesting, do something that you think is fun and worthwhile, because otherwise you won't do it well anyway.
-- Brian Kernighan

This chapter follows the model of an overture in classical music, in which the initial movement introduces the audience to a musical topic. Classical composers wove euphonious patterns that were revisited, extended, varied, and combined later in the performance. In a way, overtures are the whole symphony *en miniature*.

In this chapter, we introduce you to many of the basic constructs of the Groovy language. First though, we cover two things you need to know about Groovy to get started: code appearance and assertions. Throughout the chapter, we provide examples to jump-start you with the language, but only a few aspects of each example will be explained in detail—just enough to get you started. If you struggle with any of the examples, revisit them after having read the whole chapter.

An overture allows you to make yourself comfortable with the instruments, the sound, the volume, and the seating. So lean back, relax, and enjoy the Groovy symphony.

2.1. General code appearance

Computer languages tend to have an obvious lineage in terms of their look and feel. For example, a C programmer looking at Java code might not understand a lot of the keywords but would recognize the general layout in terms of braces, operators, parentheses, comments, statement terminators, and the like. Groovy allows you to start out in a way that is almost indistinguishable from Java and transition smoothly into a more lightweight, suggestive, idiomatic style as your knowledge of the language grows. We will look at a few of the basics—how to comment-out code, places where Java and Groovy differ, places where they're similar, and how Groovy code can be briefer because it lets you leave out certain elements of syntax.

First, Groovy is *indentation unaware*, but it is good engineering practice to follow the usual indentation schemes for blocks of code. Groovy is mostly unaware of excessive whitespace, with the exception of line breaks that end the current statement and single-line comments. Let's look at a few aspects of the appearance of Groovy code.

2.1.1. Commenting Groovy code

Single-line comments and multiline comments are exactly like those in Java, with an additional option for the first line of a script:

```
#!/usr/bin/env groovy
// some line comment
/* some multi-
   line comment */
```

Here are some guidelines for writing comments in Groovy:

- The `#!` *shebang* comment is allowed only in the first line. The shebang allows Unix shells to locate the Groovy bootstrap script and run code with it.
- `//` denotes single-line comments that end with the current line.
- Multiline comments are enclosed in `/* ... */` markers.
- Javadoc-like comments in `/** ... */` markers are treated the same as other multiline comments, but are processed by the *groovydoc* Ant task.

Other parts of Groovy syntax are similarly Java-friendly.

2.1.2. Comparing Groovy and Java syntax

Most Groovy code—but not all—appears exactly as it would in Java. This often leads to the false conclusion that Groovy's syntax is a superset of Java's syntax. Despite the similarities, neither language is a superset of the other. For example, Groovy currently doesn't support multiple initialization and iteration statements in the classsic *for(init1,init2;test;inc1,inc2)* loop. As you will see in listing 2.1, the language semantics can be slightly different even when the syntax is valid in both languages. For example, the == operator can give different results depending on which language is being used.

Beside those subtle differences, the overwhelming majority of Java's syntax is *part* of the Groovy syntax. This applies to

- The general packaging mechanism
- Statements (including package and import statements)
- Class, interface, enum, field and method definitions including nested classes; except for special cases with nested class definitions inside methods or other deeply nested blocks
- Control structures
- Operators, expressions, and assignments
- Exception handling
- Declaration of literals; with the exception of literal array initialization where the Java syntax would clash with Groovy's use of curly braces. Groovy uses a shorter bracket notation for declaring lists instead.
- Object instantiation, referencing and dereferencing objects, and calling methods
- Declaration and use of generics and annotations.

The added value of Groovy's syntax is to

- Ease access to Java objects through new expressions and operators
- Allow more ways of creating objects using literals
- Provide new control structures to allow advanced flow control
- Introduce new datatypes together with their operators and expressions
- A \ backslash at the end of a line escapes the line feed such that the statement can proceed on the following line.
- Additional parentheses force Groovy to treat the enclosed content as an expression. We will need this feature in XREF maps.

Overall, Groovy looks like Java with these additions. These additional syntax elements make the code more compact and easier to read. One interesting aspect that Groovy *adds* is the ability to leave things *out*.

2.1.3. Beauty through brevity

Groovy allows you to leave out some elements of syntax that are always required in Java. Omitting these elements often results in code that is shorter and more *expressive*. For example, compare the Java and Groovy code for encoding a string for use in a URL:

Java:

```
java.net.URLEncoder.encode("a b");
```

Groovy:

```
URLEncoder.encode ’a b’
```

By leaving out the package prefix, parentheses, and semicolon, the code boils down to the bare minimum.

The support for optional parentheses is based on the disambiguation and precedence rules as summarized in the Groovy Language Specification (GLS). Although these rules are unambiguous, they are not always intuitive. Omitting parentheses can lead to misunderstandings, even though the compiler is happy with the code. We prefer to include the parentheses for all but the most trivial situations. The compiler does not try to judge your code for readability—you must do this yourself.

Groovy automatically imports the packages `groovy.lang.*`, `groovy.util.*`, `java.lang.*`, `java.util.*`, `java.net.*`, and `java.io.*` as well as the classes `java.math.BigInteger` and `BigDecimal`. As a result, you can refer to the classes in these packages without specifying the package names. We will use this feature throughout the book, and we’ll use fully qualified class names only for disambiguation or for pointing out their origin. Note that Java automatically imports `java.lang.*` but nothing else.

There are other elements of syntax which are optional in Groovy too:

- In chapter 7, we will talk about optional `return` statements.
- Where Java demands *type declarations*, they either become optional in Groovy or can be replaced by `def` to indicate that you don’t care about the type.

- Groovy makes *type casts* optional.

- You don’t need to add the *throws* clause to your method signature when your method potentially throws a checked exception.

This section has given you enough background to make it easier to concentrate on each individual feature in turn. We’re still going through them quickly rather than in great detail, but you should be able to recognize the general look and feel of the code. With that under our belt, we can look at the principal tool we’re going to use to test each new piece of the language: assertions.

2.2. Probing the language with assertions

If you have worked with Java 1.4 or later, you are probably familiar with *assertions*. They test whether everything is right with the world as far as your program is concerned. Usually they live in your code to make sure you don’t have any inconsistencies in your logic, performing tasks such as checking invariants at the beginning and end of a method or ensuring that method arguments are valid. In this book we’ll use them to demonstrate the features of Groovy. Just as in test-driven development, where the tests are regarded as the ultimate demonstration of what a unit of code should do, the assertions in this book demonstrate the results of executing particular pieces of Groovy code. We use assertions to show not only what code can be run, but the result of running the code. This section will prepare you for reading the code examples in the rest of the book, explaining how assertions work in Groovy and how you will use them.

Although assertions may seem like an odd place to start learning a language, they’re our first port of call because you won’t understand any of the examples until you understand assertions. Groovy provides assertions with the `assert` keyword. Listing 2.1 shows what they look like.

Listing 2.1. Using assertions

```
assert(true)
assert 1 == 1
def    x =  1
assert x == 1
def    y =  1 ; assert y == 1
```

Let’s go through the lines one by one.

```
assert(true)
```

This introduces the `assert` keyword and shows that you need to provide an expression that you're asserting will be true.[15]

Footnote 15. Groovy's meaning of *truth* encompasses more than a simple boolean value, as you will see in section XREF groovy_truth.

```
assert 1 == 1
```

This demonstrates that `assert` can take full expressions, not just literals or simple variables. Unsurprisingly, `1` equals `1`. Exactly like Ruby or Scala but unlike Java, the `==` operator denotes *equality*, not *identity*. We left out the parentheses as well, because they are optional for top-level statements.

```
def x = 1
assert x == 1
```

This defines the variable `x`, assigns it the numeric value `1`, and uses it inside the asserted expression. Note that we did not reveal anything about the *type* of `x`. The `def` keyword means dynamically typed.

```
def y = 1 ; assert y == 1
```

This is the typical style we use when asserting the program status for the current line. It uses two statements on the same line, separated by a semicolon. The semicolon is Groovy's statement terminator. As you have seen before, it is optional when the statement ends with the current line.

What happens if an assertion fails? Let's try![16]

Footnote 16. This code is one the few listings that are *not* executed as part of the book production.

```
def a = 5
def b = 9
assert b == a + a    // #1 expected to fail
```

which prints to the console (yes, really!):

```
Caught: Assertion failed:

assert b == a + a       #1 expression is retained
       | |  | | |
       9 |  5 | 5       #2 referenced values
         |    10        #3 sub-expression
         false          #3 values

 at failingAssert.run(failingAssert.groovy:3)
```

Pause for a minute and think about the language features required to provide such a sophisticated error message. We will learn more about this stunning power assert feature in section XREF power_assert

Assertions serve multiple purposes:

- They can be used to reveal the current program state, as we are using them in the examples of this book. The one-line assertion above reveals that the variable `y` now has the value `1`.
- They often make good replacements for line comments, because they reveal assumptions and verify them *at the same time*. The assertion reveals that for the remainder of the code, it is assumed that `y` has the value `1`. Comments may go out of date without anyone noticing—assertions are always checked for correctness. They're like tiny unit tests sitting inside the real code.

Real Life

One real-life example of the value of assertions is in your hands righ screen). This book is constructed in a way that allows us to run the e the assertions it contains. This works as follows: There is a docbook this book that contains no code, but only placeholders that refer to fi code. With the help of a little Groovy script, all the listings are evalu normal production process even begins. For instance, the assertions evaluated and found to be correct during the substitution process. If the process stops with an error message.

The fact that you are reading a production copy of this bo production process was not stopped and all assertions succeeded. T you confidence in the correctness of the Groovy examples we provi edition, we did the same with MS-Word using Scriptom (chapter MS-Word and AntBuilder (chapter 8) to help with the building si before, the features of Groovy work best when they're used together

Most of our examples use assertions—one part of the expression will use the feature being described, and another part will be simple enough to understand on its own. If you have difficulty understanding an example, try breaking it up, thinking about the language feature being discussed and what you would expect the result to be given our description, and then looking at what *we've* said the result will be, as checked at runtime by the assertion. Figure 2.1 breaks up a more complicated assertion into the different parts.

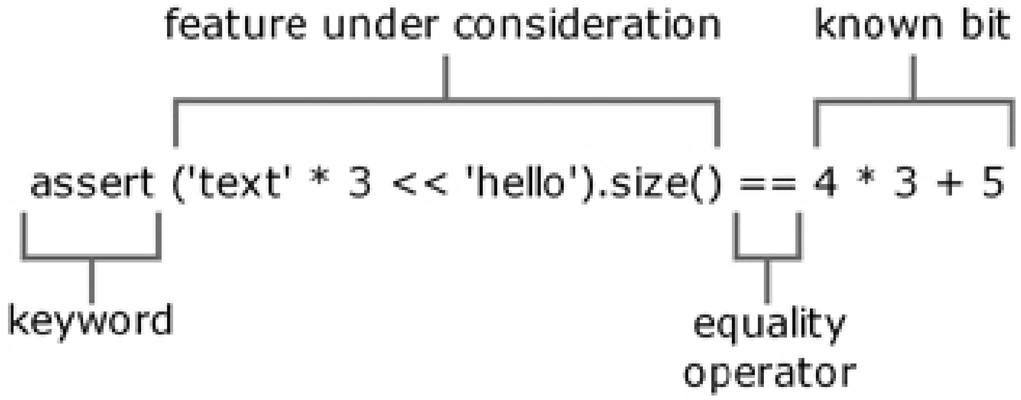

feature under consideration
known bit
assert ('text' * 3 << 'hello').size() == 4 * 3 + 5
keyword
equality operator

Figure 2.1. A complex assertion, broken up into its constituent parts

This is an extreme example—we often perform the steps in separate statements and then make the assertion itself short. The principle is the same, however: There’s code that has functionality we’re trying to demonstrate and there’s code that is trivial and can be easily understood without knowing the details of the topic at hand.

In case assertions do not convince you or you mistrust an asserted expression in this book, you can usually replace it with output to the console. For example, an assertion such as

```
assert x == ’hey, this is really the content of x’
```

can be replaced by

```
println x
```

which prints the value of `x` to the console. Throughout the book, we often replace console output with assertions for the sake of having self-checking code. This is not a common way of presenting code in books, but we feel it keeps the code and the results closer—and it appeals to our test-driven nature.

Assertions have a few more interesting features that can influence your programming style, and we’ll return to them in section 6.2.4 where we’ll cover them in more depth. Now that we have explained the tool we’ll be using to put Groovy under the microscope, you can start seeing some of the real features.

2.3. Groovy at a glance

Like many languages, Groovy has a language specification that breaks down code into statements, expressions, and so on. Learning a language from such a specification tends to be a dry experience and doesn’t take you far towards the goal of writing useful Groovy code in the shortest possible amount of time. Instead, we will present simple examples of typical Groovy constructs that make up most Groovy code: classes, scripts, beans, strings, regular expressions, numbers, lists, maps, ranges, closures, loops, and conditionals.

Take this section as a broad but shallow overview. It won't answer all your questions, but it will allow you to start experimenting with Groovy *on your own*. We encourage you to play with the language—if you wonder what would happen if you were to tweak the code in a certain way, try it! You learn best by experience. We promise to give detailed explanations in later *in-depth* chapters.

2.3.1. Declaring classes

Classes are the cornerstone of object-oriented programming, because they define the blueprints from which objects are created.

Listing 2.2 contains a simple Groovy class named `Book`, which has an instance variable `title`, a constructor that sets the title, and a getter method for the title. Note that everything looks much like Java, except there's no accessibility modifier: Methods are *public* by default.

Listing 2.2. A simple `Book` class

```
class Book {
    private String title
    Book (String theTitle) {
        title = theTitle
    }
    String getTitle(){
        return title
    }
}
```

Please save this code in a file named Book.groovy, because we will refer to it in the next section.

The code is not surprising. Class declarations look much the same in most object-oriented languages. The details and nuts and bolts of class declarations will be explained in chapter 7.

2.3.2. Using scripts

Scripts are text files, typically with an extension of .groovy, that can be executed from the command shell via

```
> groovy myfile.groovy
```

Note that this is very different from Java. In Groovy, we are executing the source code! An ordinary Java class is generated for us and executed behind the scenes. But from a user's perspective, it looks like we are executing plain Groovy source code.[17]

Footnote 17. Any Groovy code can be executed this way as long as it can be *run*; that is, it is either a script, a class with a `main` method, a *Runnable*, or a *GroovyTestCase*.

Scripts contain Groovy statements without an enclosing `class` declaration. Scripts can even contain method definitions outside of class definitions to better structure the code. You will learn more about scripts in chapter 7. Until then, take them for granted.

Listing 2.3 shows how easy it is to use the `Book` class in a script. We create a `new` instance and call the getter method on the object by using Java's *dot*-syntax. Then we define a method to read the title backward.

Listing 2.3. Using the `Book` class from a script

```
Book   gina = new Book('Groovy in Action')

assert gina.getTitle()         == 'Groovy in Action'
assert getTitleBackwards(gina) == 'noitcA ni yvoorG'

String getTitleBackwards(book) {
    String title = book.getTitle()
    return title.reverse()
}
```

Note how we are able to invoke the method `getTitleBackwards` before it is declared. Behind this observation is a fundamental difference between Groovy and scripting languages such as Ruby. A Groovy script is fully constructed—that is, parsed, compiled, and generated—*before execution*. Section 7.2 has more details about this.

Another important observation is that we can use `Book` objects without explicitly compiling the `Book` class! The only prerequisite for using the `Book` class is that Book.groovy must reside on the classpath. The Groovy runtime system will find the file, compile it transparently into a class, and yield a new `Book` object. Groovy combines the ease of scripting with the merits of object orientation.

This inevitably leads to the question of how to organize larger script-based applications. In Groovy, the preferred way is not to mesh numerous script files together, but instead to group reusable components into classes such as `Book`. Remember that such a class remains fully scriptable; you can modify Groovy code, and the changes are instantly available without further action.

It was pretty simple to write the `Book` class and the script that used it. Indeed, it's hard to believe that it can be any simpler—but it *can*, as we'll see next.

2.3.3. GroovyBeans

JavaBeans are ordinary Java classes that expose *properties*. What is a property? That's not easy to explain, because it is not a single standalone concept. It's made up from a naming convention. If a class exposes methods with the naming scheme `getName()` and `setName(name)`, then the concept describes `name` as a property of that class. The `get`- and `set`- methods are called *accessor* methods. (Some people make a distinction between *accessor* and *mutator* methods, but we don't.) Boolean properties can use an `is-` prefix instead of `get-`, leading to method names such as `isAdult`.

A *GroovyBean* is a JavaBean defined in Groovy. In Groovy, working with beans is much easier than in Java. Groovy facilitates working with beans in three ways:

- Generating the accessor methods
- Allowing simplified access to all JavaBeans (including GroovyBeans)
- Simplified registration of event handlers together with annotations that declare a property as *bindable*

Listing 2.4 shows how our `Book` class boils down to a one-liner defining the title property. This results in the accessor methods `getTitle()` and `setTitle(title)` being generated.

Listing 2.4. Defining the `BookBean` class as a GroovyBean

```
class BookBean {
    String title                                // #1 Property declaration
}

def groovyBook = new BookBean()

groovyBook.setTitle('Groovy conquers the world')   // #2 Property use with explicit
assert groovyBook.getTitle() == 'Groovy conquers the world' // #2

groovyBook.title = 'Groovy in Action'           // #3 Property use with Groovy short
assert groovyBook.title == 'Groovy in Action'   // #3
```

We also demonstrate how to access the bean the standard way with accessor methods, as well as the simplified way, where property access reads like direct field access.

Note that listing 2.4 is a fully valid script and can be executed *as is*, even though it contains a class declaration and additional code. You will learn more about this construction in chapter 7.

Also note that `groovyBook.title` is *not* a field access. Instead it is a shortcut for the corresponding accessor method. It would work even if we'd explicitly declared the property "longhand" with a `getTitle` method.

More information about methods and beans will be given in chapter 7.

2.3.4. Annotations for AST Transformations

In Groovy, you can define and use annotations just like in Java, which is a distinctive feature among JVM languages . Beyond that, Groovy also uses annotations to mark code structures for special compiler handling. Let's have a look at one of those annotations that comes with the Groovy distribution: `@Immutable`.

A Groovy bean can be marked as immutable, which means that the class becomes `final`, all its fields become `final`, and you cannot change its state after construction. 2.6 declares an immutable `FixedBean` class, calls the constructor in two different ways, and asserts that we have a standard implementation of `equals()` that supports comparison by content. With the help of a little `try-catch`, we assert that changing the state is not allowed.

Listing 2.5. Defining the immutable `FixedBean` and exercising it

```
@Immutable class FixedBook {                         // #1 AST annotation
    String title
}

def gina   = new FixedBook('Groovy in Action')       // #2 positional ctor
def regina = new FixedBook(title:'Groovy in Action') // #3 named arg ctor

assert gina.title == 'Groovy in Action'
assert gina == regina                          // #4 standard equals()

try {
    gina.title = "Oops!"                       // #5 not allowed!
    assert false, "should not reach here"
} catch (ReadOnlyPropertyException e) {}
```

It must be said that proper immutablity is not easily achieved without such help and the AST transformation does actually much more than what we see above: it adds a correct `hashCode()` implementation and enforces *defensive copying* for access to all properties that aren't immutable by themselves.

Immutable types are always helpful for a clean design but they are indispensable for *concurrent programming*: an increasingly important topic that we will cover in XREF concurrent.

The `@Immutable` AST transformation is only one of many that can enhance your code with additional characteristics. In XREF ast we will cover the full range that comes with the GDK, including `@Bindable`, `@Category`, `@Mixin`, `@Delegate`, `@Lazy`, `@Singleton`, and `@Grab`.

The acronym AST stands for abstract syntax tree, which is a representation of the code that the Groovy parser creates and the Groovy compiler works upon to generate the bytecode. In between, AST transformations can modify that AST to sneak in new method implementations or add, delete, or modify any other code structure. This approach is also called compile-time meta-programming and is not limited to the transformations that come with the GDK. You can also provide your own transformations!

2.3.5. Handling text

Just like in Java, character data is mostly handled using the `java.lang.String` class. However, Groovy provides some tweaks to make that easier, with more options for string literals and some helpful operators.

GStrings

In Groovy, string literals can appear in single or double quotes. The double-quoted version allows the use of placeholders, which are automatically resolved as required. This is a *GString*, and that's also the name of the class involved. The following code demonstrates a simple variable expansion, although that's not all GStrings can do:

```
def nick = 'ReGina'
def book = 'Groovy in Action, 2nd ed.'
assert "$nick is $book" == 'ReGina is Groovy in Action, 2nd ed.'
```

Chapter 3 provides more information about strings, including more options for GStrings, how to escape special characters, how to span string declarations over multiple lines, and the methods and operators available on strings. As you'd expect, GStrings are pretty neat.

Regular expressions

If you are familiar with the concept of *regular expressions*, you will be glad to hear that Groovy supports them *at the language level*. If this concept is new to you, you can safely skip this section for the moment. You will find a full introduction to the topic in chapter 3.

Groovy makes it easy to declare regular expression patterns, and provides operators for applying them. Figure 2.2 declares a pattern with the slashy `//`

syntax and uses the =~ find operator to match the pattern against a given string. The first line ensures that the string contains a series of digits; the second line replaces every digit with an x.

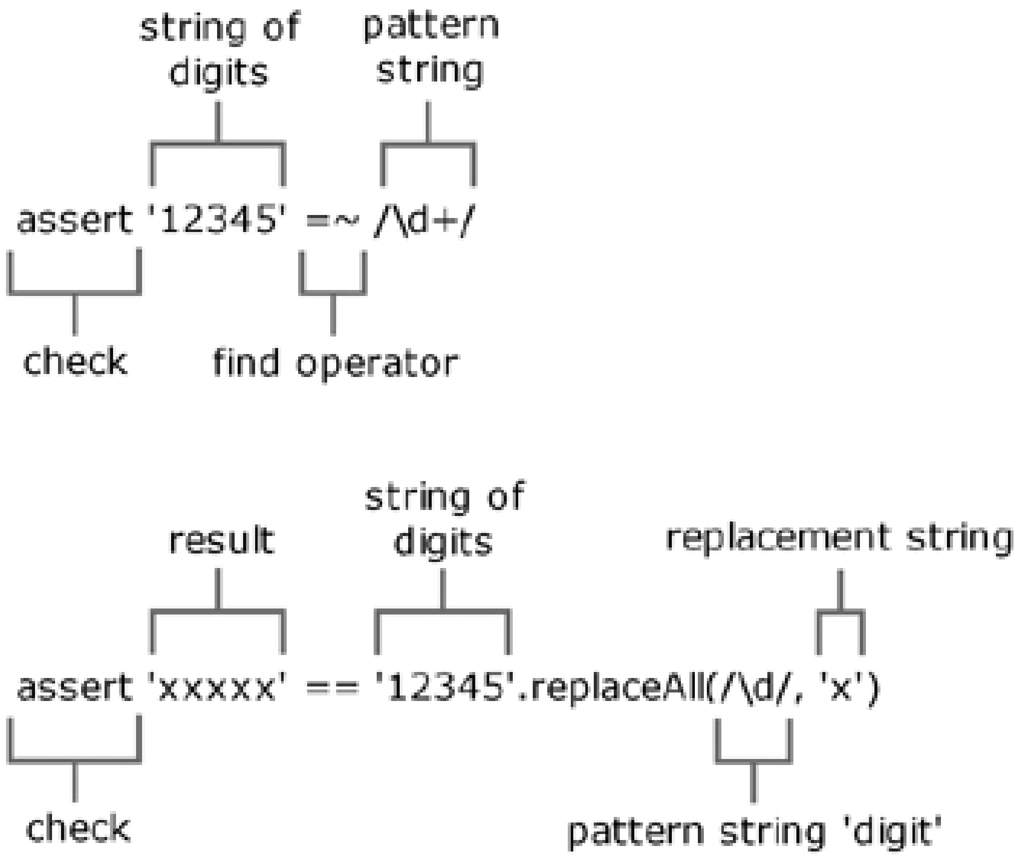


Figure 2.2. Regular expression support in Groovy through operators and slashy strings

Note that `replaceAll` is defined on `java.lang.String` and takes two string arguments. It becomes apparent that `’12345’` is a `java.lang.String`, as is the expression `/\d/`.

Chapter 3 explains how to declare and use regular expressions and goes through the ways to apply them.

2.3.6. Numbers are objects

Hardly any program can do without numbers, whether for calculations or (more frequently) for counting and indexing. Groovy *numbers* have a familiar appearance, but unlike in Java, they are first-class objects rather than primitive types.

In Java, you cannot invoke methods on primitive types. If `x` is of primitive type `int`, you cannot write `x.toString()`. On the other hand, if `y` is an object, you cannot use `2*y`.

In Groovy, both are possible. You can use numbers with numeric operators, and you can also call methods on number instances.

```
def x = 1
def y = 2
assert x + y     == 3
assert x.plus(y) == 3
assert x instanceof Integer
```

The variables `x` and `y` are objects of type `java.lang.Integer`. Thus, we can use the `plus` method. But we can just as easily use the `+` operator.

This is surprising and a major lift to object orientation on the Java platform. Whereas Java has a small but ubiquitous part of the language that isn't object-oriented at all, Groovy makes a point of using objects for everything. You will learn more about how Groovy handles numbers in chapter 3.

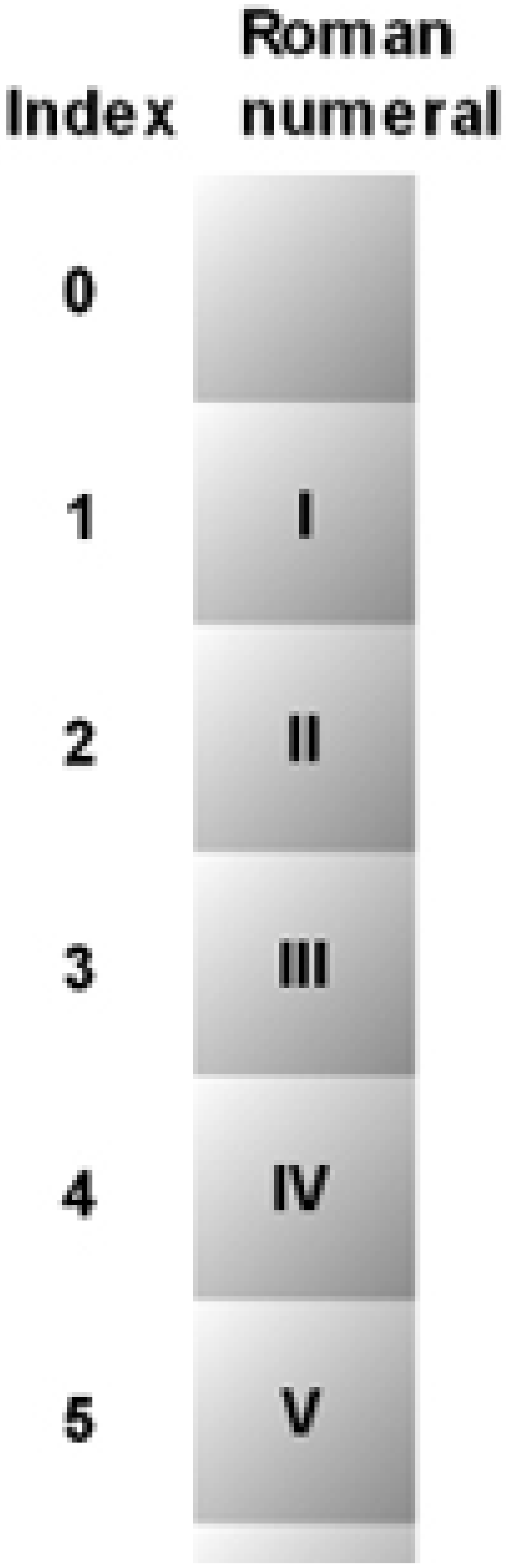
2.3.7. Using lists, maps, and ranges

Many languages, including Java, only have direct support for a single collection type—an array—at the syntax level and have language features that only apply to that type. In practice, other collections are widely used, and there is no reason why the language should make it harder to use those collections than arrays. Groovy makes collection handling simple, with added support for operators, literals, and extra methods beyond those provided by the Java standard libraries.

Lists

Java supports indexing arrays with a square bracket syntax, which we will call the *subscript operator*. Groovy allows the same syntax to be used with *lists*—instances of `java.util.List`—which allows adding and removing elements, changing the size of the list at runtime, and storing items that are not necessarily of a uniform type. In addition, Groovy allows lists to be indexed outside their current bounds, which again can change the size of the list. Furthermore, lists can be specified as literals directly in your code.

The following example declares a list of Roman numerals and initializes it with the first seven numbers, as shown in figure 2.3.


Index
Roman numeral
0
1
I
2
II
3
III
4
IV
5
V

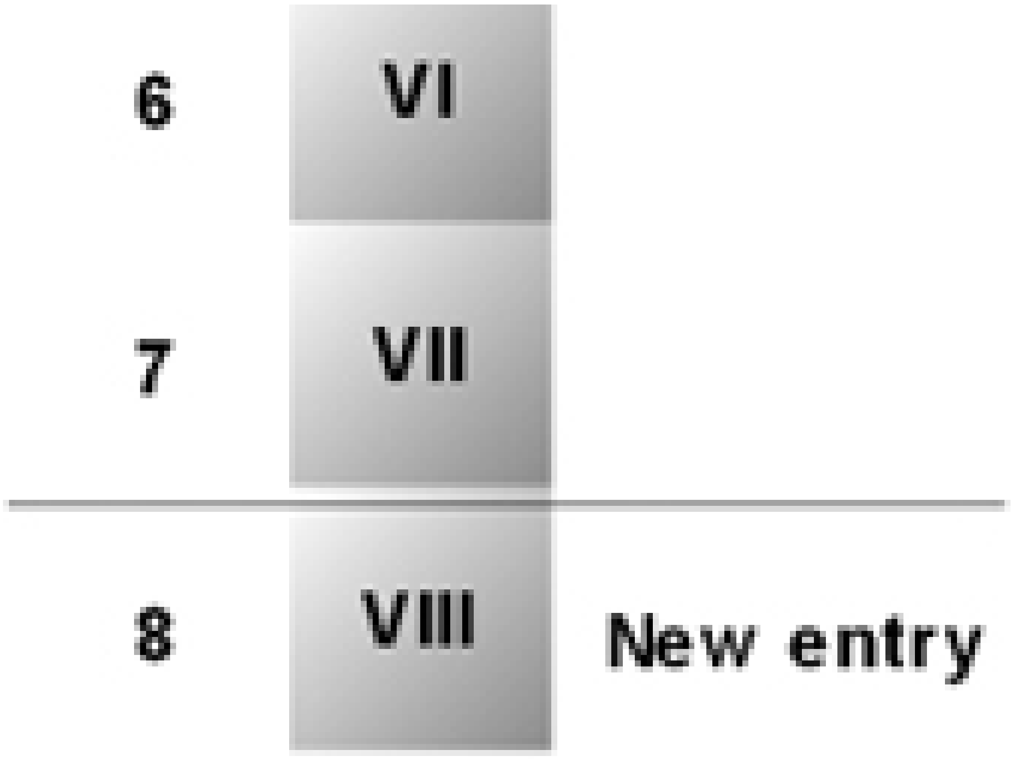


Figure 2.3. An example list where the content for each index is the Roman numeral for that index

The list is constructed such that each index matches its representation as a Roman numeral. Working with the list looks like we're working with an array, but in Groovy, the manipulation is more expressive, and the restrictions that apply to arrays are gone:

```
  def roman = ['', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII
assert roman[4] == 'IV'     // #2 List access

roman[8] = 'VIII'           // #3 List expansion
assert roman.size() == 9
```

Note that there was no list item with index `8` when we assigned a value to it. We indexed the list outside the current bounds. We'll look at the `list` datatype in more detail in section 4.2.

Simple maps

A *map* is a storage type that associates a key with a value. Maps store and retrieve values by key, whereas lists retrieve them by numeric index.

Unlike Java, Groovy supports maps at the language level, allowing them to be specified with literals and providing suitable operators to work with them. It does so with a clear and easy syntax. The syntax for maps looks like an array of key-value pairs, where a colon separates keys and values. That's all it takes.

The following example stores descriptions of HTTP [18] return codes in a map, as depicted in figure 2.4.

Footnote 18. Hypertext Transfer Protocol, the protocol used for the World Wide Web. The server returns these codes with every response. Your browser typically shows the mapped descriptions for codes above 400.

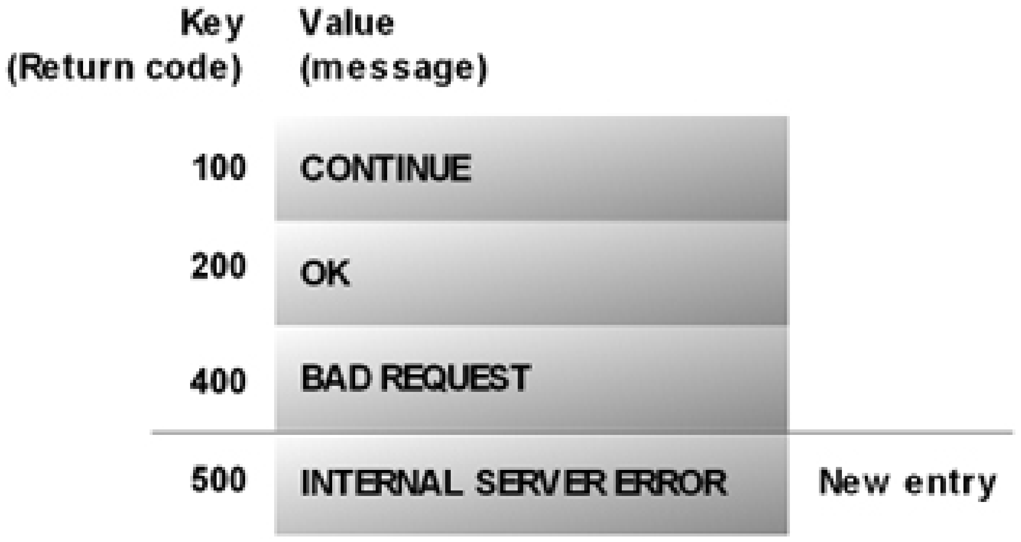


Figure 2.4. An example map where HTTP return codes map to their respective messages

You can see the map declaration and initialization, the retrieval of values, and the addition of a new entry. All of this is done with a single method call explicitly appearing in the source code—and even that is only checking the new size of the map:

```
def http = [
        100 : 'CONTINUE',
        200 : 'OK',
        400 : 'BAD REQUEST'
]
assert http[200] == 'OK'
http[500] = 'INTERNAL SERVER ERROR'
assert http.size() == 4
```

Note how the syntax is consistent with that used to declare, access, and modify lists. The differences between using maps and lists are minimal, so it's easy to remember both. This is a good example of the Groovy language designers taking commonly required operations and making programmers' lives easier by providing a simple and consistent syntax. Section 4.3 gives more information about maps and their rich feature set.

Ranges

Although *ranges* don't appear in the standard Java libraries, most programmers have an intuitive idea of what a range is—effectively a start point and an end point, with an operation to move between the two in discrete steps. Again, Groovy provides literals to support this useful concept, along with other language features such as the `for` statement, which understands ranges.

The following code demonstrates the range literal format, along with how to find the size of a range, determine whether it contains a particular value, find its start and end points, and reverse it:

```
def x = 1..10
assert x.contains(5)
assert x.contains(15) == false
assert x.size() == 10
assert x.from == 1
assert x.to == 10
assert x.reverse() == 10..1
```

These examples are limited because we are only trying to show what ranges do *on their own*. Ranges are usually used in conjunction with other Groovy features. Over the course of this book, you'll see a lot of range usages.

So much for the usual datatypes. We will now come to *closures*, a concept that doesn't exist in Java, but which Groovy uses extensively.

2.3.8. Code as objects: closures

The concept of *closures* is not a new one, but it has usually been associated with functional languages, allowing one piece of code to execute an arbitrary piece of code that has been specified elsewhere.

In object-oriented languages, the Method-Object pattern has often been used to simulate the same kind of behavior by defining types whose sole purpose is to implement an appropriate single-method interface so that instances of those types can be passed as arguments to methods, which then invoke the method on the interface.

A good example is the `java.io.File.list(FilenameFilter)` method. The `FilenameFilter` interface specifies a single method, and its only purpose is to allow the list of files returned from the `list` method to be filtered while it's being generated.

Unfortunately, this approach leads to an unnecessary proliferation of types, and the code involved is often widely separated from the logical point of use. Java uses anonymous inner classes to address these issues, but the syntax is clunky, and there are significant limitations in terms of access to local variables from the calling method. Groovy allows closures to be specified inline in a concise, clean, and powerful way, effectively promoting the Method-Object pattern to a first-class position in the language.

Because closures are a new concept to most Java programmers, it may take a little time to adjust. The good news is that the initial steps of using closures are so easy that you hardly notice what is so new about them. The *aha-wow-cool* effect comes later, when you discover their real power.

Informally, a closure can be recognized as a list of statements within curly braces, like any other code block. It optionally has a list of identifiers in order to name the parameters passed to it, with an `->` arrow marking the end of the list.

It's easiest to understand closures through examples. Figure 2.5 shows a simple closure that is passed to the `List.each` method, called on a list `[1, 2, 3]`.

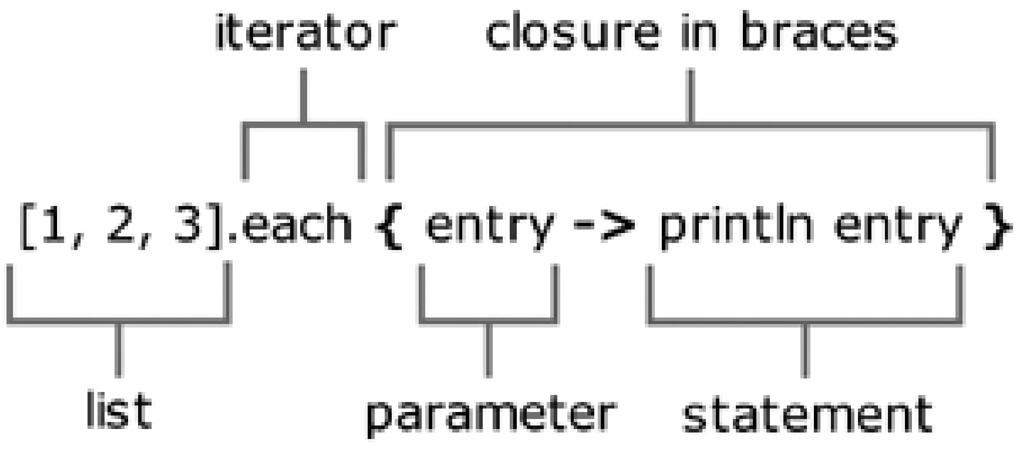


Figure 2.5. A simple example of a closure that prints the numbers 1, 2 and 3

The `List.each` method takes a single parameter—a closure. It then executes that closure for each of the elements in the list, passing in that element as the argument to the closure. In this example, the main body of the closure is a statement to print out whatever is passed to the closure, namely the parameter we've called `entry`.

Let's consider a slightly more complicated question: If *n* people are at a party and everyone clinks glasses with everybody else, how many clinks do you hear?[19] Figure 2.6 sketches this question for five people, where each line represents one clink.

Footnote 19. Or, in computer terms: What is the maximum number of distinct connections in a dense network of *n* components?

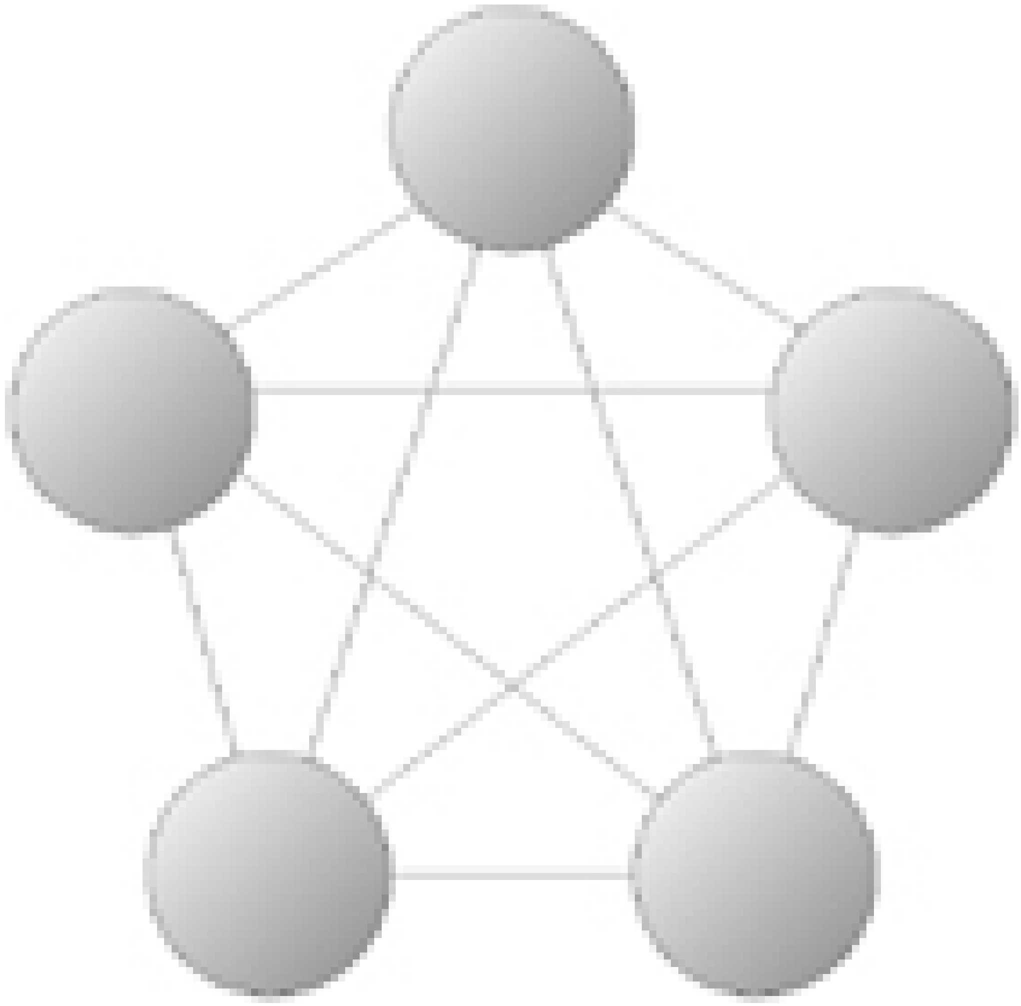


Figure 2.6. Five elements and their distinct connections, modeling five people (the circles) at a party clinking glasses with each other (the lines). Here there are 10 clinks.

To answer this question, we can use `Integer`’s `upto` method, which does *something* for every `Integer` starting at the current value and going *up to* a given end value. We apply this method to the problem by imagining people arriving at the party one by one. As people arrive, they clink glasses with everyone who is already present. This way, everyone clinks glasses with everyone else exactly once.

Listing 2.5 shows the code required to calculate the number of clinks. We keep a running total of the number of clinks, and when each guest arrives, we add the number of people already present (the guest number 1). Finally, we test the result using Gauss’s formula [20] for this problem—with 100 people, there should be 4,950 clinks.

Footnote 20. Johann Carl Friedrich Gauss (1777..1855) was a German mathematician. At the age of seven, when he was a school boy, his teacher wanted to keep the kids busy by making them sum up the numbers from 1 to 100. Gauss discovered this formula and finished the task correctly and surprisingly quickly. There are different reports on how the teacher reacted.

Listing 2.6. Counting all the clinks at a party using a closure

```
def totalClinks = 0
def partyPeople = 100
1.upto(partyPeople) { guestNumber ->
    clinksWithGuest = guestNumber-1
    totalClinks += clinksWithGuest     // #1 modifies outer scope
}
assert totalClinks == (partyPeople * (partyPeople-1)) / 2
```

How does this code relate to Java? In Java, we would have used a loop like the following snippet. The class declaration and main method are omitted for the sake of brevity:

```
// Java snippet
int totalClinks = 0;
int partyPeople = 100;
for(int guestNumber = 1;
        guestNumber <= partyPeople;
        guestNumber++) {
    int clinksWithGuest = guestNumber-1;
    totalClinks += clinksWithGuest;
}
```

Note that `guestNumber` appears four times in the Java code but only twice in the Groovy version. Don't dismiss this as a minor thing. The code should explain the programmer's intention with the simplest possible means, and expressing behavior with two words *rather than four* is an important simplification.

Also note that the `upto` method encapsulates and hides the logic of how to walk over a sequence of integers. That is, this logic appears only *one time* in the code (in the implementation of `upto`). Count the equivalent `for` loops in any Java project, and you'll see the amount of structural duplication inherent in Java.

The example has another subtle twist. The *closure* updates the `totalClinks` variable, which is defined in the outer scope. It can do so because it has access to the *enclosing* scope. That it pretty tricky to do in Java.[21]

Footnote 21. Java pours syntax vinegar over such a construct to discourage programmers from using it.

There is much more to say about the great concept of closures, and we will do so in chapter 5.

2.3.9. Groovy control structures

Control structures allow a programming language to control the flow of execution through code. There are simple versions of everyday control structures like `if-else`, `while`, `switch`, and `try-catch-finally` in Groovy, just like in Java.

In conditionals, *null* is treated like *false*, and so are empty strings, collections, and maps. The *for* loop has a

```
for(i in x) { body }
```

notation, where `x` can be anything that Groovy knows how to iterate through, such as an iterator, an enumeration, a collection, a range, a map—or literally any object, as explained in chapter 6. In Groovy, the *for* loop is often replaced by iteration methods that take a closure argument. Listing 2.6 gives an overview.

Listing 2.7. Control structures

```
if (false) assert false     // #1 'if' as one-liner

if (null)                   // #2 Null is false
{                           // #3 Blocks may start on new line
    assert false
}
else
{
    assert true
}

def i = 0                   // #4 Classic 'while'
while (i < 10) {            // #4
    i++                     // #4
}                           // #4
assert i == 10              // #4

def clinks = 0                    // #5 'for' in 'range'
for (remainingGuests in 0..9) {   // #5
    clinks += remainingGuests     // #5
}                                 // #5
assert clinks == (10*9)/2         // #5

def list = [0, 1, 2, 3]           // #6 'for' in 'list'
for (j in list) {                 // #6
    assert j == list[j]           // #6
}                                 // #6

list.each() { item ->             // #7 'each' method with a closure
    assert item == list[item]     // #7
}                                 // #7

switch(3)  {                       // #8 Classic 'switch'
    case 1 : assert false; break   // #8
    case 3 : assert true;  break   // #8
    default: assert false          // #8
}                                  // #8
```

The code in listing 2.6 should be self-explanatory. Groovy control structures are reasonably close to Java’s syntax, but we’ll go into more detail in chapter 6.

That’s it for the initial syntax presentation. You’ve got your feet wet with Groovy and you should have the impression that it is a nice mix of Java-friendly syntax elements with some new interesting twists.

Now that you know how to write your first Groovy code, it's time to explore how it gets executed on the Java platform.

2.4. Groovy's place in the Java environment

Behind the fun of Groovy looms the world of Java. We will examine how Groovy classes enter the Java environment to start with, how Groovy *augments* the existing Java class library, and finally how Groovy gets its groove: a brief explanation of the dynamic nature of Groovy classes.

2.4.1. My class is your class

Mi casa es su casa. My home is your home. That's the Spanish way of expressing hospitality. Groovy and Java are just as generous with each other's classes.

So far, when talking about Groovy and Java, we have compared the appearance of the source code. But the connection to Java is much stronger. Behind the scenes, all Groovy code runs inside the Java Virtual Machine (*JVM*) and is therefore bound to Java's object model. Regardless of whether you write Groovy classes or scripts, they run as Java classes inside the JVM.

You can run Groovy classes inside the JVM in two ways:

- You can use `groovyc` to compile *.groovy files to Java *.class files, put them on Java's classpath, and retrieve objects from those classes via the Java classloader.
- You can work with *.groovy files directly and retrieve objects from those classes via the Groovy classloader. In this case, no *.class files are generated, but rather *class objects* —that is, instances of `java.lang.Class`. In other words, when your Groovy code contains the expression `new MyClass()`, and there is a MyClass.groovy file, it will be parsed, a class of type `MyClass` will be generated and added to the classloader, and your code will get a new `MyClass` object as if it had been loaded from a *.class file.[22]

Footnote 22. We hope the Groovy programmers will forgive this oversimplification.

These two methods of converting *.groovy files into Java classes are illustrated in figure 2.7. Either way, the resulting classes have the same format as classic Java classes. Groovy enhances Java at the *source code level* but stays compatible at the *bytecode level*.

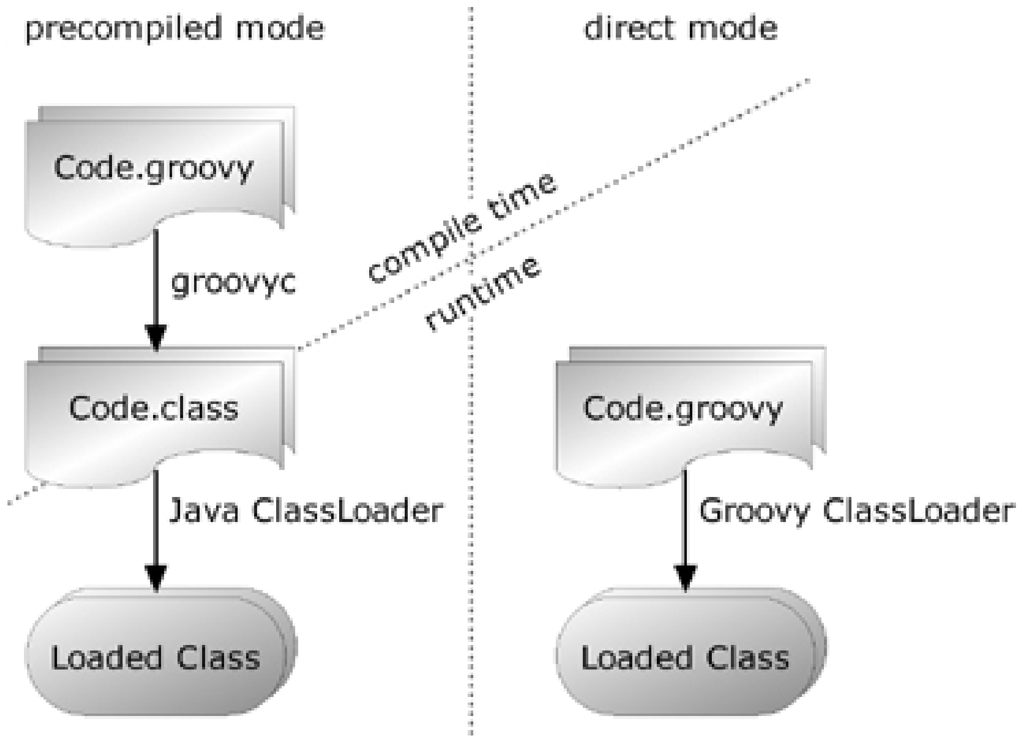


Figure 2.7. Groovy code can be compiled using groovyc and then loaded with the normal Java classloader, or loaded directly with the Groovy classloader

2.4.2. GDK: the Groovy library

Groovy’s strong connection to Java makes using Java classes from Groovy and vice versa exceptionally easy. Because they are both the same thing, there is no gap to bridge. In our code examples, every Groovy object is instantly a Java object. Even the term *Groovy object* is questionable. Both are identical objects, living in the Java runtime.

This has an enormous benefit for Java programmers, who can fully leverage their knowledge of the Java libraries. Consider a sample string in Groovy:

```
’Hello World!’
```

Because this *is* a `java.lang.String`, Java programmers knows that they can use JDK’s `String.startsWith` method on it:

```
if (’Hello World!’.startsWith(’Hello’)) {
    // Code to execute if the string starts with ’Hello’
}
```

The library that comes with Groovy is an extension of the JDK library. It provides some new classes (for example, for easy database access and XML processing), but it also adds functionality to existing JDK classes. This additional functionality is referred to as the GDK[23], and it provides significant benefits in consistency, power, and expressiveness.

Footnote 23. This is a bit of a misnomer because *DK* stands for *development kit*, which is more than just the library; it should also include supportive tools. We will use this acronym anyway, because it is conventional in the Groovy community.

Still have to write Java code? Don't get too comfortable...

Going back to plain Java and the JDK after writing Groovy with the GDK can often be an unpleasant experience! It's all too easy to become accustomed not only to the features of Groovy as a language, but also to the benefits it provides in making common tasks simpler within the standard library.

One example is the `size` method as used in the GDK. It is available on everything that is of some size: strings, arrays, lists, maps, and other collections. Behind the scenes, they are all JDK classes. This is an improvement over the JDK, where you determine an object's size in a number of different ways, as listed in table 2.1.

Various ways of determining sizes in the JDK

Type	Determine the size in JDK via...	Groovy
`Array`	`length` field	`size()` method
`Array`	`java.lang.reflect.Array.getLength(array)`	`size()` method
`String`	`length()` method	`size()` method
`StringBuffer`	`length()` method	`size()` method

`Collection`	`size()` method	`size()` method
`Map`	`size()` method	`size()` method
`File`	`length()` method	`size()` method
`Matcher`	`groupCount()` method	`size()` method

We think you would agree that the GDK solution is more consistent and easier to remember.

Groovy can play this trick by funneling all method calls through a device called `MetaClass`. This allows a dynamic approach to object orientation, only part of which involves adding methods to existing classes. You'll learn more about `MetaClass` in the next section.

When describing the built-in datatypes later in the book, we also mention their most prominent GDK properties. Appendix C contains the complete list.

In order to help you understand how Groovy objects can leverage the power of the GDK, we will next sketch how Groovy objects come into being.

2.4.3. The Groovy lifecycle

Although the Java runtime understands compiled Groovy classes without any problem, it doesn't understand .groovy source files. More work has to happen behind the scenes if you want to load .groovy files dynamically at runtime. Let's dive under the hood to see what's happening.

Some relatively advanced Java knowledge is required to fully appreciate this section. If you don't already know a bit about classloaders, you may want to skip to the chapter summary and assume that magic pixies transform Groovy source code into Java bytecode at the right time. You won't have as full an understanding of what's going on, but you can keep learning Groovy without losing sleep. Alternatively, you can keep reading and not worry when things get tricky.

Groovy *syntax* is line oriented, but the *execution* of Groovy code is not. Unlike other scripting languages, Groovy code is not processed line-by-line in the sense that each line is interpreted separately.

Instead, Groovy code is fully parsed, and a class is generated from the information that the *parser* has built. The generated class is the binding device between Groovy and Java, and Groovy classes are generated such that their format is *identical* to Java bytecode.

Inside the Java runtime, classes are managed by a classloader. When a Java classloader is asked for a certain class, it usually loads the class from a *.class file, stores it in a cache, and returns it. Because a Groovy-generated class is identical to a Java class, it can also be managed by a classloader with the same behavior. The difference is that the Groovy classloader can also load classes from *.groovy files (and do parsing and class generation before putting it in the cache).

Groovy read *.groovy files can *at runtime* as if they were *.class files. The class generation can also be done *before* runtime with the `groovyc` compiler. The compiler simply takes *.groovy files and transforms them into *.class files using the same parsing and class-generation mechanics.

Groovy class generation at work

Suppose we have a Groovy script stored in a file named MyScript.groovy, and we run it via `groovy MyScript.groovy`. The following are the class-generation steps, as shown previously in figure 2.7:

1. The file MyScript.groovy is fed into the Groovy parser.
2. The parser generates an Abstract Syntax Tree (AST) that fully represents all the code in the file.
3. The Groovy class generator takes the AST and generates Java bytecode from it. Depending on the file content, this can result in multiple classes. Classes are now available through the Groovy classloader.
4. The Java runtime is invoked in a manner equivalent to running `java MyScript`.

Figure 2.8 shows a second variant, when `groovyc` is used instead of `groovy`. This time, the classes are written into *.class files. Both variants use the same class-generation mechanism.

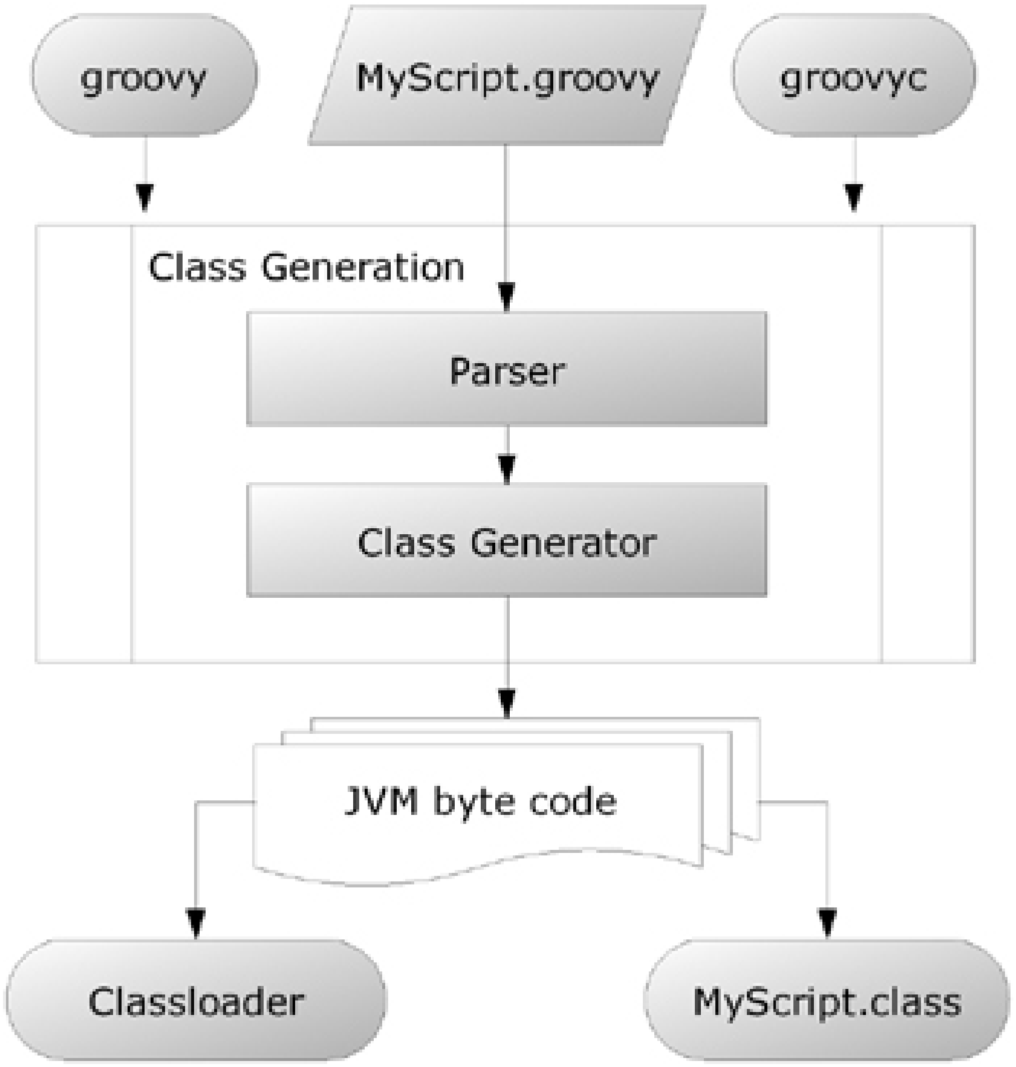


Figure 2.8. Flow chart of the Groovy bytecode generation process when executed in the runtime environment or compiled into class files. Different options for executing Groovy code involve different targets for the bytecode produced, but the parser and class generator are the same in each case.

All this is handled behind the scenes and makes working with Groovy feel like it's an interpreted language, which it isn't. Classes are always fully constructed before runtime and do not change while running. [24]

Footnote 24. This doesn't preclude *replacing* a class at runtime, when the .groovy file changes.

Given this description, you might legitimately ask how Groovy can be called a *dynamic* language if all Groovy code lives in the *static* Java class format. Groovy performs class construction and method invocation in a particularly clever way, as you shall see.

Groovy is dynamic

What makes dynamic languages so powerful is their *dynamic method dispatch*.

Allow yourself some time to let this sink in. It is *not* the dynamic typing that makes a dynamic language dynamic. It is the dynamic method dispatch.

In Grails for example, you see statements like `Album.findByArtist('Oscar Peterson')` but the `Album` class *has no such method*! Neither has any superclass. No class has such a method! The trick is that method calls are funneled through an object called a `MetaClass`, which in our case recognizes that there is no corresponding method in the bytecode of `Album` and therefore relays the call to it's `missingMethod` handler. This knows about the naming convention of Grails' dynamic finder methods and fetches your favourite albums from the database.

But since Groovy is compiled to regular Java bytecode, how is the `MetaClass` called? Well, the bytecode that the Groovy class generator produces is necessarily different from what the Java compiler would generate—not in *format* but in *content*. Suppose a Groovy file contains a statement like `foo()`. Groovy doesn't generate bytecode that reflects this method call directly, but does something like[25]

Footnote 25. The actual implementation involves a few more redirections.

```
getMetaClass().invokeMethod(this, "foo", EMPTY_PARAMS_ARRAY)
```

That way, method calls are redirected through the object's `MetaClass`. This `MetaClass` can now do tricks with method invocations such as intercepting, redirecting, adding/removing methods at runtime, and so on. This principle applies to all calls from Groovy code, regardless of whether the methods are in other Groovy objects or are in Java objects. Remember: There is no difference.

Tip The technically inclined may have fun running `groovyc` on some Groovy code and feeding the resulting class files into a decompiler such as Jad. Doing so gives you the Java code equivalent of the bytecode that Groovy generated.

Calling the `MetaClass` for every method call seems to imply a considerable performance hit, and, yes, this flexibility comes at the expense of runtime performance. However, this hit is not quite as bad as you might expect, since the MetaClass implementation comes with some clever caching and shortcut strategies that allow the Java just-in-time compiler and the hot-spot technology to step in.

A less obvious but perhaps more important consideration is the effect that

Groovy’s dynamic nature has on the compiler. Notice that for example `Album.findByArtist(’Oscar Peterson’)` is not known at compile time but the compiler has to compile it anyway. Now if you have mistyped the method name by accident, the compiler cannot warn you! In fact, the compiler has to accept almost any method call that you throw at him and the code will fail later at runtime.[26] But do not despair! What the compiler cannot do, other tools can. Your IDE can do more than the compiler because it has contextual knowledge of what you are doing. It will warn you on method calls that it cannot resolve and in the case above, it even gives you code completion and refactoring support for Grails’ dynamic finder methods.

Footnote 26. That is, the code fails at unit-test time, right?

A way of using dynamic code is to put the source in a string and ask Groovy to evaluate it. You will see how this works in chapter 11. Such a string can be constructed literally or through any kind of logic. Be warned though: You can easily get overwhelmed by the complexity of dynamic code generation.

Here is an example of concatenating two strings and evaluating the result:

```
  def code = '1 + '
code += System.getProperty('java.class.version')
assert code == '1 + 49.0'
assert 50.0 == evaluate(code)
```

Note that `code` is an ordinary string! It happens to contain `’1 + 49.0’`, which is a valid Groovy expression (a *script*, actually). Instead of having a programmer write this expression (say, `println 1 + 49.0`), the program puts it together at runtime! The `evaluate` method finally executes it.

Wait—didn’t we claim that line-by-line execution isn’t possible, and code has to be fully constructed as a class? How can `code` be *executed* like this? The answer is simple. Remember the left-hand path in figure 2.7? Class generation can transparently happen at runtime. The only new feature here is that the class-generation input can also be a *string* like `code` rather than the content of a *.groovy file.

The ability to evaluate an arbitrary string of code is the distinctive feature of scripting languages. That means Groovy can operate as a scripting language although it is a general-purpose programming language in itself.

2.5. Summary

That's it for our initial overview. Don't worry if you don't feel you've mastered everything we've covered—we'll go over it all in detail in the upcoming chapters.

We started by looking at how this book demonstrates Groovy code using assertions. This allows us to keep the features we're trying to demonstrate and the results of using those features close together within the code. It also lets us automatically verify that our listings are correct.

You got a first impression of Groovy's code notation and found it both similar to and distinct from Java at the same time. Groovy is similar with respect to defining classes, objects, and methods. It uses keywords, braces, brackets, and parentheses in a very similar fashion; however, Groovy's notation is more lightweight. It needs less scaffolding code, fewer declarations, and fewer lines of code to make the compiler happy. This may mean that you need to change the pace at which you read code: Groovy code says more in fewer lines, so you typically have to read more slowly, at least to start with.

Groovy is bytecode compatible with Java and obeys Java's protocol of full class construction before execution. But Groovy is still fully dynamic, generating classes transparently at runtime when needed. Despite the fixed set of methods in the bytecode of a class, Groovy can modify the set of available methods as visible from a Groovy caller's perspective by routing method calls through the `MetaClass`, which we will cover in depth in chapter 7. Groovy uses this mechanism to enhance existing JDK classes with new capabilities, together named GDK.

You now have the means to write your first Groovy scripts. Do it! Grab the Groovy shell (`groovysh`) or the console (`groovyConsole`), and write your own code. As a side effect, you have also acquired the knowledge to get the most out of the examples that follow in the upcoming in-depth chapters.

For the remainder of part 1, we will leave the surface and dive into the deep sea of Groovy. This may be unfamiliar, but don't worry. We'll return to the sea level often enough to take some deep breaths of Groovy code *in action*.

3 The simple Groovy datatypes

Understanding the Groovy type system of “optional typing” and the simple Groovy datatypes.

- Groovy's approach to typing
- Operators as method implementations
- Strings, regular expressions, and numbers

Do not worry about your difficulties in Mathematics. I can assure you mine are still greater.

-- Albert Einstein

Groovy supports a limited set of datatypes at the *language* level; that is, it offers constructs for literal declarations and specialized operators. This set contains the simple datatypes for strings, regular expressions, and numbers, as well as the collective datatypes for ranges, lists, and maps. This chapter covers the simple datatypes; the next chapter introduces the collective datatypes.

Before we go into details, we'll talk about about Groovy's general approach to typing. With this in mind, you can appreciate Groovy's approach of treating everything as an object and all operators as method calls. You will see how this improves the level of object orientation in the language compared to Java's division between primitive types and reference types.

We then describe the natively supported datatypes individually. By the end of this chapter, you will be able to confidently work with Groovy's simple datatypes and have a whole new understanding of what happens when you write `1+1`.

3.1. Objects, objects everywhere

In Groovy, everything is an object. It is, after all, an object-oriented language. Groovy doesn't have the slight “fudge factor” of Java, which is object-oriented apart from some built-in types. In order to explain the choices made by Groovy's designers, we'll first go over some basics of Java's type system. We will then explain how Groovy addresses the difficulties presented, and finally examine how Groovy and Java can still interoperate with ease due to automatic boxing and unboxing where necessary.

3.1.1. Java's type system--primitives and references

Java distinguishes between *primitive* types (such as `boolean`, `short`, `int`, `float`, `double`, `char`, and `byte`) and *reference* types (such as `Object` and `String`). There is a fixed set of *primitive* types, and these are the only types that have *value semantics*--where the value of a variable of that type is the actual number (or character, or true/false value). You cannot create your own value types in Java.

Reference types (everything apart from primitives) have *reference semantics* --the value of a variable of that type is only a *reference* to an object. Readers with a C/C++ background may wish to think of a reference as a pointer--it's a similar concept. If you change the value of a reference type variable, that has no effect on the object it was previously referring to--you're just making the variable refer to a different object, or to no object at all. The reverse is true too: changing the *contents* of an object doesn't affect the value of a variable referring to that object.

You cannot call methods on values of primitive types, and you cannot use them where Java expects objects of type `java.lang.Object`. For each primitive type, Java has a *wrapper type*--a reference type that stores a value of the primitive type in an object. For example, the wrapper for `int` is `java.lang.Integer`.

On the other hand, operators such as `*` in `3*2` or `a*b` are *not* supported for arbitrary[27] reference types, but only for primitive types (with the notable exception of `+`, which is also supported for strings).

Footnote 27. From Java 5 onwards, the autoboxing feature may kick in to unbox the wrapper object to its primitive payload and apply the operator.

The Groovy code in 3.9 calls methods on seemingly primitive types (first with a literal declaration and then on a variable), which is not allowed in Java where you need to explicitly create the integer wrapper to convince the compiler. While calling + on strings is allowed in Java, calling the - (minus) operator is not. Groovy allows both.

Listing 3.1. Groovy allows methods to be called on types that are declared like primitive types in Java even when declared literally. Unlike Java, it also supports operators on reference types.

```
(60 * 60 * 24 * 365).toString();         // invalid Java

int secondsPerYear = 60 * 60 * 24 * 365;
secondsPerYear.toString();               // invalid Java

new Integer(secondsPerYear).toString();

assert "abc" - "a" == "bc"               // invalid Java
```

The Groovy way looks more consistent and involves some language sophistication that we are going to explore next.

3.1.2. Groovy's answer--everything's an object

In order to make Groovy fully object-oriented, and because at the JVM level Java does not support object-oriented operations such as method calls on primitive types, the Groovy designers decided to do away with primitive types. When Groovy needs to store values that would have used Java's primitive types, Groovy uses the wrapper classes already provided by the Java platform. Table 3.1 provides a complete list of these wrappers.

Java's primitive datatypes and their wrappers

Primitive type	Wrapper type	Description
byte	`java.lang.Byte`	8-bit signed integer
short	`java.lang.Short`	16-bit signed integer
int	`java.lang.Integer`	32-bit signed integer
long	`java.lang.Long`	64-bit signed integer

float	`java.lang.Float`	Single-precision (32-bit) floating-point value
double	`java.lang.Double`	Double-precision (64-bit) floating-point value
char	`java.lang.Character`	16-bit Unicode character
boolean	`java.lang.Boolean`	Boolean value (true or false)

Any time you see what looks like a primitive literal value (for example, the number `5`, or the Boolean value `true`) in Groovy source code, that is a reference to an instance of the appropriate wrapper class. For the sake of brevity and familiarity, Groovy allows you to declare variables as if they were primitive type variables. Don't be fooled--the type used is really the wrapper type. Strings and arrays are not listed in table 3.1 because they are already *reference* types, not primitive types--no wrapper is needed.

While we have the Java primitives under the microscope, so to speak, it's worth examining the numeric literal formats that Java and Groovy each use. They are slightly different because Groovy allows instances of `java.math.BigDecimal` and `java.math.BigInteger` to be specified using literals in addition to the usual binary floating-point types. Table 3.2 gives examples of each of the literal formats available for numeric types in Groovy.

Numeric literals in Groovy

Type	Example literals
`java.lang.Integer`	`15, 0x1234ffff`
`java.lang.Long`	`100L, 200l` [28]

java.lang.Float	1.23f, 4.56F
java.lang.Double	1.23d, 4.56D
java.math.BigInteger	123g, 456G
java.math.BigDecimal	1.23, 4.56, 1.4E4, 2.8e4, 1.23g, 1.23G

Notice how Groovy decides whether to use a `BigInteger` or a `BigDecimal` to hold a literal with a “G” suffix depending on the presence or absence of a decimal point. Furthermore, notice how `BigDecimal` is the default type of non-integer literals-- `BigDecimal` will be used unless you specify a suffix to force the literal to be a `Float` or a `Double`.

3.1.3. Interoperating with Java--automatic boxing and unboxing

Converting a primitive value into an instance of a wrapper type is called *boxing* in Java and other languages that support the same notion. The reverse action--taking an instance of a wrapper and retrieving the primitive value--is called *unboxing*. Groovy performs these operations automatically for you where necessary. This is primarily the case when you call a Java method from Groovy. This automatic boxing and unboxing is known as *autoboxing*.

You've already seen that Groovy is designed to work well with Java, so what happens when a Java method takes primitive parameters or returns a primitive return type? How can you call that method from Groovy? Consider the existing method in the `java.lang.String` class: `int indexOf (int ch)`.

You can call this method from Groovy like this:

```
assert 'ABCDE'.indexOf(67) == 2
```

From Groovy's point of view, we're passing an `Integer` containing the value `67` (the Unicode value for the letter *C*), even though the method expects a parameter of primitive type `int`. Groovy takes care of the unboxing. The method returns a primitive type `int` that is boxed into an `Integer` as soon as it enters the world of Groovy. That way, we can compare it to the `Integer` with value `2` back in the Groovy script.

Figure 3.1 shows the process of going from the Groovy world to the Java world and back.

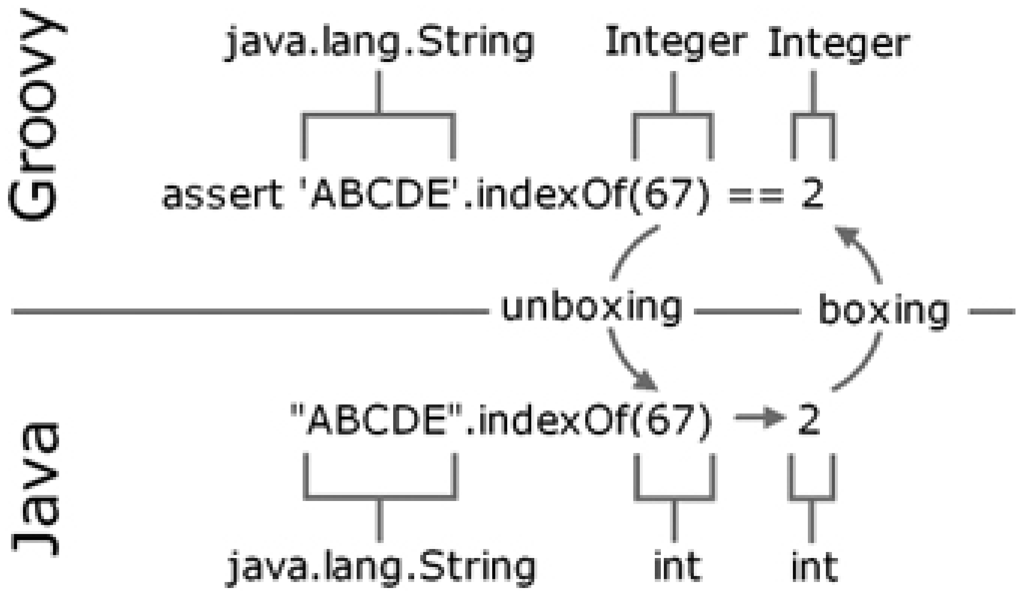


Figure 3.1. Autoboxing in action: An Integer parameter is unboxed to an int for the Java method call, and an int return value is boxed into an Integer for use in Groovy.

All of this is transparent--you don't need to do anything in the Groovy code to enable it. Now that you understand autoboxing, the question of how to apply operators to objects becomes interesting. We'll explore this question next.

3.1.4. No intermediate unboxing

If in `1+1` both numbers are objects of type `Integer`, you may be wondering whether those `Integers` unboxed to execute the *plus* operation on primitive types.

THe answer is no: Groovy is more object-oriented than Java. It executes this expression as `1.plus(1)`, calling the `plus()` method of the first `Integer` object, and passing[29] the second `Integer` object as an argument. The method call returns an `Integer` object of value `2`.

Footnote 29. The phrase “passing an object” is short for “passing a reference to an object”. In Groovy and Java alike, only references are passed as arguments: objects themselves are never passed.

This is a powerful model. Calling methods on objects is what object-oriented languages should do. It opens the door for applying the full range of object-oriented capabilities to those operators.

Let's summarize. No matter how literals (numbers, strings, and so forth) appear in Groovy code, they are always objects. Only at the border to Java are they boxed and unboxed. Operators are a shorthand for method calls. Now that you have seen how Groovy handles types when you tell it what to expect, let's examine what it does when you don't give it any type information.

3.2. The concept of optional typing

So far, we haven't used any type declarations in our sample Groovy scripts--or have we? Well, we haven't used them in the way that you're familiar with in Java. We assigned strings and numbers to variables and didn't care about the type. Behind the scenes, Groovy implicitly assumes these variables to be of static type `java.lang.Object`. This section discusses what happens when a type *is* specified, and the pros and cons of doing it either way.

3.2.1. Assigning types

Groovy offers the choice of explicitly specifying variable types just as you do in Java. Table 3.3 gives examples of optional type declarations It's tricky - anything talking about a "type declaration" makes me think it's a type being declared, not a variable. I guess what we're really talking about is "variable declarations using optional typing" but that's a mouthful. I'm normally an absolute stickler for getting terminology right, but if you'd like to fudge this slightly for the sake of more readable text, that's fine. and the type used at runtime. The `def` keyword is used to indicate that no particular type is specified.

Example Groovy statements and the resulting runtime type

Statement	Type of value	Comment
`def a = 1`	`java.lang.Integer`	Implicit typing
`def b = 1.0f`	`java.lang.Float`	
`int c = 1`	`java.lang.Integer`	Explicit typing using the Java primitive type names

`float d = 1`	`java.lang.Float`	
`Integer e = 1`	`java.lang.Integer`	Explicit typing using reference type names
`String f = '1'`	`java.lang.String`	

As we stated earlier, it doesn't matter whether you declare a variable to be of type `int` or `Integer`. Groovy uses the reference type (`Integer`) either way.

It is important to understand that regardless of whether a variable's type is explicitly declared, the system is *type safe*. Unlike untyped languages, Groovy doesn't allow you to treat an object of one type as an instance of a different type without a well-defined conversion being available. For instance, you could never assign a `java.util.Date` to a reference of type `java.lang.Number`, in the hope that you'd end up with an object that you could use for calculation. That sort of behavior would be dangerous--which is why Groovy doesn't allow it any more than Java does.

3.2.2. Groovy is type-safe at runtime

The Web is full of heated discussions of whether static or dynamic typing is “better” while it often remains unclear what either should actually mean. The word “static” is usually associated with the appearance of type markers in the code, that is while

```
String greeting = readFromConsole()
```

is considered *static* because of the `String` type marker, unmarked code like

```
def greeting = readFromConsole()
```

is often considered *dynamic*. In the latter, the type of `greeting` is whatever the method call returns at runtime. Surely the type of "greeting" is really just "Object" or possibly *no* type (depending on whether this is meant to be a Groovy example or not). It's worth differentiating between the type of the variable and the type of the value it happens to be initially assigned with. Unfortunately, while I can critique this, it's harder to really suggest a fix... it would probably require rewriting the whole paragraph to avoid ending up as a clunky mixture of our voices. Thoughts? And since in a dynamic language like Groovy, it is not foreseeable at compile time what type the `readFromConsole()` method will eventually return[30], there is no point in doing any compile time checks. What we know, though, is that the return type will be assignable to `java.lang.Object`, which becomes our compile-time type in this scenario.

Footnote 30. It may for example be intercepted, relayed or replaced by a different method.

Since type markers (and also type casts) are optional in Groovy, that concept is called *optional typing*.

The above may sound as if type markers were superfluous, but they play an important role at runtime--for the method dispatch as we will see in XREF method_dispatch but also for our current concern: type-safe assignments.

Groovy uses type markers to enforce the Java type system at runtime. Yes, you have read this correctly: *Groovy enforces the Java type system!* But it only does so at runtime, where Java does so with a mixture of compile time and runtime checks. Java enforces the type system to a large extend at compile time based on static information, which gives “static typing” its second meaning. The fact that Java does part of the work at runtime can easily be inferred from the fact that Java programs can still raise `ClassCastException`s and other runtime typing errors.

All this explains why the Groovy compiler[31] takes no issue with

Footnote 31. Your IDE will present you a big warning, though. It can apply additional logic like *dataflow analysis* and *type inference* to even discover more hidden assignment errors. It is your responsibility as a developer how to deal with these warnings.

```
Integer myInt = new Object()
println myInt
```

But when running the code, the cast from `Object` to `Integer` is enforced and you will see

```
org.codehaus.groovy.runtime.typehandling.GroovyCastException:
    Cannot cast object 'java.lang.Object@5b0bc6'
    with class 'java.lang.Object' to class 'java.lang.Integer'
```

In fact, this is the exact same effect you see if you write a typecast on the right-hand-side of the assignment in Java. Consider this Java code:

```
Integer myInt = (Integer) returnsObject(); // Java!
```

The Java compiler will check whether `returnsObject()` returns an object of a type that can sensibly be cast to `Integer`. Let's assume that the declared return type is `Object`. That makes `Object` the *compile-time type*[32] of the `returnsObject()` reference. We hope that at runtime it will yield an `Integer`, which becomes its *runtime type*[33]. The Groovy code

Footnote 32. This is usually also called the “static” type but we avoid this term here to avoid further confusion.

Footnote 33. Often called the “dynamic” type - a term we avoid for the same reason.

```
Integer myInt = returnsObject()
```

is the exact equivalent of the Java code above as far as the type handling is concerned. The Groovy compiler inserts type casting logic for you that makes sure that the right-hand side of an assignment is cast to the type of the left-hand side. Consequently, when using the dynamic programming style as in

```
def myInt = returnsObject()
```

we would cast to `Object` since that is assumed when `def` is used. But this can never have any effect because *every* object is at least of type `Object` and Groovy optimizes the cast away.

Declared types give you a number of benefits. They are means of documentation and communication but most of all, they enable you to *reason about your code*. For example, consider this code snippet:

```
Integer myInt = returnsObject()
println(++myInt)
```

The second line is guarded by the first line; there is *no way*, that it would *ever* be called if `myInt` was not of type `Integer`. Therefore we can reason that the `++` operator will be found and work as expected. As a second example, consider a method definition with a parameter that bears a type marker:

```
def printNext(Integer myInt) {
    println(++myInt)
}
```

There is *no possible way*, that this method could *ever* be called with an argument that is not of type `Integer`! Even though the compiler accepts code like `printNext(new Object())` this will *never* result in calling our method above. And now to a common misconception:

Groovy types are NOT DYNAMIC, they NEVER CHANGE

If I could make the ink blink, I would! The word “dynamic” does no type of a reference, once declared, can ever change. Once we've decl `myInt` we cannot execute `myInt = new Object()`. This will t `GroovyCastException`. We can only assign a value which Gro `Integer`.

As you see, the phrase “dynamic typing” can be misleading and i

Type declarations and type casts also play an important role in the Groovy method dispatch that we will examine in XREF method_dispatch. Casts come with some additional logic to make development easier.

3.2.3. Let the casting work for you

To complete the picture, Groovy actually applies some convenience logic when casting, which is mainly concerned with casting primitive types to their wrapper classes and vice versa, arrays to lists, characters to integers, Java's type widening for numeric types, applying the “Groovy truth” (see XREF groovy_truth) for casts to boolean, calling `toString()` for casts to string, and so on. The exhaustive list can be looked up in `DefaultTypeTransformation.castToType`.

Two notable features are baked into the Groovy type casting logic that may be surprising at first, but make for really elegant code: casting lists and maps to arbitrary classes. 3.10 introduces these features by creating `Point`, `Rectangle`, and `Dimension` objects.

Listing 3.2. Casting lists and maps to arbitrary classes

```
import java.awt.*

Point topLeft  = new Point(0, 0) // classic
Point botRight = [100, 100]      // List cast
Point center   = [x:50, y:50]    // Map cast

assert botRight instanceof Point
assert center   instanceof Point

def rect = new Rectangle()
rect.location = [0, 0]               // Point
rect.size = [width:100, height:100] // Dimension
```

As you see, implicit runtime casting can lead to very readable code, especially in cases like property assignments where Groovy knows that `rect.size` is of type `java.awt.Dimension` and can cast your list or map of constructor arguments onto that. You don't have to worry about it: Groovy infers the type for you.

We have seen the value of type markers and pervasive casting. But since Groovy offers optional typing, what is the use case for omitting type markers?

3.2.4. The case for optional typing

Omitting type markers is not only convenient for the lazy programmer who does some ad-hoc scripting, but is also useful for relaying and duck typing. Suppose you get an object as the result of a method call, and you have to relay it as an argument to some other method call without doing anything with that object yourself:

```
def node = document.findMyNode()
log.info node
db.store node
```

In this case, you're not interested in finding out what the heck the actual type and package name of that node are. You are spared the work of looking them up, declaring the type, and importing the package. You also communicate: “That's just something”.

The second usage of unmarked typing is calling methods on objects that have no guaranteed type. This is often called *duck typing*, and we will explain it in more detail in section 7.3.2. This allows the implementation of generic functionality with high reusability.

For programmers with a strong Java background, it is not uncommon to start programming Groovy almost entirely using type declarations, and gradually shift into a more dynamic mode over time. This is legitimate because it allows everybody to use what they are confident with.

Rule of thumb	Experienced Groovy programmers tend to follow this rule of thumb: As soon as you *think* about the type of a reference, declare it; if you're thinking of it as “just an object,” leave the type out.

Whether you declare your types or not, you'll find that Groovy lets you do a lot more than you may expect. Let's start by looking at the ability to override operators.

3.3. Overriding operators

Overriding refers to the object-oriented concept of having types that specify behavior and subtypes that override this behavior to make it more specific. When a language bases its operators on method calls and allows these methods to be overridden, the approach is called *operator overriding*.

It's more conventional to use the term *operator overloading*, which means almost the same thing. The difference is that *overloading* suggests that you have multiple implementations of a method (and thus the associated operator) that differ only in their parameter types.

We will show you which operators can be overridden, show a full example of how overriding works in practice, and give some guidance on the decisions you need to make when operators work with multiple types.

3.3.1. Overview of overridable operators

As you saw in section 3.1.2, `1+1` is just a convenient way of writing `1.plus(1)`. This is achieved by class `Integer` having an implementation of the `plus` method.

This convenient feature is also available for other operators. Table 3.4 shows an overview.

Method-based operators

Operator	Name	Method	Works with
`a + b`	Plus	`a.plus(b)`	Number, String, StringBuffer, Collection, Map, Date, Duration
`a – b`	Minus	`a.minus(b)`	Number, String, List, Set, Date, Duration
`a * b`	Star	`a.multiply(b)`	Number, String, Collection
`a / b`	Divide	`a.div(b)`	Number

`a % b`	Modulo	`a.mod(b)`	Integral number

`a++`
 	
`++a`
	Post increment
	
	Pre increment
		`a.next()`	Iterator, Number, String, Date, (Range)

`a` `a`	Post decrement Pre decrement	`a.previous()`	Iterator, Number, String, Date, (Range)
`-a`	Unary minus	`a.negative()`	Number, ArrayList
`+a`	Unary plus	`a.positive()`	Number, ArrayList
`a ** b`	Power	`a.power(b)`	Number
`a | b`	Numerical or	`a.or(b)`	Number, Boolean, BitSet, Process
`a & b`	Numerical and	`a.and(b)`	Number, Boolean, BitSet
`a ^ b`	Numerical xor	`a.xor(b)`	Number, Boolean, BitSet
`~a`	Bitwise complement	`a.bitwiseNegate()`	Number, String (the latter returning a regular expression pattern)

`a[b]`	Subscript	`a.getAt(b)`	Object, List, Map, CharSequence, Matcher, many more
`a[b] = c`	Subscript assignment	`a.putAt(b, c)`	Object, List, Map, StringBuffer, many more
`a << b`	Left shift	`a.leftShift(b)`	Integral number, also used like “append” to StringBuffers, Writers, Files, Sockets, Lists
`a >> b`	Right shift	`a.rightShift(b)`	Number
`a >>> b`	Right shift unsigned	`a.rightShiftUnsigned(b)`	Number
`switch(a){ case b: }`	Classification	`b.isCase(a)`	Object, Class, Range, Collection, Pattern, Closure; also used with Collection c in `c.grep(b)`, which returns all items of c where `b.isCase(item)`
`a in b`	Classification	`b.isCase(a)`	see above
`a == b`	Equals	If a implements `Comparable` then `a.compareTo(b)==0` else `a.equals(b)`	Object; consider `hashCode()`[34]
`a != b`	Not equal	`! a == b`	Object

`a <=> b`	Spaceship	`a.compareTo(b)`	`java.lang.Comparable`
`a > b`	Greater than	`a.compareTo(b) > 0`	
`a >= b`	Greater than or equal to	`a.compareTo(b) >= 0`	
`a < b`	Less than	`a.compareTo(b) < 0`	
`a <= b`	Less than or equal to	`a.compareTo(b) <= 0`	
`a as type`	Enforced coercion	`a.asType (typeClass)`	Any type

The case of equals

Nothing is easier than determine whether `a==b` is true, right? Well, you want this to be a useful equality check. First, if both are null, the equal. Second, if they reference the same object they are equal withc checking. In other words `a==a` for all values of `a`.

But there is more. If `a>=b` and `a<=b` then we can deduce tha But this may impose a conflict if we have a `Comparable` obje implement `equals` consistently. This is why Groovy only `compareTo` method for `Comparable` objects when doing the and ignores the `equals` method in this case. You find the full logi in the Groovy runtime u `DefaultTypeTransformation.compareEqual(a,b)`

You can easily use any of these operators with your own classes. Just implement the respective method. Unlike in Java, there is no need to implement a specific interface.

Strictly speaking, Groovy has even more operators in addition to those in table

3.4, such as the dot operator for referencing fields and methods. Their behavior can also be overridden. They come into play in chapter 7.

This is all good in theory, but let's see it all works in practice.

3.3.2. Overridden operators in action

Listing 3.1 demonstrates an implementation of the *equals* `==` and *plus* `+` operators for a `Money` class. It is an implementation of the *Value Object*[35]pattern. We allow values of the same currency to be summed, but do not support multicurrency addition.

Footnote 35. See http://c2.com/cgi/wiki?ValueObject.

We implement `equals` indirectly by using the `@Immutable` annotation as introduced in 2.6. Remember that `==` (or `equals`) denotes object *equality* (equal values), not *identity* (same object instances).

Listing 3.3. Overriding the addition and equality operators

```
@Immutable class Money {             // #1 overrides == operator
    int     amount
    String  currency

    Money plus (Money other) {      // #2 implements + operator
        if (null == other) return this
        if (other.currency != currency) {
            throw new IllegalArgumentException(
                "cannot add $other.currency to $currency")
        }
        return new Money(amount + other.amount, currency)
    }
}

Money  buck = new Money(1, 'USD')
assert buck
assert buck        == new Money(1, 'USD')  // #3 use overridden ==
assert buck + buck == new Money(2, 'USD')  // #4 use implemented +
```

Since every immutable object automatically gets a value-based implementation of `equals`, we get away with only a minimal declaration at ❶ . The use of this operator is shown at ❸ , where one dollar becomes equal to any other dollar.

At ❷ , the `plus` operator is not *overridden* in the strict sense of the word, because there is no such operator in `Money`'s superclass (`Object`). In this case, *operator implementing* is the best wording. This is used at ❹ , where we add two `Money` objects.

To explain the difference between *overriding* and *overloading*, here is a possible overload for `Money`'s `plus` operator. In listing 3.1, `Money` can only be added to other `Money` objects. However, we might also want to be able to add `Money` with code like this:

```
assert buck + 1 == new Money(2, 'USD')
```

We can provide the additional method

```
Money plus (Integer more) {
    return new Money(amount + more, currency)
}
```

that overloads the `plus` method with a second implementation that takes an `Integer` parameter. The Groovy method dispatch finds the right implementation at runtime.

> Our `plus` operation on the `Money` class returns `Money` objects in both cases. We describe this by saying that `Money`'s `plus` operation is *closed* under its type. Whatever operation you perform on an instance of `Money`, you end up with another instance of `Money`.

This example leads to the general issue of how to deal with different parameter types when implementing an operator method. We will go through some aspects of this issue in the next section.

3.3.3. Making coercion work for you

Implementing operators is straightforward when both operands are of the same type. Things get more complex with a mixture of types, say

```
1 + 1.0
```

This adds an `Integer` and a `BigDecimal`. What is the return type? Section 3.6 answers this question for the special case of numbers, but the issue is more general. One of the two arguments needs to be promoted to the more general type. This is called *coercion*.

When implementing operators, there are three main issues to consider as part of coercion.

Supported argument types

You need to decide which argument types and values will be allowed. If an operator must take a potentially inappropriate type, throw an

`IllegalArgumentException` where necessary. For instance, in our `Money` example, even though it makes sense to use `Money` as the parameter for the `plus` operator, we don't allow different currencies to be added together.

Promoting more specific arguments

If the argument type is a more specific one than your own type, promote it to *your* type and return an object of *your* type. To see what this means, consider how you might implement the `plus` operator if you were designing the `BigDecimal` class, and what you'd do for an `Integer` argument.

`Integer` is more specific than `BigDecimal`: Every `Integer` value can be expressed as a `BigDecimal`, but the reverse isn't true. So for the `BigDecimal.plus(Integer)` operator, we would consider promoting the `Integer` to `BigDecimal`, performing the addition, and then returning another `BigDecimal`--even if the result could accurately be expressed as an `Integer`.

Handling more general arguments with double dispatch

If the argument type is more general, call *its* operator method with *yourself* (“this,” the current object) as an argument. Let *it* promote *you*. This is also called *double dispatch*[36], and it helps to avoid duplicated, asymmetric, possibly inconsistent code. Let's reverse our previous example and consider `Integer.plus (BigDecimal operand)`.

Footnote 36. Double dispatch is usually used with overloaded methods: *`a.method(b)`* calls *`b.method(a)`* where *`method`* is overloaded with *`method(TypeA)`* and *`method(TypeB)`* .

We would consider returning the result of the expression `operand.plus(this)`, delegating the work to `BigDecimal`'s `plus(Integer)` method. The result would be a `BigDecimal`, which is reasonable--it would be odd for `1+1.5` to return an `Integer` but `1.5+1` to return a `BigDecimal`.

Of course, this is only applicable for *commutative*[37] operators. Test rigorously, and beware of endless cycles.

Footnote 37. An operator is *commutative* if the operands can be exchanged without changing the result of the operation. For example, *plus* is usually required to be commutative (`a+b==b+a`) but *minus* is not (`a-b!=b-a`).

Groovy's conventional behavior

Groovy's general strategy of coercion is to return the most general type. Other languages such as Ruby try to be smarter and return the *least* general type that can be used without losing information from range or precision. The Ruby way saves memory at the expense of processing time. It also requires that the language promote a type to a more general one when the operation would generate an overflow of that type's range. Otherwise, intermediary results in a complex calculation could truncate the result.

Now that you know how Groovy handles types in general, we can delve deeper into what it provides for each of the datatypes it supports at the language level. We begin with the type that is probably used more than any other non-numeric type: the humble string.

3.4. Working with strings

Considering how widely strings are used, many languages--including Java--provide few language features to make them easier to handle. Scripting languages tend to fare better in this regard than mainstream application languages, so Groovy takes on board some of those extra features. This section examines what's available in Groovy and how to make the most of the extra abilities.

Groovy strings come in two flavors: plain strings and *GString*s. Plain strings are instances of `java.lang.String`, and GStrings are instances of `groovy.lang.GString`. GStrings allow placeholder expressions to be resolved and evaluated at runtime. Many scripting languages have a similar feature, usually called *string interpolation*, but it's more primitive than the GString feature of Groovy. Let's start by looking at each flavor of string and how they appear in code.

3.4.1. Varieties of string literals

Java allows only one way of specifying string literals: placing text in quotes “like this”. If you want to embed dynamic values within the string, you have to either call a formatting method (made easier but still far from simple in Java 1.5) or concatenate each constituent part. If you specify a string with a lot of backslashes in it (such as a Windows file name or a regular expression), your code becomes hard to read, because you have to double the backslashes. If you want a lot of text spanning several lines in the source code, you have to make each line contain a complete string (or several complete strings).

Groovy recognizes that not every use of string literals is the same, so it offers a variety of options. These are summarized in table 3.5.

Summary of the string literal styles available in Groovy

Start/end characters	Example	Placeholder resolved?	Backslash escapes?
Single quote	`'hello Dierk'`	No	Yes
Double quote	`"hello $name"`	Yes	Yes

Triple single quote (`'''`)	`'''==========`
	
	`Total: $0.02`
	
	`=========='''`
		No	Yes

Triple double quote (`"""`)	`"""first $line`
	
	`second $line`
	
	`third $line"""`
		Yes	Yes

Forward slash	`/x(\d*)y/`	Yes	Occasionally[38]

The aim of each form is to specify the text data you want with the minimum of fuss. Each of the forms has a single feature that distinguishes it from the others:

- The single-quoted form never pays any attention to placeholders. This is closely equivalent to Java string literals.

- The double-quoted form is the equivalent of the single-quoted form, except that if the text contains unescaped dollar signs, it is treated as a GString instead of a plain string. GStrings are covered in more detail in the next section.

- The triple-quoted form (or *multiline* string literal) allows the literal to span several lines. New lines are always treated as `\n` regardless of the platform, but all other whitespace is preserved as it appears in the text file. Multiline string literals may also be GStrings, depending on whether single quotes or double quotes are used. Multiline string literals act similar to `HERE`-documents in Ruby or Perl.

- The *slashy* form of string literal allows strings with backslashes to be specified simply without having to escape all the backslashes. This is particularly useful with regular expressions, as you'll see later. Only when a backslash is followed by a *u* does it need to be escaped[39]--at which point life is slightly harder, because specifying `\u` involves using a GString or specifying the Unicode escape sequence for a backslash.

Footnote 39. This is slightly tricky in a slashy string and involves either using a GString such as `/${'\\'}/` or using the Unicode escape sequence. A similar issue occurs if you want to use a dollar sign. This is a small (and rare) price to pay for the benefits available, however.

As we hinted earlier, Groovy uses a similar mechanism for specifying special characters, such as linefeeds and tabs. In addition to the Java escapes, dollar signs can be escaped in Groovy to allow them to be easily specified without the compiler treating the literal as a GString. The full set of escaped characters is specified in table 3.6.

Escaped characters as known to Groovy

Escaped special character	Meaning
`\b`	Backspace
`\t`	Tab
`\r`	Carriage return
`\n`	Line feed
`\f`	Form feed
`\\`	Backslash
`\$`	Dollar sign

`\uabcd`	Unicode character U+ *abcd* (where *a*, *b*, *c* and *d* are hex digits)
`\abc`[40]	Unicode character U+ *abc* (where *a*, *b*, and *c* are octal digits, and *b* and *c* are optional)
`\'`	Single quote
`\"`	Double quote

Note that in a double-quoted string, single quotes don't need to be escaped, and vice versa. In other words, `'I said, "Hi."'` and `"don't"` both do what you hope they will. For the sake of consistency, both still *can* be escaped in each case. Likewise, dollar signs can be escaped in single-quoted strings, even though they don't need to be. This makes it easier to switch between the forms.

Note that Java uses single quotes for *character* literals, but as you have seen, Groovy cannot do so because single quotes are already used to specify *strings*. However, you can achieve the same as in Java when providing the type explicitly:

```
char a = 'x'
```

or

```
Character b = 'x'
```

The `java.lang.String 'x'` is cast into a `java.lang.Character`. If you want to coerce a string into a character at other times, you can do so in either of the following ways:

```
'x' as char
```

or

```
'x'.toCharacter()
```

As a GDK goody, there are more `to*` methods to convert a string, such as `toInteger`, `toLong`, `toFloat`, and `toDouble`.

Whichever literal form is used, unless the compiler decides it is a GString, it ends up as an instance of `java.lang.String`, just like Java string literals. So far, we have only teased you with allusions to what GStrings are capable of. Now it's time to spill the beans.

3.4.2. Working with GStrings

GStrings are like strings with additional capabilities.[41] They are literally declared in double quotes. What makes a double-quoted string literal a GString is the appearance of placeholders. Placeholders may appear in a full `${expression}` syntax or an abbreviated `$reference` syntax. See the examples in 3.12.

Footnote 41. `groovy.lang.GString` isn't actually a subclass of `java.lang.String`, and couldn't be, because `String` is final. However, GStrings can usually be *used* as if they were strings--Groovy coerces them into strings when it needs to.

Listing 3.4. Working with GStrings

```
def me       = 'Tarzan'                                           //|#1 Abbreviated
def you      = 'Jane'                                             //|#1 dollar syntax
def line     = "me $me - you $you"                                //|#1
assert  line == 'me Tarzan - you Jane'                            //|#1

def date = new Date(0)                                            //|#2 Extended
def out  = "Year $date.year Month $date.month Day $date.date" //|#2 abbreviation
assert out == 'Year 70 Month 0 Day 1'                             //|#2

out = "Date is ${date.toGMTString()} !"                           //|#3 Full syntax with
assert out == 'Date is 1 Jan 1970 00:00:00 GMT !'                 //|#3 curly braces
                                                                  //#4 Multiline GString
def sql = """
SELECT FROM MyTable
  WHERE Year = $date.year
"""
assert sql == """
SELECT FROM MyTable
  WHERE Year = 70
"""                                                               //#4 Multiline GString

out = "my 0.02$"                                                 //|#5 Literal dollar si
assert out == 'my 0.02$'                                          //|#5
```

Within a GString, simple references to variables can be dereferenced with the dollar sign. This simplest form is shown at ❶ , whereas ❷ shows this being extended to use property accessors with the dot syntax. You will learn more about accessing properties in chapter 7.

The full syntax uses dollar signs and curly braces, as shown at ❸ . It allows arbitrary Groovy expressions within the curly braces. The curly braces denote a *closure*.

In real life, GStrings are handy in templating scenarios. A GString is used in ❹ to create the string for an SQL query. Groovy provides even more sophisticated templating support, as shown in chapter 8. If you need a dollar character within a template (or any other GString usage), you must escape it with a backslash as shown in ❺ .

Although GStrings behave like `java.lang.String` objects for all operations that a programmer is usually concerned with, they are implemented differently to capture the fixed and the dynamic parts (the so-called *values*) separately. This is revealed by the following code:

```
def me      = 'Tarzan'
def you     = 'Jane'
def line    = "me $me - you $you"
assert line == 'me Tarzan - you Jane'
assert line instanceof GString
assert line.strings[0] == 'me '
assert line.strings[1] == ' - you '
assert line.values[0]  == 'Tarzan'
assert line.values[1]  == 'Jane'
```

Placeholder evaluation time

Each placeholder inside a GString is evaluated at declaration time and the resulting *value* is stored in the GString object. By the time the GString is converted into a `java.lang.String` (by calling its `toString` method or casting it to a string), each value gets written [42] to the string. Because the logic of how to write a value can be elaborate for certain types (most notably *closures*), this behavior can be used in advanced ways that make the evaluation of such placeholders appear to be lazy. See chapter 13 for examples of this.

Footnote 42. See `Writer.write(Object)` in section 8.2.4.

You have seen the Groovy language support for declaring strings. What follows is an introduction to the use of strings in the Groovy *library*. This will also give you a first impression of the seamless interplay of Java and Groovy. We start in typical Java style and gradually slip into Groovy mode, carefully watching each step.

3.4.3. From Java to Groovy

Now that you have your strings easily declared, you can have some fun with them.

Because they are objects of type `java.lang.String`, you can call `String`'s methods on them or pass them as parameters wherever a string is expected, such as for easy console output:

```
System.out.print("Hello Groovy!");
```

This line is equally valid Java and Groovy. You can also pass a literal Groovy string in single quotes:

```
System.out.print('Hello Groovy!');
```

Because this is such a common task, the GDK provides a shortened syntax:

```
print('Hello Groovy!');
```

You can drop parentheses and semicolons, because they are optional and do not help readability in this case. The resulting Groovy style boils down to

```
print 'Hello Groovy!'
```

Looking at this last line only, you cannot tell whether this is Groovy, Ruby, Perl, or one of several other line-oriented scripting languages. It may not look sophisticated, but it boils the code down to the *essence* by cutting down on *ceremony* (Stuart Halloway).

Listing 3.3 presents more of the mix-and-match between core Java and additional GDK capabilities. How would you judge the signal-to-noise ratio of each line?

In this listing, we're using getAt(x). Should we also show charAt(x) to demonstrate the Java equivalent

Listing 3.5. A miscellany of string operations

```
String greeting = 'Hello Groovy!'

assert greeting.startsWith('Hello')

assert greeting.getAt(0) == 'H'
assert greeting[0]       == 'H'

assert greeting.indexOf('Groovy') >= 0
assert greeting.contains('Groovy')

assert greeting[6..11]  == 'Groovy'

assert 'Hi' + greeting - 'Hello' == 'Hi Groovy!'

assert greeting.count('o') == 3

assert 'x'.padLeft(3)      == '  x'
assert 'x'.padRight(3,'_') == 'x__'
assert 'x'.center(3)        == ' x '
assert 'x' * 3              == 'xxx'
```

These self-explanatory examples give an impression of what is possible with strings in Groovy. If you have ever worked with other scripting languages, you may notice that a useful piece of functionality is missing from listing 3.3: changing a string in place. Groovy cannot do so because it works on instances of `java.lang.String` and obeys Java's *invariant* of strings being *immutable*.

Before you say "What a lame excuse!" here is Groovy's answer to changing strings: Although you cannot work on `String`, you can still work on `StringBuffer`![43] On a `StringBuffer`, you can work with the `<<` *left shift* operator for appending and the subscript operator for in-place assignments. Using the *left shift* operator on `String` returns a `StringBuffer`. Here is the `StringBuffer` equivalent to listing 3.3:

Footnote 43. Future versions may use a `StringBuilder` instead. `StringBuilder` was introduced in Java 1.5 to reduce the synchronization overhead of `StringBuffers`. Typically, `StringBuffers` are used only in a single thread and then discarded--but `StringBuffer` itself is thread-safe, at the expense of synchronizing each method call.

```
def greeting = 'Hello'

greeting <<= ' Groovy'  // #1 Leftshift and assign

assert greeting instanceof java.lang.StringBuffer

greeting << '!'         //#2 Leftshift on StringBuffer

assert greeting.toString() == 'Hello Groovy!'

greeting[1..4] = 'i'    //#3 Substring 'ello' becomes 'i'

assert greeting.toString() == 'Hi Groovy!'
```

Note

Although the expression `stringRef << string` returns a `StringBuffer`, that `StringBuffer` is not automatically assigned to the *stringRef* (see ❶). When used on a `String`, it needs explicit assignment; on `StringBuffer` it doesn't. With a `StringBuffer`, the data in the existing object is changed (see ❷)--with a `String` we can't change the existing data, so we have to return a new object instead.

Throughout the next sections, you will gradually add to what you have learned about strings as you discover more language features. `String` has gained several new methods in the GDK. You've already seen a few of these, but you'll see more as we talk about working with regular expressions and lists. The complete list of GDK methods on strings is listed in appendix C.

Working with strings is one of the most common tasks in programming, and for script programming in particular: reading text, writing text, cutting words, replacing phrases, analyzing content, search and replace--the list is amazingly long. Think about your own programming work. How much of it deals with strings?

Groovy supports you in these tasks with comprehensive string support, but that's not the whole story. The next section introduces *regular expressions*, which cut through text like a chainsaw: difficult to operate but extremely powerful.

3.5. Working with regular expressions

Once a programmer had a problem. He thought he could solve it with a regular expression. Now he had two problems.

-- from a fortune cookie

Suppose you had to prepare a table of contents for this book. You would need to collect all the headings like “3.5 Working with regular expressions”--paragraphs that start with a number or with a number, a dot, and another number. The rest of the paragraph would be the heading. This would be cumbersome to code naïvely: iterate over each character; check whether it is a line start; if so, check whether it is a digit; if so, check whether a dot and a digit follow. Puh--lots of rope, and we haven't even covered numbers that have more than one digit.

Regular expressions come to the rescue. They allow you to *declare* such a *pattern* rather than programming it. Once you have the pattern, Groovy lets you work with it in numerous ways.

Regular expressions are prominent in scripting languages and have also been available in the Java library since JDK 1.4. Groovy relies on Java's *regex* (*reg*ular *ex*pression) support and adds three operators for convenience:

- The regex *find* operator =~
- The regex *match* operator ==~
- The regex *pattern* operator *`~String`*

An in-depth discussion about regular expressions is beyond the scope of this book. Our focus is on Groovy, not on regexes. We give the shortest possible introduction to make the examples comprehensible and provide you with a jump-start.

Regular expressions are defined by *patterns*. A pattern can be anything from a simple character, a fixed string, or something like a date format made up of digits and delimiters, up to descriptions of balanced parentheses in programming languages. Patterns are declared by a sequence of symbols. In fact, the pattern description is a language of its own. Some examples are shown in table 3.7. Note that these are the raw patterns, not how they would appear in string literals. In other words, if you stored the pattern in a variable and printed it out, this is what you'd want to see. It's important to make the distinction between the pattern itself and how it's represented in code as a literal.

Simple regular expression pattern examples

Pattern	Meaning
`some text`	Exactly “some text”.
`some\s+text`	The word “some” followed by one or more whitespace characters followed by the word “text”.
`^\d+(\.\d+)? (.*)`	Our introductory example: headings of level one or two. `^` denotes a line start, `\d` a digit, `\d+` one or more digits. Parentheses are used for grouping. The question mark makes the first group optional. The second group contains the title, made of a dot for any character and a star for any number of such characters.
`\d\d/\d\d/\d\d\d\d`	A date formatted as exactly two digits followed by slash, two more digits followed by a slash, followed by exactly four digits.

A pattern like one of the examples in table 3.7 allows you to declare *what* you are looking for, rather than having to program *how* to find something. Next we'll see how patterns appear as literals in code and what can be done with them. We will then revisit our initial example with a full solution, before examining some performance aspects of regular expressions and finally showing how they can be used for classification in `switch` statements and for collection filtering with the `grep` method.

3.5.1. Specifying patterns in string literals

How do you put the sequence of symbols that declares a pattern inside a string?

In Java, this causes confusion. Patterns use lots of backslashes, and to get a backslash in a Java string literal, you need to double it. This leads to Java strings which are very hard to read in terms of the raw pattern involved.. It gets even worse if you need to match an actual backslash in your pattern--the pattern language escapes that with a backslash too, so the Java regex string literal needed to match `a\b` is `"a\\\\b"`.

Groovy does much better. As you saw earlier, there is the *slashy* form of string literal, which doesn't require you to escape the backslash character and still works like a normal GString. Listing 3.4 shows how to declare patterns conveniently.

Listing 3.6. Regular expression GStrings

```
assert "abc" == /abc/
assert "\d" == /d/

def reference = "hello"
assert reference == /$reference/

assert "$" == /$/
```

The slashy syntax doesn't require the dollar sign to be escaped. Note that you have the choice to declare patterns in either kind of string.

Tip Sometimes the slashy syntax interferes with other valid Groovy expressions such as line comments or numerical expressions with multiple slashes for division. When in doubt, put parentheses around your pattern like `(/pattern/)`. Parentheses force the parser to interpret the content as an expression.

Symbols

The key to using regular expressions is knowing the pattern symbols. For convenience, table 3.8 provides a short list of the most common ones. Put an earmark on this page so you can easily look up the table. You will use it a lot.

Regular expression symbols (excerpt)

Symbol	Meaning
`.`	Any character
`^`	Start of line (or start of document, when in single-line mode)
`$`	End of line (or end of document, when in single-line mode)
`\d`	Digit character
`\D`	Any character except digits
`\s`	Whitespace character
`\S`	Any character except whitespace
`\w`	Word character
`\W`	Any character except word characters
`\b`	Word boundary
`()`	Grouping

`(` *x* `|` *y* `)`	*x* or *y*, as in `(Groovy|Java|Ruby)`
`\1`	Backmatch to group one: for example, find doubled characters with `(.)\1`
x `*`	Zero or more occurrences of *x*
x `+`	One or more occurrences of *x*
x `?`	Zero or one occurrence of *x*
x `{` *m* `,` *n* `}`	At least *m* and at most *n* occurrences of *x*
x `{` *m* `}`	Exactly *m* occurrences of *x*
`[a-f]`	Character class containing the characters *a, b, c, d, e, f*
`[^a]`	Character class containing any character except *a*
`(?is:x)`	Switches mode when evaluating *x* ; i turns on `ignoreCase`, `s` means single-line mode

Tip Symbols tend to have the same first letter as what they represent: for example, *d*igit, *s*pace, *w*ord, and *b*oundary. Uppercase symbols define the complement; think of them as a warning sign for *no*.

More to consider:

- Use grouping properly. The *expanding* operators such as *star* and *plus* bind closely; `ab+` matches `abbbb`. Use `(ab)+` to match `ababab`.
- In normal mode, the expanding operators are *greedy*, meaning they try to match the longest substring that matches the pattern. Add an additional question mark after the operator to put them into *restrictive* mode. You may be tempted to extract the *href* from an HTML anchor element with this regex: `href="(.*)"`. But `href= "(.*?)"` is probably better. The first version matches until the *last* double quote in your text; the latter matches until the *next* double quote.[44]

Footnote 44. This is only to explain the greedy behavior of regular expression, not to explain how HTML is parsed correctly, which would involve a lot of other topics such as ordering of attributes, spelling variants, and so forth.

This is only a brief description of the regex pattern format, but a complete specification comes with your JDK, as part of the Javadoc for `java.util.regex.Pattern`. It may change marginally between JDK versions; for JDK 1.5, it can be found online at http://java.sun.com/j2se/1.5/docs/api/java/util/regex/Pattern.html.

Use the 1.6 link instead? See the Javadoc to learn more about different evaluation modes, positive and negative lookahead, back references, and posix characters.

It always helps to test your expressions before putting them into code. There are online applications that allow interactive testing of regular expressions: for example, http://www.nvcc.edu/home/drodgers/ceu/resources/test_regexp.asp. You should be aware that not all regular expression pattern languages are exactly the same. You may get unexpected results if you take a regular expression designed for use in .NET and apply it in a Java or Groovy program. Although there aren't many differences, the differences that do exist can be hard to spot. Even if you take a regular expression from a book or a web site, you should still test that it works in your code.

Once you have declared the pattern you want, you need to tell Groovy how to apply it. We will explore a whole variety of usages.

3.5.2. Applying patterns

For a given string and pattern, Groovy supports the following tasks for regular expressions:

- Tell whether the pattern fully matches the whole string.

- Tell whether there is an occurrence of the pattern in the string.
- Count the occurrences.
- Do something with each occurrence.
- Replace all occurrences with some text.
- Split the string into multiple strings by cutting at each occurrence.

Listing 3.5 shows how Groovy sets patterns into action. Unlike most other examples, this listing contains some comments. This reflects real life and is not for illustrative purposes. The use of regexes is best accompanied by this kind of comment for all but the simplest patterns.

Listing 3.7. Regular expressions

```
def twister = 'she sells sea shells at the sea shore of seychelles'

// twister must contain a substring of size 3
// that starts with s and ends with a
assert twister =~ /s.a/                                       // #1 Regex find oper

def finder = (twister =~ /s.a/)                               // #2 Find expression
assert finder instanceof java.util.regex.Matcher              // #2 matcher object

// twister must contain only words delimited by single spaces
assert twister ==~ /(w+ w+)*/                               // #3 Regex match opera

def WORD = /w+/
matches = (twister ==~ /($WORD $WORD)*/)                      // #4 Match expressio
assert matches instanceof java.lang.Boolean                   // #4 to a boolean

assert (twister ==~ /s.e/) == false                           // #5 Match is full u

def wordsByX = twister.replaceAll(WORD, 'x')
assert wordsByX == 'x x x x x x x x x x'

def words = twister.split(/ /)                                // #6 Split returns a
assert words.size() == 10
assert words[0] == 'she'
```

❶ and ❷ have an interesting twist. Although the regex *find* operator evaluates to a `Matcher` object, it can also be used as a Boolean conditional. We will explore how this is possible when examining the “Groovy Truth” in chapter 6.

Tip

To remember the difference between the =~ *find* operator and the ==~ *match* operator, recall that *match* is more restrictive, because the pattern needs to cover the whole string. The demanded coverage is “longer” just like the operator itself.

See your Javadoc for more information about the `java.util.regex.Matcher` object, such as how to walk through all the matches and how to work with *groupings* within each match.

Common regex pitfalls

You do not need to fall into the regex traps yourself. We have already done this for you. We have learned the following:

- When things get complex (note, this is *when*, not *if*), comment verbosely.
- Use the slashy syntax instead of the regular string syntax, or you will get lost in a forest of backslashes.
- Don't let your pattern look like a toothpick puzzle. Build your pattern from subexpressions like `WORD` in listing 3.5.
- Put your assumptions to the test. Write some assertions or unit tests to test your regex against static strings. Please don't send us any more flowers for this advice; an email with the subject “Assertions saved my life today” will suffice.

3.5.3. Patterns in action

You're now ready to do everything you wanted to do with regular expressions, except we haven't covered “do something with each occurrence”. *Something* and *each* sounds like a cue for a closure to appear, and that's the case here. `String` has a method called `eachMatch` that takes a regex as a parameter along with a closure that defines what to do on each match.

What is a match?

A match is the occurrence of a regular expression pattern in therefore a string: a substring of the original string. When the p groupings like in `/begin(.*?)end/`, we need to know more in just the string matching the whole pattern, but also what part of that each group. Therefore, the match becomes a list of strings, contai match at position 0 with group matches being available as `match` group number *n*. Groups are numbered by the sequence of parentheses.

The match gets passed into the closure for further analysis. In our musical example in listing 3.6, we append each match to a result string.

Listing 3.8. Working on each match of a pattern

```
def myFairStringy = 'The rain in Spain stays mainly in the plain!'
```

```
// words that end with 'ain': bw*ainb
def wordEnding = /w*ain/
def rhyme = /b$wordEndingb/
def found = ''
myFairStringy.eachMatch(rhyme) { match ->                      // #1 String.eachMatc
    found += match + ' '
}
assert found == 'rain Spain plain '

found = ''
(myFairStringy =~ rhyme).each { match ->                       // #2 Matcher.each {}
    found += match + ' '
}
assert found == 'rain Spain plain '

def cloze = myFairStringy.replaceAll(rhyme){ it-'ain'+'___' }  //#3 String.replaceAl
assert cloze == 'The r___ in Sp___ stays mainly in the pl___!'
```

There are two different ways to iterate through matches with identical behavior: use ❶ *`String.eachMatch(Pattern)`* , or use ❷ *`Matcher.each()`* , where the `Matcher` is the result of applying the regex find operator to a string and a pattern. ❸ shows a special case for replacing each match with some dynamically derived content from the given closure. The variable `it` refers to the matching substring. The result is to replace “ain” with underscores, but only where it forms part of a rhyme.

In order to fully understand how the Groovy regular expression support works, we need to look at the `java.util.regex.Matcher` class. It is a JDK class that encapsulates knowledge about

- How often and at what position a pattern matches
- The groupings for each match

The GDK enhances the `Matcher` class with simplified array-like access to this information. In Groovy, you can think about a *matcher* as if it was a list of all its *matches*. This is what happens in the following example that matches all non-whitespace characters:

```
def matcher = 'a b c' =~ /S/

assert matcher[0]     == 'a'
assert matcher[1..2]  == ['b','c']
assert matcher.size() == 3
```

The interesting part comes with *groupings* in the match. If the pattern contains parentheses to define groups, then the result of asking for a particular match is an array of strings are than a single one: the same behavior as we mentioned for eachMatch. Again, the first result (at index 0) is the match for the whole pattern. Consider this example, where each match finds pairs of strings that are separated by a colon. For later processing, the match is split into two groups, for the left and the right string:

```
def matcher = 'a:1 b:2 c:3' =~ /(S+):(S+)/

assert matcher.hasGroup()
assert matcher[0] == ['a:1', 'a', '1']  // 1st match
assert matcher[1][2] == '2'             // 2nd match, 2nd group
```

In other words, what `matcher[0]` returns depends on whether the pattern contains groupings.

This also applies to the matcher's `each` method, which comes with a convenient notation for groupings. When the processing closure defines multiple parameters, the list of groups is distributed over them:

```
def matcher = 'a:1 b:2 c:3' =~ /(S+):(S+)/
matcher.each { full, key, value  ->
    assert full.size()  == 3
    assert key.size()   == 1  // a,b,c
    assert value.size() == 1  // 1,2,3
}
```

This matcher matches three times passing the full match and the two groups into the closure on each match. The above enables us to assign meaningful names to the group matches. We decided to call them `key` and `value`, which much better reveals their intent than `match[1]` and `match[2]` would.

We advise to use group names whenever the group count is fix. Groovy supports the spreading of match groups over closure parameters for all methods that pass a match into a closure. For example you can use it with the `String.eachMatch(regex){match->}` method.

Implementation detail	Groovy internally stores the most recently used matcher (per thread). It can be retrieved with the static property `Matcher.lastMatcher`. You can also set the index property of a matcher to make it look at the respective match with `matcher.index = x`. Both can be useful in some exotic corner cases. See `Matcher`'s API documentation for details.

We will revisit the `Matcher` class later in various places. It is particularly interesting because it plays so well with Groovy's approach of letting classes decide how to iterate over themselves and reusing that behavior pervasively. `Matcher` and `Pattern` work in combination and are the key abstractions for regexes in Java and Groovy. You have seen `Matcher`, and we'll have a closer look at the `Pattern` abstraction next.

3.5.4. Patterns and performance

Finally, let's look at performance and the pattern operator ~*String*.

The pattern operator transforms a string into an object of type `java.util.regex.Pattern`. For a given string, this pattern object can be asked for a *matcher* object.

The rationale behind this construction is that patterns are internally backed by a *finite state machine* that does all the high-performance magic. This machine is compiled when the pattern object is created. The more complicated the pattern, the longer the creation takes. In contrast, the *matching* process as performed by the machine is extremely fast.

The pattern operator allows you to split pattern-creation time from pattern-matching time, increasing performance by reusing the finite state machine. Listing 3.7 shows a poor-man's performance comparison of the two approaches. The precompiled pattern version is at least twice as fast (although these kinds of measurements can differ wildly).

Listing 3.9. Increase performance with pattern reuse.

```
def twister = 'she sells sea shells at the sea shore of seychelles'
// some more complicated regex:
// word that starts and ends with same letter
def regex = /b(w)w*1b/
def many  = 100 * 1000

start = System.nanoTime()
many.times{
    twister =~ regex                        // #1 Find operator with implicit patter
}
timeImplicit = System.nanoTime() - start

start = System.nanoTime()
pattern = ~regex                            // #2 Explicit pattern construction
many.times{
    pattern.matcher(twister)                // #3 Apply pattern on a string
}
timePredef = System.nanoTime() - start

assert timeImplicit > timePredef * 2        // #4 up to factor 5
```

To find words that start and end with the same character, we used the `\1` backmatch to refer to that character. We prepared its usage by putting the word's first character into a group, which happens to be group 1.

Note the difference in spelling in ❶ . This is not `a =~ b` but `a = ~b`. Tricky.

Use whitespace wisely

The observant reader may spot a language issue: What happens if yc without any whitespace? Is that the `=~` *find* operator, or is it an assig pattern to `a`? For the human reader, it is ambiguous. Not so for the G greedy and will parse this as the *find* operator.

It goes without saying that being explicit with whitespace is goo style, even when the meaning is unambiguous for the parser. Do human reader, which will probably be you.

Don't forget that performance should usually come second to readability--at least to start with. If reusing a pattern means bending your code out of shape, you should ask yourself how critical the performance of that particular area is before making the change. Measure the performance in different situations with each version of the code, and balance ease of maintenance with speed and memory requirements.

3.5.5. Patterns for classification

Listing 3.8 completes our journey through the domain of patterns. The `Pattern` object, as returned from the *pattern* operator, implements an `isCase(String)` method that is equivalent to a full match of that pattern with the string. This classification method is a prerequisite for using patterns conveniently with the `in` operator, the `grep` method and in `switch` cases.

The example classifies words that consist of exactly four characters. The pattern therefore consists of the word character class `\w` followed by the `{4}` quantification.

Listing 3.10. Patterns for classification

```
def fourLetters = ~/w{4}/

assert fourLetters.isCase('work')

assert 'love' in fourLetters

switch('beer'){
    case ~/w{4}/ : assert true; break
    default      : assert false
}
beasts = ['bear','wolf','tiger','regex']
assert beasts.grep(fourLetters) == ['bear','wolf']
```

Tip Classifications read nicely with `in`, `switch` and `grep`. It's rare to call `classifier.isCase(candidate)` directly, but when you see such a call it's easiest to read it from right to left: “*candidate* is a case of *classifier*”.

Patterns are also prevalent in the Groovy library (see XREF GDK). Most of those methods give you the choice between using either a string that describes the regular expression (conventionally this parameter is called “regex”) or supplying a pattern object instead (conventionally called “pattern”). This applies to the following methods on `String`:

```
String find     (Pattern pattern)
String find     (Pattern pattern) { match -> ... }
List   findAll  (Pattern pattern)
List   findAll  (Pattern pattern) { match -> ... }
String eachMatch(Pattern pattern) { match -> ... }
```

Some notable examples of this rule are

```
replaceFirst(Pattern pattern, String replacement)
replaceAll  (String regex) { match -> ... }
replaceAll  (Pattern pattern, String replacement)
matches     (Pattern pattern)
```

and the various forms of `splitEachLine`. Another special case is `minus` because you can use it to either remove a fixed substring from a string or remove a pattern match. But the latter only works if the operand is already a `Pattern` object rather than a regex string, obviously--otherwise the meaning of `'a' - 'b'` would be ambiguous.

At times, regular expressions can be difficult beasts to tame, but mastering them adds a new quality to all text-manipulation tasks. Once you have a grip on them, you'll hardly be able to imagine having programmed (some would say *lived*) without them. Writing this book without their help would have been very hard indeed. Groovy makes regular expressions easily accessible and straightforward to use.

This concludes our coverage of text-based types, but of course computers have always dealt with numbers as well as text. Working with numbers is easy in most programming languages, but that doesn't mean there's no room for improvement. Let's see how Groovy goes the extra mile when it comes to numeric types.

3.6. Working with numbers

The available numeric types and their declarations in Groovy were introduced in section 3.1.

We've already seen that for decimal numbers, the default type is `java.math.BigDecimal`. This is a feature to get around the most common misconceptions about floating-point arithmetic. We're going to look at which type is used where and what extra abilities have been provided for numbers in the GDK.

3.6.1. Coercion with numeric operators

It is always important to understand what happens when you use one of the numeric operators.

Most of the rules for the addition, multiplication, and subtraction operators are the same as in Java, but there are some changes regarding floating-point behavior, and `BigInteger` and `BigDecimal` also need to be included. The rules are straightforward. The first rule to match the situation is used.

For the operations `+`, `-`, and `*`:

- If either operand is a `Float` or a `Double`, the result is a `Double`. (In Java, when only `Float` operands are involved, the result is a `Float` too.)
- Otherwise, if either operand is a `BigDecimal`, the result is a `BigDecimal`.
- Otherwise, if either operand is a `BigInteger`, the result is a `BigInteger`.
- Otherwise, if either operand is a `Long`, the result is a `Long`.
- Otherwise, the result is an `Integer`.

Table 3.9 depicts the scheme for quick lookup. Types are abbreviated by uppercase letters.

Numerical coercion

`+-*`	B	S	I	C	L	BI	BD	F	D
`Byte`	I	I	I	I	L	BI	BD	D	D

`Short`	I	I	I	I	L	BI	BD	D	D
`Integer`	I	I	I	I	L	BI	BD	D	D
`Character`	I	I	I	I	L	BI	BD	D	D
`Long`	L	L	L	L	L	BI	BD	D	D
`BigInteger`	BI	BI	BI	BI	BI	BI	BD	D	D
`BigDecimal`	BD	BD	BD	BD	BD	BD	BD	D	D
`Float`	D	D	D	D	D	D	D	D	D
`Double`	D	D	D	D	D	D	D	D	D

Other aspects of coercion behavior:

- Like Java but unlike Ruby, no coercion takes place when the result of an operation exceeds the current range, except for the power operator.
- For *division*, if any of the arguments is of type `Float` or `Double`, the result is of type `Double`; otherwise the result is of type `BigDecimal` with the maximum precision of both arguments, rounded half up. The result is normalized--that is, without trailing zeros.
- Integer division (keeping the result as an integer) is achievable through explicit casting or by using the `intdiv()` method.
- The *shifting* operators are only defined for types `Integer` and `Long`. They do not coerce to other types.
- The *power* operator coerces to the next best type that can take the result in terms of range and precision, in the sequence `Integer`, `Long`, `Double`.
- The *equals* operator coerces to the more general type before comparing.

Rules can be daunting without examples, so this behavior is demonstrated in table 3.10.

Numerical expression examples

Expression	Result type	Comments
`1f*2f`	`Double`	In Java, this would be `Float`.
`(Byte)1+(Byte)2`	`Integer`	As in Java, integer arithmetic is always performed in at least 32 bits.
`1*2L`	`Long`	
`1/2`	`BigDecimal (0.5)`	In Java, the result would be the integer 0.
`(int)(1/2)`	`Integer (0)`	This is normal coercion of `BigDecimal` to `Integer`.
`1.intdiv(2)`	`Integer (0)`	This is the equivalent of the Java `1/2`.
`Integer.MAX_VALUE+1`	`Integer`	Non- *power* operators wrap without promoting the result type.
`2**31`	`Integer`	
`2**33`	`Long`	The *power* operator promotes where necessary.

`2**3.5`	`Double`	
`2G+1G`	`BigInteger`	
`2.5G+1G`	`BigDecimal`	
`1.5G==1.5F`	`Boolean (true)`	The `Float` is promoted to a `BigDecimal` before comparison.
`1.1G==1.1F`	`Boolean (false)`	1.1 can't be exactly represented as a `Float` (or indeed a `Double`), so when it is promoted to `BigDecimal`, it isn't equal to the exact `BigDecimal` 1.1G but rather 1.100000023841858G.

The only surprise is that there is no surprise. In Java, results like in the fourth row are often surprising--for example, `(1/2)` is always zero because when both operands of division are integers, only integer division is performed. To get `0.5` in Java, you need to write `(1f/2)`.

This behavior is especially important when using Groovy to enhance your application with user-defined input. Suppose you allow super-users of your application to specify a formula that calculates an employee's bonus, and a business analyst specifies it as `businessDone * (1/3)`. With Java semantics, this will be a bad year for the poor employees.

3.6.2. GDK methods for numbers

The GDK defines all applicable methods from table 3.4 to implement overridable operators for numbers such as `plus`, `minus`, `power`, and so forth. They all work without surprises. In addition, the methods below fulfil their self-describing duty:

```
assert 1 == (-1).abs()
assert 2 == 2.5.toInteger()         // conversion
assert 2 == 2.5 as Integer          // enforced coercion
assert 2 == (int) 2.5               // cast
assert 3 == 2.5f.round()
assert 3.142 ==  Math.PI.round(3)
assert 4 == 4.5f.trunc()
assert 2.718 == Math.E.trunc(3)
assert '2.718'.isNumber()           // String methods
assert 5 == '5'.toInteger()
assert 5 == '5' as Integer
assert 53 == (int) '5'              // gotcha!
assert '6 times' == 6 + ' times'    // Number + String
```

As you can see, there are various conversion possibilities: the `toInteger()` method (also available for `Double`, `Float` and so on), enforced coercion with the `as` operator that calls the `asType(class)` method and the humble cast.

Don't cast strings to numbers!

In Groovy, you can cast a string of length one directly to a char. But `char` and `int` are essentially the same thing on the Java platform. This leads to the gotcha where `'5'` is cast to its unicode value `53`. Instead, use the the type conversion methods.

More interestingly, the GDK also defines the methods `times`, `upto`, `downto` , and `step`. They all take a closure argument. Listing 3.9 shows these methods in action: `times` is just for repetition, `upto` is for walking a sequence of increasing numbers, `downto` is for decreasing numbers, and `step` is the general version that walks until the end value by successively adding a step width.

Listing 3.11. GDK methods on numbers

```
def store = ''
10.times{                       // #1 Repetition
    store += 'x'
}
assert store == 'xxxxxxxxxx'

store = ''
1.upto(5) { number ->           // #2 Walking up with loop variable
    store += number
}
assert store == '12345'

store = ''
2.downto(-2) { number ->        // #3 Walking down
    store += number + ' '
}
assert store == '2 1 0 -1 -2 '

store = ''
0.step(0.5, 0.1 ){ number ->    // #4 Walking with step width
    store += number + ' '
}
assert store == '0 0.1 0.2 0.3 0.4 '
```

Calling methods on numbers can feel unfamiliar at first when you come from

Java. Just remember that numbers are objects and you can treat them as such.

As we've seen, numbers in Groovy work in a natural way and protect you against the most common errors with floating-point arithmetic. In most cases, there is no need to remember all details of coercion. When the need arises, this section may serve as a reference.

The strategy of making objects available in unexpected places starts to become an ongoing theme. You have seen it with numbers, and section 4.1 will show the same principle applied to ranges.

3.7. Summary

Contrary to popular belief, Groovy gives you the same type safety as Java, albeit at runtime instead of Java's mix of compile-time and runtime. This approach is a prerequisite to enable the awesome power of dynamic language features such as pretended methods, flexible bindings for scripts, templates and closures, and all the other metaprogramming goodness that we will explore in the course of this book.

Making common activities more convenient is one of Groovy's main promises. Consequently, Groovy promotes even the simple datatypes to first-class objects and implements operators as method calls to make the benefits of object orientation ubiquitously available.

Developer convenience is further enhanced by allowing a variety of means for string literal declarations, whether through flexible GString declarations or with the slashy syntax for situations where extra escaping is undesirable, such as regular expression patterns. GStrings contribute to another of Groovy's central pillars: concise and expressive code. This allows the reader a clearer insight into the runtime string value, without having to wade through reams of string concatenation or switch between format strings and the values replaced in them.

Regular expressions are well represented in Groovy, again confirming its comfortable place among other top of stack languages. Utilizing regular expressions is an everyday exercise, and a language that treated them as second-class citizens would be severely hampered. Groovy effortlessly combines Java's libraries with language support, retaining the regular expression dialect familiar to Java programmers with the ease of use found in scripting.

The Groovy way of treating numbers with respect to type conversion and precision handling leads to intuitive usage, even for non-programmers. This becomes particularly important when Groovy scripts are used for smart configurations of larger systems where business users may provide formulas--for example, to define share-valuation details.

Strings, regular expressions, and numbers alike profit from numerous methods that the GDK introduces on top of the JDK. A clear pattern has emerged already--Groovy is a language designed for the ease of those developing in it, concentrating on making repetitive tasks as simple as they can be without sacrificing the power of the Java platform.

You'll soon see that this focus on ease of use extends far beyond the simple types that Java developers are used to having built-in language support for. The Groovy designers are well aware of other concepts that are rarely far from a programmer's mind. The next chapter shows how intuitive operators, enhanced literals, and extra GDK methods are also available with Groovy's collective data types: ranges, lists, and maps.

4 The collective Groovy datatypes

- Working with ranges
- Workings with arrays and lists
- Working with maps

The intuitive mind is a sacred gift and the rational mind is a faithful servant. We have created a society that honors the servant and has forgotten the gift.
-- Albert Einstein

The nice thing about computers is that they never get tired of repeatedly doing the same task. This is probably the single most important quality that justifies letting them take part in our life. Searching through countless files or web pages, downloading emails every 10 minutes, looking up all values of a stock symbol for the last quarter to paint a nice graph--these are only a few examples where the computer needs to repeatedly process an item of a data collection. It is no wonder that a great deal of programming work is about collections.

Because collections are so prominent in programming, Groovy alleviates the tedium of using them by directly supporting datatypes of a collective nature: ranges, lists, and maps. In accordance with what you have seen of the simple datatypes, Groovy's support for collective datatypes encompasses new lightweight means for literal declaration, specialized operators, and numerous GDK enhancements.

The notation that Groovy uses to set its collective datatypes into action will be new to Java programmers, but as you will see, it is easy to understand and remember. You will pick it up so quickly that you will hardly be able to imagine there was a time when you were new to the concept.

Despite the new notation possibilities, lists and maps have the exact same semantics as in Java. This situation is slightly different for ranges, because they don't have a direct equivalent in Java. So let's start our tour with that topic.

4.1. Working with ranges

Think about how often you've written a loop like this:

```
for (int i=0; i< upperBound; i++){
    // do something with i
}
```

Most of us have done this thousands of times. It is so common that we hardly ever think about it. Take the opportunity to do it now. Does the code tell you what it does or how it does it?

After careful inspection of the variable, the conditional, and the incrementation, we see that it's an iteration starting at zero and not reaching the upper bound, assuming there are no side effects on `i` in the loop body. We have to go through the description of *how* the code works to find out *what* it does.

Next, consider how often you've written a conditional such as this:

```
if (x >= 0 && x <= upperBound) {
    // do something with x
}
```

The same thing applies here: We have to inspect *how* the code works in order to understand *what* it does. Variable `x` must be between zero and an upper bound for further processing. It's easy to overlook that the upper bound is now inclusive.

Now, we're not saying that we make mistakes using this syntax on a regular basis. We're not saying that we can't get used to (or indeed haven't gotten used to) the C-style `for` loop, as countless programmers have over the years. What we're saying is that it's harder than it needs to be; and, more important, it's *less expressive* than it could be. Can you understand it? Absolutely. Then again, you could understand this chapter if it were written entirely in capital letters--that doesn't make it a good idea, though.

Groovy allows you to reveal the meaning of such code pieces by providing the concept of a *range*. A range has a left bound and a right bound. You can do *something* for *each* element of a range, effectively iterating through it. You can determine whether a candidate element falls inside a range. In other words, a range is an interval plus a strategy for how to move through it.

By introducing the concept of ranges, Groovy extends your means of expressing your intentions in the code.

We will show how to specify ranges, how the fact that they are objects makes them ubiquitously applicable, how to use custom objects as bounds, and how they're typically used in the GDK.

4.1.1. Specifying ranges

Ranges are specified using the double dot `..` range operator between the left and the right bound. This operator has a low precedence, so you often need to enclose the declaration in parentheses. Ranges can also be declared using their respective constructors.

The `..<` range operator specifies a half-exclusive range--that is, the value on the right is not part of the range:

```
left..right
left..<right
```

Ranges usually have a lower left bound and a higher right bound. When this is switched, we call it a *reverse* range. Ranges can also be any combination of the types we've described. Listing 4.1 shows these combinations and how ranges can have bounds other than integers, such as dates and strings. Groovy supports ranges at the language level with the special *for-in-range* loop.

Listing 4.1. Range declarations

```
assert (0..10).contains(0)              //|#1 Inclusive range
assert (0..10).contains(5)              //|#1
assert (0..10).contains(10)             //|#1
                                        //|#1
assert (0..10).contains(-1) == false    //|#1
assert (0..10).contains(11) == false    //|#1

assert (0..<10).contains(9)             //|#2 Half-exclusive range
assert (0..<10).contains(10) == false   //|#2


def a = 0..10                           //|#3 Reference to range
assert a instanceof Range               //|#3
assert a.contains(5)                    //|#3

a = new IntRange(0,10)                  //|#4 Explicit construction
assert a.contains(5)                    //|#4

assert (0.0..1.0).contains(0.5) == false      // #5 Containment
assert (0.0..1.0).containsWithinBounds(0.5)   // #6 Bounds

def today     = new Date()
def yesterday = today-1
assert (yesterday..today).size() == 2   // #7 Date range
assert ('a'..'c').contains('b')         // #8 String range

def store = ''
for (element in 5..9) {                 // #9 for in range loop
    store += element
}
assert store == '56789'

store = ''
for (element in 9..5) {                 // #10 Reverse loop
    store += element
}
assert store == '98765'

store = ''
(9..<5).each { element ->              // #11 Reverse range, each
    store += element
}
assert store == '9876'
```

Note that we assign a range to a variable in ❶ . In other words, the variable holds a reference to an object of type `groovy.lang.Range`.

Date objects can be used in ranges, as in ❷ , because the GDK adds the `previous` and `next` methods to `Date`, which increase or decrease the date by one day.

By the Way The GDK also adds *minus* and *plus* operators to `java.util.Date`, which increase or decrease the date by the given number of days.

The `String` methods `previous` and `next` are added by the GDK to make strings usable for ranges, as in ❸ . The last character in the string is incremented/decremented, and over-/underflow is handled by appending a new character or deleting the last character.

We can walk through a range with the `each` method, which presents the current value to the given closure with each step, as shown in ❹ . If the range is reversed, we will walk through the range backward. If the range is half-exclusive, the sequence stops immediately before reaching the right bound.

4.1.2. Ranges are objects

Because every range is an object, you can pass a range around and call its methods. The most prominent methods are `each`, which executes a specified closure for each element in the range, and `contains`, which specifies whether a value is part of the range such that it will be hit when walking over the range. The `containsWithinBounds` method tells whether the argument lies in the interval between the bounds. Note that a value that lies in the range *interval* is still not considered as being *contained* in the range if the iteration logic of never reaches it.

Being first-class objects, ranges can also participate in the game of operator overriding (see section 3.3) by providing an implementation of the `isCase` method, with the same meaning as `contains`. That way, you can use ranges with the `in` operator, as `grep` filters and as `switch` cases. This is shown in listing 4.2.

Listing 4.2. Ranges are objects

```
def result = ''                                       //|#1 Range for iteration
(5..9).each { element ->                              //|#1
    result += element                                 //|#1
}                                                     //|#1
assert result == '56789'                              //|#1

assert 5 in 0..10                                     //|#2 Range for classification
assert (0..10).isCase(5)                              //|#2
                                                      //|#2
def age = 36                                          //|#2
switch(age){                                          //|#2
    case 16..20 : insuranceRate = 0.05 ; break        //|#2
    case 21..50 : insuranceRate = 0.06 ; break        //|#2
    case 51..65 : insuranceRate = 0.07 ; break        //|#2
    default: throw new IllegalArgumentException()     //|#2
}                                                     //|#2
assert insuranceRate == 0.06                          //|#2

def ages = [20, 36, 42, 56]                           //|#3 Range as filter
def midage = 21..50                                   //|#3
assert ages.grep(midage) == [36, 42]                  //|#3
```

Using a range in conjunction with the `grep` method ❷ is a good example of how useful it is to be able to pass around range objects: The `midage` range gets passed as an argument to the `grep` method.

Classification through ranges as shown at ❶ is common in the business world: interest rates for different ranges of allocated assets, transaction fees based on volume ranges, and salary bonuses based on ranges of business done. Although technical people prefer using functions, business people tend to use ranges. When you're modeling the business world in software, classification by ranges can be very handy.

4.1.3. Ranges in action

Listing 4.1 made use of date and string ranges. In fact, any datatype can be used with ranges, provided that both of the following are true:

- The type implements `next` and `previous`; that is, it overrides the ++ and operators.
- The type implements `java.lang.Comparable`; that is, it implements `compareTo`, effectively overriding the <=> *spaceship* operator.

As an example, listing 4.3 implements a `Weekday` class that represents a day of the week. Each `Weekday` is constructed with an index that represents the `'Sun'` through `'Sat'` day of the week but we do not normalize the index to fall in between `0` and `6`; this allows weekday ranges to span multiple weeks. A little list maps indexes to weekday name abbreviations.

We implement `next` and `previous` to return the respective new `Weekday` object. `compareTo` simply compares the indexes.

With this preparation, we can construct a range of working days and iterate through it, reporting the work done until we finally reach the well-deserved weekend. Oh, and our boss wants to assess the weekly work report. An assertion does this on his behalf.

Listing 4.3. Custom ranges: weekdays

```
class Weekday implements Comparable {
    static final DAYS = [
        'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'
    ]
    private int idx = 0

    Weekday(index)     { idx = index }         // #1 Allow all values

    Weekday next()     { new Weekday(idx+1) }             // #2 Range bound methods
    Weekday previous() { new Weekday(idx-1) }             // #2
    int compareTo(Object other) { this.idx <=> other.idx } // #2

    String toString() {
        def index = idx % DAYS.size()
        while (index < 0) index += DAYS.size()
        DAYS[index]
    }
}

def mon = new Weekday(1)
def fri = new Weekday(5)

def report = ''
for (day in mon..fri) {                 // #3 Working through the week
    report += day.toString() + ' '
}
assert report == 'Mon Tue Wed Thu Fri '

7.times { mon++ }
def diary = 'no work on '
for (day in ++fri ..< mon) {            // #4 Enjoying the weekend
    diary += day.toString() + ' '
}
assert diary == 'no work on Sat Sun '
```

This code can be placed inside one script file, even though it contains both a class declaration and script code. The `Weekday` class is like an inner class to the script.

Using `7.times{mon++}` to let `mon` point to next week's Monday may come as a little surprise. But we need to do this or ❹ would work backwards through the week again, which we rather want to avoid.

Custom ranges are helpful in cases where you can't enumerate all cases, or need special behavior like `Weekday` ranges spanning over multiple weeks. For simpler cases Groovy allows you to use `enum`s as range boundaries. 4.23 uses this feature to warn us like a mother.

Listing 4.4. Enum as range boundary

```
enum Month {
    Jan, Feb, Mar, Apr, May, Jun,
    Jul, Aug, Sep, Oct, Nov, Dec
}
def noClams   = Month.May .. Month.Aug
def thisMonth = Month.Aug

boolean iWarnedYou  = false
if (thisMonth in noClams) {                 // #1 'in' operator
    println "Don't eat clams this month,"
    println "it has no 'r' in its name!"
    iWarnedYou = true
}
assert iWarnedYou
```

Compared to the Java alternatives, ranges have proven to be a flexible solution. *For* loops and conditionals are not objects, cannot be reused, and cannot be passed around, but ranges can. Ranges let you focus on *what* the code does, rather than *how* it does it. This is a pure declaration of your intent, as opposed to fiddling with indexes and boundary conditions.

Using custom ranges is the next step forward. Actively look through your code for possible applications. Ranges slumber everywhere, and bringing them to life can significantly improve the expressiveness of your code. With a bit of practice, you may find ranges in very unexpected places. This is a sure sign that new language concepts can change your perception of the world.

We'll see ranges come up again while we look at the subscript operator on lists, the built-in datatype that we are going to cover next.

4.2. Working with lists

In a recent Java project, we had to write a method that takes a Java array and adds an element to it. This seemed like a trivial task, but we forgot how awkward Java programming could be. (We're spoiled from too much Groovy programming.) Java arrays cannot be changed in length, so you cannot add elements easily. One way is to convert the array to a `java.util.List`, add the element, and convert back. A second way is to construct a new array of `size+1`, copy the old values over, and set the new element to the last index position. Either takes some lines of code.

But Java arrays also have their benefits in terms of language support. They work with the subscript operator to easily retrieve elements of an array by index like `value = myarray[index]`, or to store elements at an index position with `myarray[index] = newElement`.

We'll see how Groovy lists give you the best of both approaches, extending the features for smart operator implementations, method overloading, and using lists as Booleans. In Groovy you don't have to care whether a method returns an array (like `Class.getMethods()`) or a list (like `ProcessBuilder.command()`) as you can work with both of them in the same way. With Groovy lists, you will also discover new ways of leveraging the Java Collections API.

4.2.1. Specifying lists

Listing 4.4 shows various ways of specifying lists. The primary way is with square brackets around a sequence of items, delimited with commas:

```
[item, item, item]
```

The sequence can be empty to declare an empty list. Lists are by default of type `java.util.ArrayList` and can also be declared explicitly by calling the respective constructor. The resulting list can still be used with the subscript operator. In fact, this works with any type of list, as we show here with type `java.util.LinkedList`.

Lists can be created and initialized at the same time by calling `toList` on ranges.

Listing 4.5. Specifying lists

```
List myList = [1, 2, 3]

assert myList.size() == 3
assert myList[0]     == 1
assert myList instanceof ArrayList

List emptyList = []
assert emptyList.size() == 0
```

```
List longList = (0..1000).toList()
assert longList[555] == 555

List explicitList = new ArrayList()
explicitList.addAll(myList)             // #1 Fill from myList
assert explicitList.size() == 3
explicitList[0] = 10
assert explicitList[0] == 10

explicitList = new LinkedList(myList)   // #1
assert explicitList.size() == 3
explicitList[0] = 10
assert explicitList[0] == 10

assert args instanceof String[]         // #2 Command-line args
assert args.size() == 0                 // #3 Array as list

List flat = [0, *myList, 4]             // #4 Spread
assert flat == [0, 1, 2, 3, 4]
```

We use the `addAll(`*`Collection`*`)` method from `java.util.List` at ❶ to easily fill the lists. As an alternative, the collection to fill from can be passed right into the constructor, as we have done with `LinkedList`.

We also see how even arrays can be treated like lists in ❸ , where we call list's `size()` method on the string array of command line arguments. The GDK extends all arrays, collection objects, and strings with a `toList` method that returns a newly generated list of the contained elements. Strings are treated as lists of characters.

Finally, literal declarations of lists can also include the *content* of other lists by adding those prefixed with the `*` *spread* ❹ operator. This operator *spreads* its content into the list such that the result contains each of the elements of the original list, rather than the list object itself.

4.2.2. Using list operators

Lists implement some of the overridable operators that you saw in section 3.3. Listing 4.4 contained two of them: the `getAt` and `putAt` methods to implement the subscript operator. But this was a simple use that works with a mere index argument. There's much more to the list operators than that.

The subscript operator

The GDK overloads the `getAt` method with range and collection arguments to access a range or a collection of indexes. This is demonstrated in Listing 4.5.

The same strategy is applied to `putAt`, which is overloaded with a *Range* argument, assigning a list of values to a whole sublist.

Listing 4.6. Accessing parts of a list with the overloaded subscript operator

```
def myList = ['a','b','c','d','e','f']

assert myList[0..2]  == ['a','b','c']       //#1 getAt(Range)
assert myList[0,2,4] == ['a','c','e']       //#2 getAt(collection of indexes)

myList[0..2] = ['x','y','z']                //#3 putAt(Range)
assert myList == ['x','y','z','d','e','f']

myList[3..5] = []                           //#4 Removing a sublist
assert myList == ['x','y','z']

myList[1..1] = [0, 1, 2]                    //#5 Inserting a sublist
assert myList == ['x', 0, 1, 2, 'z']
```

Subscript assignments with ranges do not need to be of identical size. When the assigned list of values is smaller than the range or even empty, the list shrinks, as shown at ❹ . When the assigned list of values is bigger, the list grows, as in ❺ .

Using a range within a subscript assignment is a convenience feature to access Java's excellent sublist support for lists. See the Javadoc for `java.util.List#sublist` for more information.

In addition to positive index values, lists can also be subscripted with negative indexes that count from the end of the list backward. Figure 4.1 show how positive and negative indexes map to an example list `[0,1,2,3,4]`.

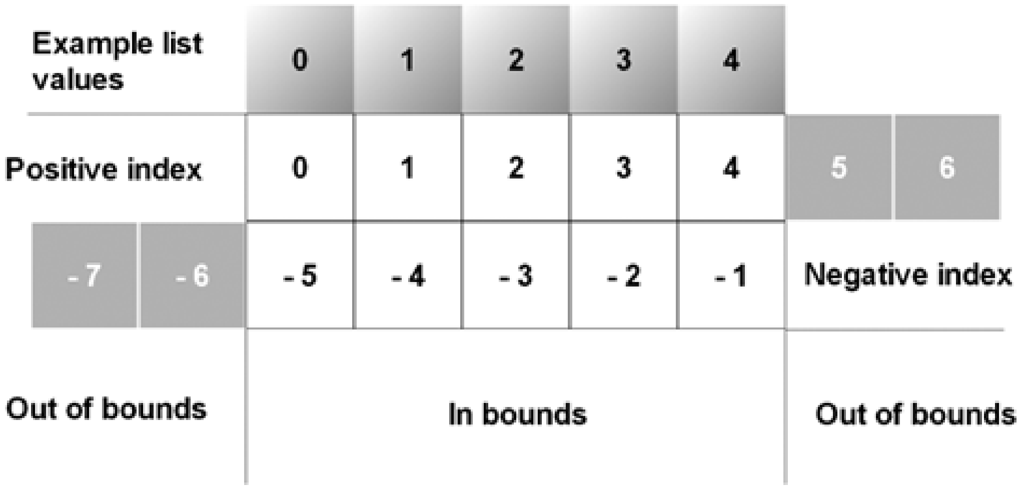


Figure 4.1. Positive and negative indexes of a list of length five, with “in bounds” and “out of bounds” classification for indexes

Consequently, you get the last entry of a non-empty list with `list[-1]` and the next-to-last with `list[-2]`. Negative indexes can also be used in ranges, so `list[-3..-1]` gives you the last three entries. When using a *reversed* range, the resulting list is reversed as well, so `list[4..0]` is `[4,3,2,1,0]`. In this case, the result is a new list object rather than a *sublist* in the sense of the JDK. Even mixtures of positive and negative indexes are possible, such as `list[1..-2]` to cut away the first entry and the last entry.

> Ranges in 's subscript operator are `IntRange`s. Half-exclusive `IntRange`s are mapped to inclusive ones at range construction time, before the subscript operator e comes into play. This can lead to surprises when mixing positive left and negative ri bounds with exclusiveness; for example, `IntRange (0..<-2)` gets mapped to `(0..-1)`, such that `list[0..<-2]` is effectively `list[0..-1]`.
>
> Although this is stable and works predictably, it may be confusing for the readers of your code, who may expect it to work like `list[0..-3]`. We suggest that you avoid situations like this for the sake of clarity.

Adding and removing items

Although the subscript operator can be used to change any individual element of a list, there are also operators available to change the contents of the list in a more drastic way. They are `plus(Object)`, `plus(Collection)`, `leftShift(Object)`, `minus(Collection)`, and `multiply`. Listing 4.6 shows them in action. The `plus` method is overloaded to distinguish between adding an element and adding all elements of a collection. The `minus` method only works with collection parameters.

Listing 4.7. List operators involved in adding and removing items

```
List myList = []

myList += 'a'                               //#1 plus(Object)
assert myList == ['a']

myList += ['b','c']                         //#2 plus(Collection)
assert myList == ['a','b','c']

myList = []
myList <<  'a' << 'b'                       //#3 leftShift is like append
assert myList == ['a','b']

assert myList - ['b'] == ['a']              //#4

assert myList * 2 == ['a','b','a','b']      //#5
```

While we're talking about operators, it's worth noting that we have used the `==` operator on lists, happily assuming that it does what we expect. Now we see how it works: The `equals` method on lists tests that two collections have equal elements. See the Javadoc of `java.util.List#equals` for details.

Control structures

Groovy lists are more than flexible storage places. They also play a major role in organizing the execution flow of Groovy programs. Listing 4.7 shows the use of lists with Groovy's `if`, `in` `switch`, `grep` and `for`.

Listing 4.8. Lists taking part in control structures

```
List myList = ['a', 'b', 'c']

assert myList.isCase('a')
assert 'b' in myList

def candidate = 'c'
switch(candidate){
    case myList : assert true; break        //#1 Classify by containment
    default     : assert false
}

assert ['x','a','z'].grep(myList) == ['a']  //#2 Intersection filter

myList = []
if (myList) assert false                    //#3 Empty lists are false

// Lists can be iterated with a 'for' loop
def expr = ''
for (i in [1,'*',5]){                       //#4 for in Collection
    expr += i
}
assert expr == '1*5'
```

In ❶ and ❷ , you see the trick that you already know from patterns and ranges: implementing `isCase` and getting a `grep` filter, `in` tests and a `switch` classification for free.

❸ is a little surprising. Inside a Boolean test, empty lists evaluate to `false`.

❹ shows looping over lists or other collections and also demonstrates that lists can contain mixtures of types.

4.2.3. Using list methods

There are so many useful methods on the `List` type that we cannot provide an example for all of them in the language description. The large number of methods comes from the fact that the Java interface `java.util.List` is already fairly wide (25 methods in JDK 1.5).

Furthermore, the GDK adds methods to the `List` interface, to the `Collection` interface, and to `Object`. Therefore, many methods are available on the `List` type, including all methods of `Collection` and `Object`.

Appendix C has the complete overview of all methods added to `List` by the GDK. The Javadoc of `java.util.List` has the complete list of its JDK methods.

While working with lists in Groovy, there is no need to be aware of whether a method stems from the JDK or the GDK, or whether it is defined in the `List` or `Collection` interface. However, for the purpose of describing the Groovy `List` datatype, we're going to cover all the GDK methods on lists and collections, but not all the combinations of overloaded methods, and not methods we've already covered in earlier examples. We'll only provide examples of the JDK methods that we consider particularly important.

Manipulating list content

A first set of methods is presented in Listing 4.8. It deals with changing the content of the list by adding and removing elements; combining lists in various ways; sorting, reversing, and flattening nested lists; and creating new lists from existing ones.

Listing 4.9. Methods to manipulate list content

```
assert [1, [2, 3]].flatten() == [1, 2, 3]
assert [1, 2, 3].intersect([4, 3, 1]) == [3, 1]
assert [1, 2, 3].disjoint([4, 5, 6])

List list = [1, 2]
list.push 3
assert list == [1, 2, 3]
popped = list.pop()                          //#1 List as Stack
assert popped == 3
assert list == [1, 2]

assert [1, 2].reverse() == [2, 1]

assert [3, 1, 2].sort() == [1, 2, 3]

def kings = ['Dierk', 'Paul']
kings = kings.sort {item -> item.size() }    //#2 Sort by size
assert kings == ['Paul', 'Dierk']

kings.sort {a, b -> b[0] <=> a[0] }          //#3 Reverse sort by first char
assert kings == ['Paul', 'Dierk']

list = ['a', 'b', 'c']
list.remove(2)                               //#4 Remove by index
assert list == ['a', 'b']
list.remove('b')                             //#5 Remove by value
assert list == ['a', 'b'] - 'b'

list = ['a', 'b', 'b', 'c']
list.removeAll(['b', 'c'])                   //#6 Remove all
assert list == ['a', 'b', 'b', 'c'] - ['b', 'c']
```

```
def doubled = [1, 2, 3].collect {item ->    //#7 Converting
    item * 2
}
assert doubled == [2, 4, 6]

def squares = [0, 1, 4]
[3, 4, 5].collect(squares) {item -> item * item }
assert squares == [0, 1, 4, 9, 16, 25]

def odd = [1, 2, 3].findAll {item ->        //#8 Filtering
    item % 2 == 1
}
assert odd == [1, 3]

assert 1 == [1, 2, 1].find { it % 2 == 1 }
```

List elements can be of any type, including other nested lists. This can be used to implement lists of lists, the Groovy equivalent of multidimensional arrays in Java. For nested lists, the `flatten` method provides a flat view of all elements.

An intersection of lists contains all elements that appear in both lists. Collections can also be checked for being `disjoint`--that is, whether their intersection is empty.

Lists can be used like *stacks*, with the usual stack behavior on push (or `<<`) and `pop`, as in ❶ .

When list elements are `Comparable`, there is a natural sort order which is used by a simple call to `sort()`. Alternatively, the comparison logic of the sort can be specified as a closure, as in ❷ and ❸ . In the first example, we pass one argument in the comparing *closure*. This allows us to specify which item feature should be used for comparison. In our case that is the `size()` of each string. The second example uses a more general comparator that takes both items and decides to compare the first character of the second argument with the first character of the first argument, effectively doing a reverse sort that way. Note that although the sort method returns the modified list, we can also omit the assignment, since the list itself is also modified in place. In this case Groovy is aligned with an oddity of the JDK.

Elements can be removed by index, as in ❹ , or by value, as in ❺ . We can also remove all the elements that appear as values in the second list. These removal methods are the only ones in the listing that are available in the JDK. Note that while the JDK remove methods modify the list in place, Groovy's minus operator returns a modified copy.

The `collect` method, seen in ❼ , returns a new list containing the result of applying the specified closure to each element in the original list. In the example, we use it to retrieve a new list where each entry of the original list is multiplied by two. A second variant of `collect` adds the collected calculations to an existing collection, a list of squares in this case.

With `findAll`, used in ❽ , we retrieve a list of all items for which the closure evaluates to `true`. In the example, we use the modulo operator to find all odd numbers. The method has a `find` companion that returns the first element in the collection that satisfies the given closure.

Two issues related to changing an existing list are removing duplicates and removing `null` values. One way to remove duplicate entries is to convert the list to a datatype that is free of duplicates: a `Set`. This can be achieved by calling a `Set`'s constructor with that list as an argument.

```
def x = [1, 1, 1]
assert [1] == new HashSet(x).toList()
assert [1] == x.unique()
assert [1] == [1, '1'].unique { item -> item.toInteger() }
```

If you don't want to create a new collection but do want to keep working on your cleaned list, you can use the `unique` method, which ensures that the sequence of entries is not changed by this operation. The method takes a closure as an optional parameter that allows to specify what “uniqueness” should mean in this context.

Removing `null` from a list can be done by keeping all non- `null`s--for example, with the `findAll` methods that you have seen previously:

```
List x = [1, null, null, 2]

assert [1, 2] == x.findAll { it != null }
assert [1, 2] == x.grep { it }

assert [1, 2] == x - [null]

x.removeAll([null])
assert [1, 2] == x
```

You can see there's an even shorter version with `grep`, but in order to understand its mechanics, you need more knowledge about closures (chapter 5) and “ The Groovy truth” (chapter 6). Just take it for granted until then. Of course, it is also possibly to simply use the various JDK `remove` methods or the GDK `minus` operator.

Accessing list content

Lists have methods to query their elements for certain properties, iterate through them, and retrieve accumulated results.

Query methods include a `count` of given elements in the list, `min` and `max`, a `find` method that finds the first element that satisfies a closure, and methods to determine whether `every` or `any` element in the list satisfies a closure.

Iteration can be achieved as usual, stepping forward with `each` or backward with `eachReverse`.

Cumulative methods come in simple and sophisticated versions. The `join` method is simple: It returns all elements as a string, using a given delimiter. The `inject` method is inspired by Smalltalk. It uses a closure to inject new functionality. That functionality operates on an intermediate result and the current element of the iteration. The first parameter of the `inject` method is the initial value of the intermediate result. Would it be worth using the word "accumulator" here, or would that cause more confusion? (Jon) In listing 4.9, we use this method to sum the elements in a list and then use it a second time to multiply them.

Listing 4.10. List query, iteration, and accumulation

```
def list = [1, 2, 3]

assert list.first()  == 1
assert list.head()   == 1
assert list.tail()   == [2, 3]
assert list.last()   == 3
assert list.count(2) == 1                  //|#1 Querying
assert list.max()    == 3                  //|#1
assert list.min()    == 1                  //|#1
                                           //|#1
def even = list.find { item ->             //|#1
    item % 2 == 0                          //|#1
}                                          //|#1
assert even == 2                           //|#1
                                           //|#1
assert list.every { item -> item < 5 }     //|#1
assert list.any   { item -> item < 2 }     //|#1

def store = ''
list.each { item ->                        //|#2 Iteration
    store += item                          //|#2
}                                          //|#2
assert store == '123'                      //|#2
                                           //|#2
store = ''                                 //|#2
list.reverseEach { item ->                 //|#2
    store += item                          //|#2
}                                          //|#2
assert store == '321'                      //|#2
                                           //|#2
store = ''                                 //|#2
list.eachWithIndex { item, index ->        //|#2
    store += "$index:$item "               //|#2
}                                          //|#2
assert store == '0:1 1:2 2:3 '             //|#2
assert list.join('-') == '1-2-3'            //|#3 Accumulation
                                            //|#3
result = list.inject(0) { clinks, guests -> //|#3
    clinks + guests                         //|#3
}                                           //|#3
assert result == 0 + 1 + 2 + 3              //|#3
assert list.sum() == 6                      //|#3
                                            //|#3
factorial = list.inject(1) { fac, item ->   //|#3
    fac * item                              //|#3
}                                           //|#3
assert factorial == 1 * 1 * 2 * 3           //|#3
```

Understanding and using the `inject` method can be a bit challenging if you're new to the concept. Note that it is exactly parallel to the *iteration* examples, with `store` playing the role of the intermediary result. The benefit is that you do not need to introduce that extra variable to the outer scope of your accumulation, and your closure has no side effects on that scope.

This has already been a long list of methods but there is even more. First, the methods `max`, `min`, and `unique` all come in three flavors: without parameters, with a closure parameter, and with a comparator. The `sum` method also comes with an overload taking a closure to specify how objects should be summed. In the examples below, we use the `it` shortcut that refers to the item that is passed into the closure.

```
def kings = ['Dierk', 'Paul']
assert kings.max { item -> item.size() } == 'Dierk'
assert kings.min { item -> item.size() } == 'Paul'
assert kings.sum { item -> item.size() } == 9
```

We've already mentioned that lists can be casted to arbitrary classes, which results in the appropriate constructor calls. But there are other options: lists--and indeed all other collections-- also support enforced type coercion with the `as` operator by implementing the `asType` method. Groovy supports coercions to types `List`, `Set`, `SortedSet`, `Queue` and `Stack`, plus any subtype of `List` may implement the `asType` method. The code below gives some examples of this in action.

```
Set names = ['Dierk', 'Paul'] as Set
assert names instanceof Set

assert names.toListString() ==~ /[w+, w+]/
assert names.asList() instanceof List

java.awt.Point p = [10, 20]
assert p.x == 10
```

Sometimes it is convenient to partition a collection (list, set, range, and so on) based on some condition. The `split` method does so by returning two lists: a first one that contains all the elements that satisfy the given closure, and a second one for the rest. We use Groovy's parallel assignment feature (see XREF parallel_assignment) to assign the return values to two different variables. In some ways `split` can be regarded as a special version of the more general `groupBy` method. This returns a map that uses a closure's return values as keys with values that collect the list of items that returned that key. We will learn more about maps in XREF map.

```
def list = [0, 3, 2, 1]
def (small, big) = list.split { it < 2 }
assert small == [0, 1]
assert big   == [3, 2]

def group = list.groupBy { it % 2 }
assert group[0] == [0, 2]
assert group[1] == [3, 1]
```

Finally, there are nested lists where a list item is a list itself. A natural example of this is a table, where you could have a list of rows, and each row is a list of values. Groovy adds three methods to support working nested lists: `collectAll` works like `collect` but recurses into nested collections, `transpose` applies the eponymous matrix operation, and `combinations` returns a list of all item combinations of all the nested lists.

```
def table = [
    [0, 1],
    [2, 3]
]
table = table.collectAll { item -> item + 1 }
assert table == [
    [1, 2],
    [3, 4]
]
assert table.transpose() == [
    [1, 3],
    [2, 4]
]
assert table.combinations() == [
    [1, 3], [2, 3], [1, 4], [2, 4]
]
```

The GDK introduces two more convenience methods for lists: `asImmutable` and `asSynchronized`. These methods protect the list from unintended content changes and concurrent access. They become particularly important when dealing with parallel programming that we will encounter in XREF parallel_programming. See these methods' Javadocs for more details on the topic.

4.2.4. Lists in action

After all these artificial examples, you deserve to see a real one. Here it is: We will implement Tony Hoare's Quicksort [45] algorithm in listing 4.10. To make things more interesting, we will do so in a generic way rather than demanding a specific datatype for sorting. We rely on *duck typing*-- as long as something walks like a duck and talks like a duck, we happily treat it as a duck. In this case, this means that as long as we can use the <, =, and > operators with our list items, we treat them as if they were comparable.

Footnote 45. See http://en.wikipedia.org/wiki/Quicksort.

The goal of Quicksort is to be sparse with comparisons. The strategy relies on finding a good *pivot* element in the list that serves to split the list into two sublists: one with all elements smaller than the pivot, the second with all elements bigger than the pivot. Quicksort is then called recursively on the sublists. The rationale behind this is that you never need to compare elements from one list with elements from the other list. If you always find the perfect pivot, which exactly splits your list in half, the algorithm runs with a complexity of n*log(n). In the worst case, you choose a border element every time, and you end up with a complexity of n2. In listing 4.10, we choose the middle element of the list, which is a good choice for the frequent case of preordered sublists.

Listing 4.11. Quicksort with lists

```
def quickSort(list) {
    if (list.size() < 2) return list
    def pivot  = list[list.size().intdiv(2)]
    def left   = list.findAll { item -> item <  pivot }       //|#1 Classify by pivo
    def middle = list.findAll { item -> item == pivot }       //|#1
    def right  = list.findAll { item -> item >  pivot }       //|#1
    return quickSort(left) + middle + quickSort(right)        //#2 Recursive call
}

assert quickSort([])                 == []
assert quickSort([1])                == [1]
assert quickSort([1,2])              == [1,2]
assert quickSort([2,1])              == [1,2]
assert quickSort([3,1,2])            == [1,2,3]
assert quickSort([3,1,2,2])          == [1,2,2,3]
assert quickSort([1.0f,'a',10,null])== [null, 1.0f, 10, 'a'] // #3 Item type mix
assert quickSort('bca')              == 'abc'.toList()        // #4 Non-list type
```

In contrast to the simple description, we actually use three lists in ❶ rather than three. Use this implementation when you don't want to lose items that appear multiple times.

Our duck-typing approach is powerful when it comes to sorting different types. We can sort a list of mixed content types, as at ❷ , or even sort the characters in a string, as shown at ❸ . This is possible because we did not demand any specific type to hold our items. As long as that type implements `size`, `getAt(` *index* `)`, and `findAll`, we are happy to treat it as a *sortable*. Actually, we used duck typing twice: for the items and for the structure.

By the Way The `sort` method that comes with Groovy uses Java's sorting implementation that beats our example in terms of worst-case performance. It guarantees a complexity of n*log(n). However, we win on a different front.

Of course, our implementation could be optimized in various ways. Our goal was to be tidy and flexible, not to be the fastest on the block.

If we had to explain the Quicksort algorithm without the help of Groovy, we would sketch it in pseudocode that looks very similar to listing 4.10. In other words, the Groovy code itself is an ideal description of what it does. Imagine what this can mean to your codebase, when all your code reads like a formal documentation of its purpose!

You have seen lists to be one of Groovy's strongest workhorses. They are always at hand; they are easy to specify in-line, and using them is easy due to the operators supported. The plethora of available methods may be intimidating at first, but that is also the source of lists' power. You are now able to add them to your carriage and let them pull the weight of your code.

The next section about maps will follow the same principles that you have seen for lists: extending the Java collection's capabilities while providing efficient shortcuts.

4.3. Working with maps

Suppose you were about to learn the vocabulary of a new language, and you set out to find the most efficient way of doing so. It would surely be beneficial to focus on those words that appear most often in your texts. So, you would take a collection of your texts and analyze the word frequencies in that text corpus.[46]

Footnote 46. Analyzing word frequencies in a text corpus is a common task in computer linguistics and is used for optimizing computer-based learning, search engines, voice recognition, and machine translation programs.

How does Groovy help you here? For the time being, assume that you can work on a large string. You have numerous ways of splitting this string into words. But how do you count and store the word frequencies? You cannot have a distinct variable for each possible word you encounter. Finding a way of storing frequencies in a list is possible but inconvenient--more suitable for a brain teaser than for good code. Maps come to the rescue.

Some pseudocode to solve the problem could look like this:

```
for each word {
     if (frequency of word is not known)
         frequency[word] = 0
     frequency[word] += 1
}
```

This looks like the list syntax, but with strings as indexes rather than integers. In fact, Groovy maps appear like lists, allowing any arbitrary object to be used for indexing.

In order to describe the map datatype, we show how maps can be specified, what operations and methods are available for maps, some surprisingly convenient features of maps, and, of course, a map-based solution for the word-frequency exercise.

4.3.1. Specifying maps

The specification of maps is analogous to the list specification that you saw in the previous section. Just like lists, maps make use of the subscript operator to retrieve and assign values. The difference is that maps can use any arbitrary type as an argument to the subscript operator, where lists are bound to integer indexes and ranges.

Another key difference between lists and maps is in terms of ordering. Lists are inherently ordered sequences of entries, whereas most map implementations provide no way of iterating over their entries in a particular order. However, there are exceptions to the rule, and Groovy defaults to using `java.util.LinkedHashMap` for map literals so that the order in which the entries are specified in the source code is preserved in the runtime map.

Simple maps are specified with square brackets around a sequence of items, delimited with commas. The difference in syntax between lists and maps is that in a map the items are key-value pairs that are delimited by colons:

```
[key:value, key:value, key:value]
```

In principle, any arbitrary type can be used for keys or values. When using exotic[47] types for keys, you need to obey the rules as outlined in the Javadoc for `java.util.Map`.

Footnote 47. *Exotic* in this sense refers to types whose instances change their `hashCode` during their lifetime. There is also a corner case with GStrings if their values write themselves lazily.

The character sequence `[:]` declares an empty map. By default, map literals create instances of `java.util.LinkedHashMap`, but you can specify a different implementation by calling the respective constructor. The resulting map can still be used with the subscript operator. In fact, this works with any type of map, however it was created, as you can see in listing 4.11 with type `java.util.TreeMap`.

Listing 4.12. Specifying maps

```
def myMap = [a:1, b:2, c:3]

assert myMap instanceof LinkedHashMap
assert myMap.size() == 3
assert myMap['a']   == 1

def emptyMap = [:]
assert emptyMap.size() == 0

def explicitMap = new TreeMap()
explicitMap.putAll(myMap)
assert explicitMap['a'] == 1

def composed     = [x:'y', *:myMap]         // #1 Spread
assert composed == [x:'y', a:1, b:2, c:3]
```

In listing 4.11, we use the `putAll(`*`Map`*`)` method from `java.util.Map` to easily fill the example map. An alternative would have been to pass `myMap` as an argument to `TreeMap`'s constructor. Just like list literals, map declarations can include the content of existing ones by using the `*` spread operator.

For the common case where the keys are strings, you can omit the quotes in map declarations:

```
assert ['a':1] == [a:1]
```

This is only allowed if the key contains no special characters (it needs to follow the rules for valid identifiers), is neither a Groovy keyword nor a complex expression.

This notation is very helpful in the vast majority of all cases but also has a corner case when the content of a local variable is used as a key in a literal declaration. Suppose you have local variable `x` with content `'a'`. Because `[x:1]` is equal to `['x':1]`, how can you make it equal to `['a':1]`? The trick is that you can force Groovy to recognize a symbol as an expression by putting it inside parentheses:

```
def x = 'a'
assert ['x':1] == [x:1]
assert ['a':1] == [(x):1]
```

You won't need this functionality often, as *literal* declarations rarely use symbols (local variables, fields, properties) as keys-- but when you do, forgetting the parentheses is a likely source of errors.

4.3.2. Using map operators

The simplest operations with maps are storing objects in the map with a *key* and retrieving them back using that key. Listing 4.12 these fundamental operations. One option for retrieving is using the subscript operator. As you have probably guessed, this is implemented with map's `getAt` method. A second option is to use the key like a *property* with a simple dot-syntax. You will learn more about properties in chapter 7. A third option is the `get` method, which additionally allows you to pass a default value to be returned if the key is not yet in the map. If no default is given, `null` will be used as the default. If a call to `get(`*`key, default`*`)` returns the default because the key is not found, the *key:default* pair is added to the map.

Listing 4.13. Accessing maps (GDK map methods)

```
def myMap = [a:1, b:2, c:3]

assert myMap['a']       == 1      //|#1 Retrieve existing elements
assert myMap.a          == 1      //|#1
assert myMap.get('a')   == 1      //|#1
assert myMap.get('a',0) == 1      //|#1

assert myMap['d']       == null   //|#2 Attempt to retrieve
assert myMap.d          == null   //|#2 missing elements
assert myMap.get('d')   == null   //|#2

assert myMap.get('d',0) == 0      //|#3 Default value
assert myMap.d          == 0      //|#3

myMap['d'] = 1                    //|#4 Single putAt
assert myMap.d == 1               //|#4
myMap.d = 2                       //|#4
assert myMap.d == 2               //|#4
```

Assignments to maps can be done using the subscript operator or via the *dot-key* syntax. If the key in the *dot-key* syntax contains special characters, it can be put in quotes, like so:

```
def myMap = ['a.b':1]
assert myMap.'a.b' == 1
```

Just writing `myMap.a.b` would not work here--that would be the equivalent of calling `myMap.get('a').get('b')`.

Listing 4.13 shows how information can easily be gleaned from maps, largely using core JDK methods from `java.util.Map`. Using `equals`, `size`, `containsKey`, and `containsValue` is straightforward, as shown in listing 4.13. The methods `keySet` and `values` both return a *set* of keys and values: a collection that is flat like a list but has no duplicate entries and no inherent ordering. Luckily, since Groovy uses a `LinkedHashSet` by default, these collections retain their order. See the Javadoc of `java.util.LinkedHashSet` for details. In order to compare such a set against a list, we have to convert one or the other. When converting the list to a set, we're comparing two sets, which means that ordering is irrelevant. A stronger equality condition that includes ordering applies when we convert the set of values to a list before comparing it to a fixed list.

A map can also be converted into a collection by calling the `entrySet` method, which returns a set of entries. Each entry can then be asked for its `key` and `value` properties.

Listing 4.14. Query methods on maps

```
def myMap = [a:1, b:2, c:3]
def other = [b:2, c:3, a:1]

assert myMap == other                                  //#1 Call to equals

assert myMap.isEmpty()  == false                       //|#2 JDK methods
assert myMap.size()     == 3                           //|#2
assert myMap.containsKey('a')                          //|#2
assert myMap.containsValue(1)                          //|#2
assert myMap.entrySet() instanceof Collection          //|#2

assert myMap.any   {entry -> entry.value > 2  }        //|#3 GDK methods
assert myMap.every {entry -> entry.key   < 'd'}        //|#3
assert myMap.keySet() == ['a','b','c'] as Set          // #4 Lenient set equals
assert myMap.values().toList() == [1, 2, 3]            // #5 Strong list equals
```

The GDK adds two more informational methods to the JDK map type: `any` and every, as in ❸ . They work analogously to the identically named methods for lists: They return a Boolean value to indicate whether *any* or *every* entry in the map satisfies a given closure.

With the information about the map, we can iterate over it in a number of ways: over the entries, or over keys and values separately. Because the sets that are returned from keySet, values and entrySet are collections, we can use them with the *for-in-collection* type loops. Listing 4.14 goes through some of the possible combinations.

Listing 4.15. Iterating over maps (GDK)

```
def myMap = [a:1, b:2, c:3]

def store = ''
myMap.each { entry ->        //|#1 Each entry
    store += entry.key       //|#1
    store += entry.value     //|#1
}                            //|#1
assert store == 'a1b2c3'

store = ''
myMap.each { key, value ->   //|#2 Each key/value pair
    store += key             //|#2
    store += value           //|#2
}                            //|#2
assert store == 'a1b2c3'

store = ''
for (key in myMap.keySet()) { //|#3 For in set
    store += key              //|#3
}                             //|#3
assert store == 'abc'
```

Map's each method uses closures in two ways: Passing one parameter into the closure means that it is an *entry*; passing two parameters means it is a key and a value. The latter is more convenient to work with for common cases.

Finally, the contents of the map can be changed in various ways, as shown in listing 4.15. Removing elements works with the original JDK methods. The GDK introduces the following new capabilities:

- Creating a subMap of all entries with keys from a given collection (the JDK has such a method only for SortedMaps)
- findAll entries in a map that satisfy a given closure
- find one entry that satisfies a given closure, where unlike lists it depends on the map type whether it supports the notion of a *first* entry
- collect in a list whatever a closure returns for each entry, optionally adding to a given

collection

Listing 4.16. Changing the contents of a map and building new objects from it

```
def myMap = [a:1, b:2, c:3]
myMap.clear()
assert myMap.isEmpty()

myMap = [a:1, b:2, c:3]
myMap.remove('a')
assert myMap.size() == 2

assert [a:1] + [b:2] == [a:1, b:2]

myMap = [a:1, b:2, c:3]
def abMap = myMap.subMap(['a', 'b'])
assert abMap.size() == 2

abMap = myMap.findAll   { entry -> entry.value < 3 }
assert abMap.size() == 2
assert abMap.a      == 1

def found = myMap.find  { entry -> entry.value < 2 }
assert found.key   == 'a'
assert found.value == 1

def doubled = myMap.collect { entry -> entry.value *= 2 }
assert doubled instanceof List
assert doubled.every    { item -> item % 2 == 0 }

def addTo = []
myMap.collect(addTo)    { entry -> entry.value *= 2 }
assert doubled instanceof List
assert addTo.every      { item -> item % 2 == 0 }
```

The first two examples (`clear` and `remove`) are from the core JDK; the rest are all GDK methods. The `collect`, `find`, and `findAll` methods act as they would with lists, operating on map entries instead of list elements. The `subMap` method is analogous to `subList`, but it specifies a collection of keys as a filter for the view onto the original map.

In order to assert that the `collect` method works as expected, we recall a trick that we learned about lists: We use the `every` method on the list to make sure that every entry is even. The `collect` method comes with a second version that takes an additional collection parameter. It adds all closure results directly to this collection, avoiding the need to create temporary lists.

From the list of available methods that you have seen for other datatypes, you may miss our dearly beloved `isCase` for use with `grep` and `switch`. Don't we want to classify with maps? Well, we need to be more specific: Do we want to classify by the keys or by the values? Either way, an appropriate `isCase` is available when working on the map's `keySet` or `values`.

We may not have an `isCase` method on maps, but we have an elegant `asType` implementation that allows to use the `as` operator for coercing a map into *any* type you fancy. To make use of this feature, we declare a map where each key represents a method name and the value is a closure that implements that method. In 4.36 the method is `compare` and the implementation compares the absolute values of the arguments. The `as` operator allows us to use this map as an implementation of the `Comparator` type that we can pass to the `sort` method.

Listing 4.17. Maps and enforced coercion *as* some other type

```
def absComp = [
    compare: { a,b -> a.abs() <=> b.abs() }
]
def list = [-3, -1, 2]
list.sort(absComp as Comparator)
assert list == [-1, 2, -3]
```

Note that

- The `as` operator can be used with classes and interfaces alike
- We only need to provide implementations for methods that are effectively called. For example in 4.36 we provide no implementation for the `equals` method, even though the `Comparator` interface would require us to do so.

While enforced type coercion with `as` is an explicit operation, we have already seen the implicit casting of a map in listing 3.10 that allows for conversions like

```
java.awt.Point p = [x:50, y:50]
```

The GDK introduces two more methods for the map datatype: `asImmutable` and `asSynchronized`. These methods protect the map from unintended content changes and concurrent access.

4.3.3. Maps in action

In 4.37, we revisit our initial example of counting word frequencies in a text corpus. The strategy is to use a map with each distinct word serving as a key. The mapped value of that word is its frequency in the text corpus. We go through all words in the text and increase the frequency value of that respective word in the map. We need to make sure that we can increase the value when a word is hit the first time and there is no entry yet in the map. Luckily, the `get(key,default)` method makes this very simple.

We then take all keys, put them in a list, and sort it such that it reflects the order of frequency. Finally, we play with the capabilities of lists, ranges, and strings to print a nice statistic.

The text corpus under analysis is Baloo the Bear's anthem on his attitude toward life.

Listing 4.18. Counting word frequency with maps

```
def textCorpus =
"""
Look for the bare necessities
The simple bare necessities
Forget about your worries and your strife
I mean the bare necessities
Old Mother Nature's recipes
That bring the bare necessities of life
"""

def words = textCorpus.tokenize()
def wordFrequency = [:]
words.each { word ->
    wordFrequency[word] = wordFrequency.get(word,0) + 1     //#1
}
def wordList = wordFrequency.keySet().toList()
wordList.sort { wordFrequency[it] }                         //#2

def statistic = "n"
wordList[-1..-4].each { word ->
    statistic += word.padLeft(12)    + ': '
    statistic += wordFrequency[word] + "n"
}
assert statistic == """
 necessities: 4
        bare: 4
         the: 3
        your: 2
"""
```

❶ The example nicely combines our knowledge of Groovy's datatypes. Counting the word frequency is essentially a one-liner. It's even shorter than the pseudocode that we used to start this section.

❷ Having the `sort` method on the `wordList` accept a closure turns out to be very useful, because it is able to implement its comparison logic on the `wordFrequency` map--on an object totally different from the `wordList`.

Just as an exercise, try to do that in Java, count the lines, and compare the readability of the two solutions.

Lists and maps make a powerful duo. There are whole languages that build on just these two datatypes (such as Perl, with list and hash) and build all other datatypes and even objects upon them.

Their power comes from the complete and mindfully engineered Java Collections Framework. Thanks to Groovy, this power is now right at our fingertips.

So far, we've casually switched back and forth between Groovy and Java collection datatypes. We will throw more light on this interplay in the next section.

4.4. Notes on Groovy collections

The Java Collections API is the basis for all the nice support that Groovy gives you through lists and maps. In fact, Groovy not only uses the same abstractions, it even works on the very same classes that make up the Java Collections API.

This is exceptionally convenient for those who come from Java and already have a good understanding of it. If you haven't, and you are interested in more background information, have a look at your Javadoc starting at `java.util.Collection`.

The JDK also ships with a guide and a tutorial about Java collections. It is located in your JDK's doc folder under *guide/collections*.

One of the typical peculiarities of the Java collections is that you shouldn't try make a *structural* change while you're iterating through it. A structural change is one that adds an entry, removes an entry, or changes the sequence of entries when the collection is sequence-aware. This applies even when iterating through a view onto the collection, such as using `list[range]`.

4.4.1. Understanding concurrent modification

If you fail to meet this constraint, you will see a `ConcurrentModificationException`. For example, you cannot remove all elements from a list by iterating through it and removing the first element at each step:

```
def list = [1, 2, 3, 4]
list.each{ list.remove(0) } // throws ConcurrentModificationException !!
```

Concurrent in this sense does not necessarily mean that a second thread changed the underlying collection. As shown in the example, even a single thread of control can break the “structural stability” constraint.

In this case, the correct solution is to use the `clear` method. The Collections API has lots of such specialized methods. When searching for alternatives, consider `collect`, `addAll`, `removeAll`, `findAll`, and `grep`.

This leads to a second issue: Some methods work on a copy of the collection and return it when finished; other methods work directly on the collection object they were called on (we call this the *receiver* [48] object).

Footnote 48. From the Smalltalk notion of describing method calls on an object as sending a message to the receiver.

4.4.2. Distinguishing between copy and modify semantics

Generally, there is no easy way to anticipate whether a method modifies the receiver or returns a copy. Some languages have naming conventions for this, but Groovy couldn't do so because all Java methods are directly visible in Groovy and Java's method names could not be made compliant to such a convention. But Groovy tries to adapt to Java and follow the patterns visible in the Collections API:

- Methods that modify the receiver typically don't return a collection. Examples: `add`, `addAll`, `remove`, `removeAll`, and `retainAll`. Counter-examples: `sort` and `unique`.

- Methods that return a collection typically don't modify the receiver. Examples: `grep`, `findAll`, `collect`. Counter-examples: `sort` and `unique`.

- Methods that modify the receiver have *imperative* names. They sound like there could be an exclamation mark behind them. (Indeed, this is Ruby's naming convention for such methods.) Examples: `add`, `addAll`, `remove`, `removeAll`, `retainAll`, `sort`. Counter-examples: `collect`, `grep`, `findAll`, which are imperative but do not modify the receiver and return a modified copy.

- The preceding rules can be mapped to operators, by applying them to the names of their method counterparts: `<< leftShift` is imperative and modifies the receiver (on lists, unfortunately not on strings--doing so would break Java's invariant of strings being immutable); `+ plus` and `- minus` are not imperative and return a copy.

These are not clear rules but only heuristics to give you some guidance. Whenever you're in doubt and object identity is important, have a look at the documentation or write a few assertions.

4.5. Summary

This has been a long trip through the landscape of Groovy's datatypes. There were lots of different paths to explore that led to new and interesting places.

We introduced ranges as objects that--as opposed to control structures--have their own time and place of creation,I don't really understand this "time and place of creation" bit (Jon) can be passed to methods as parameters, and can be returned from method calls. This makes them very flexible, and once the concept of a range is available, many uses beyond simple control structures suggest themselves. The most natural example you have seen is extracting a section of a list using a range as the operand to the list's subscript operator.

Lists and maps are more familiar to Java programmers than ranges but have suffered from a lack of language support in Java itself. Groovy recognizes just how often these datatypes are used, gives them special treatment in terms of literal declarations, and of course provides operators and extra methods to make life even easier. The lists and maps used in Groovy are the same ones encountered in Java and come with the same rules and restrictions, although these become less onerous due to some of the additional methods available on the collections.

Throughout our coverage of Groovy's datatypes, you have seen closures used ubiquitously to make functionality available in a simple and unobtrusive manner. In the next chapter, we will demystify the concept, explain the common and some not-so-common applications, and show how you can spice up your own code with closures.