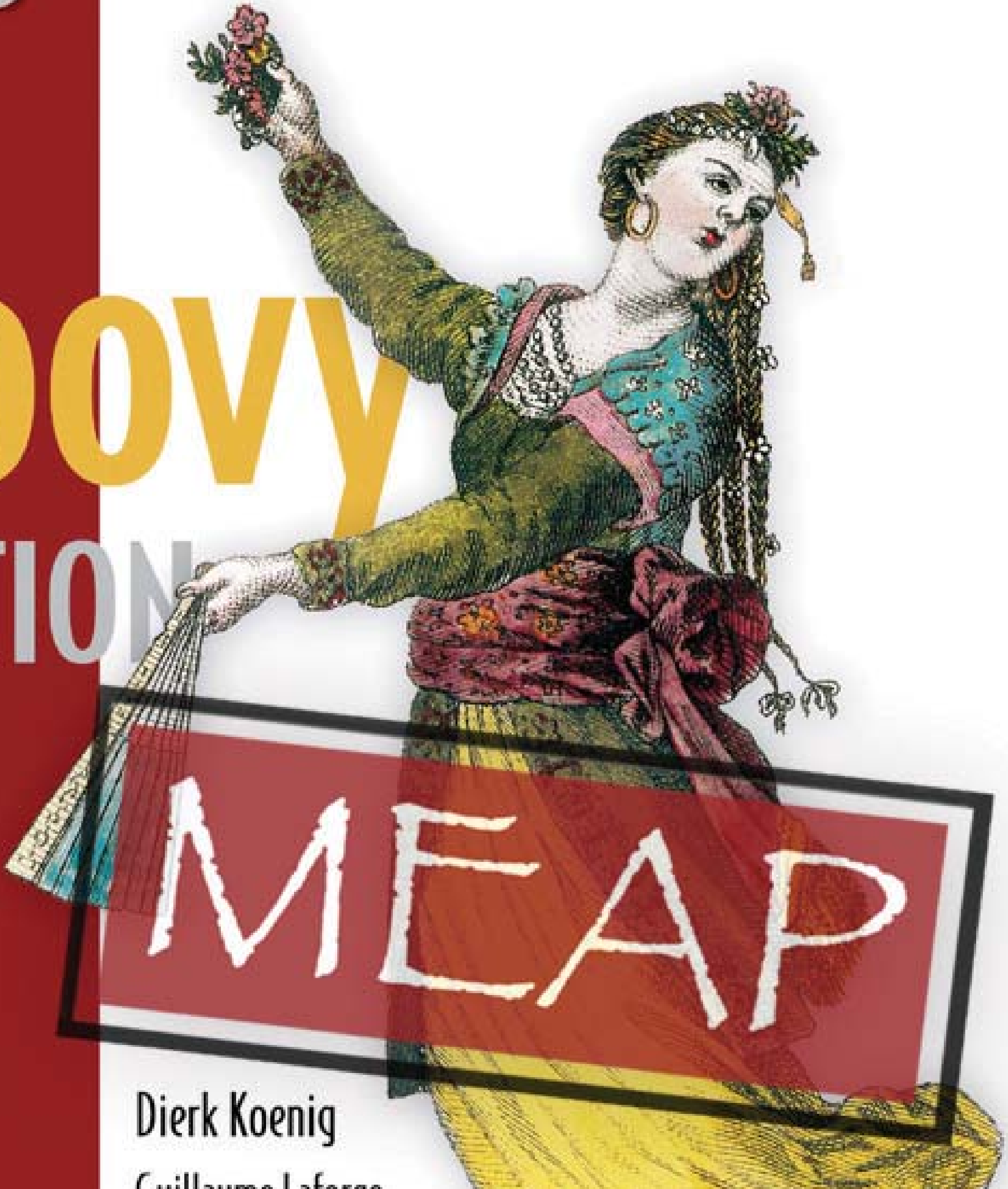
SECOND EDITION

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
Groovy in Action, Second Edition, version 11

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”[1] approach: it obeys the Java object model and always keeps the perspective of a Java programmer. It doesn't force you into any new programming paradigm but offers you those advanced capabilities that you legitimately expect from a “top of stack” language.

Footnote 1 thanks to Mac Liaw for this wording

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

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

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

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

1.1 The Groovy story

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

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

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

Figure 1.1 shows how this unique combination defines Groovy's position in the varied world of languages for the Java platform.[2] We don't want to offend anyone by specifying exactly where we believe any particular other language might fit in the figure, but we're confident of Groovy's position.

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

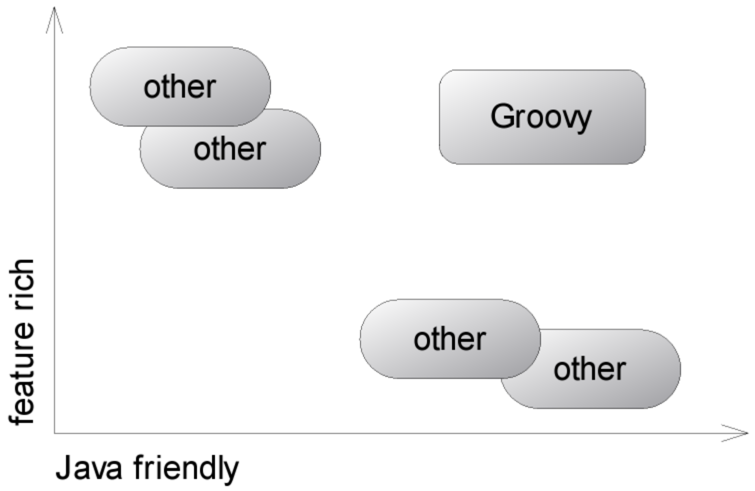


Figure 1.1 The landscape of JVM-based languages. Groovy is feature rich and Java friendly--it excels at both sides instead of sacrificing one for the sake of the other.

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

Some languages may offer more advanced features than Groovy. Not so many languages may claim to fit equally well to the Java language. None can currently touch Groovy when you consider both aspects together: Nothing provides a better combination of Java friendliness and a complete feature set.

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

1.1.1 What is Groovy?

Groovy is an optionally typed, dynamic language for the Java platform with many features that are inspired by languages like Python, Ruby, and Smalltalk, making them available to Java developers using a Java-like syntax. Unlike other alternative languages, it is designed as a *companion*, not a replacement for Java.

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

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

Does this make Groovy just a layer of syntactic sugar? Not at all. Although everything you do in Groovy *could* be done in Java, it would be madness to write the Java code required to work Groovy's magic. Groovy performs a lot of work behind the scenes to achieve its agility and dynamic nature. As you read this book, try to think every so often about what would be required to mimic the effects of Groovy using Java. Many of the Groovy features that seem extraordinary at first--encapsulating logic in objects in a natural way, building hierarchies with barely any code other than what is *absolutely* required to compute the data, expressing database queries in the normal application language before they are translated into SQL, manipulating the runtime behavior of individual objects after they have been created--all of these are tasks that Java wasn't designed for.

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

1.1.2 Playing nicely with Java: seamless integration

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

SEAMLESS INTEGRATION

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

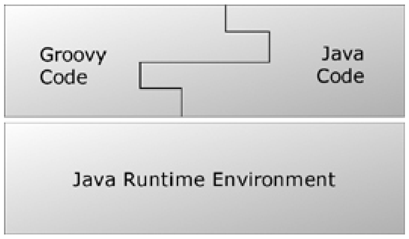


Figure 1.2 Groovy and Java join together in a tongue-and-groove fashion.

Consequently, calling Java from Groovy is a nonissue. When developing in Groovy, you end up doing this all the time without noticing. Every Groovy type is a subtype of `java.lang.Object`. Every Groovy object is an instance of a type in the normal way. A Groovy date *is* a `java.util.Date`. You can call all methods on it that you know are available for a `Date` and you can pass it as an argument to any method that expects a `Date`.

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

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

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

You can then call methods on the instance, pass the reference as an argument to methods, and so forth. The JVM is blissfully unaware that the code was written in

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

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

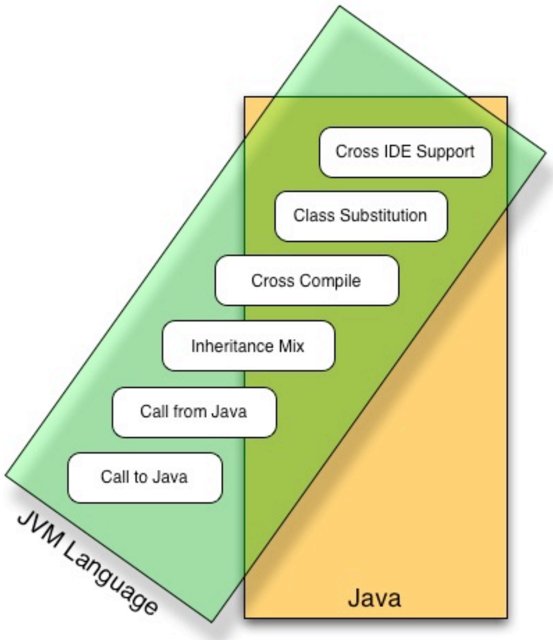


Figure 1.3 The integration ladder shows increasing cross-language support from simple calls for interoperability up to seamless tool integration.

One step up on the integration ladder and we meet the issue of references. A Groovy class may reference a Java class (that goes without saying) and a Java class may reference a Groovy class, as we have seen above. We can even have circular references and `groovyc` compiles them all transparently. Even better, the leading IDEs provide cross-language compile, navigation, and refactoring such that you hardly ever need to care about the project build setup. You are free to choose Java or Groovy when implementing any class for that matter. Such a tight build-time integration is a challenge for every other language.

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

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

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

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

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

SYNTAX ALIGNMENT

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

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

today = new Date()         // Groovy

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

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

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

The Groovy solution is short, precise, and more compact than regular Java. Groovy does not need to import the `java.util` package or specify the `Date` type. This is very handy when using Groovy to evaluate user input. In those cases, one cannot assume that the user is proficient in Java package structures or willing to write more code than necessary. Additionally, Groovy doesn't require semicolons when it can understand the code without them. Despite being more compact, Groovy is fully comprehensible to a Java programmer.

The Ruby solution is listed to illustrate what Groovy avoids: a different packaging concept (`require`), a different comment syntax, and a different object-creation syntax. Scala introduces a new wildcard syntax with underscores and has its own way of declaring whether a reference is supposed to be (in Java terms) “final” or not (`var` vs. `val`). The user has to provide one or the other. Clojure doesn't support wildcard imports as of now and shows two alternative ways of instantiating a Java class, both of which differ syntactically from Java.

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

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

1.1.3 Power in your code: a feature-rich language

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

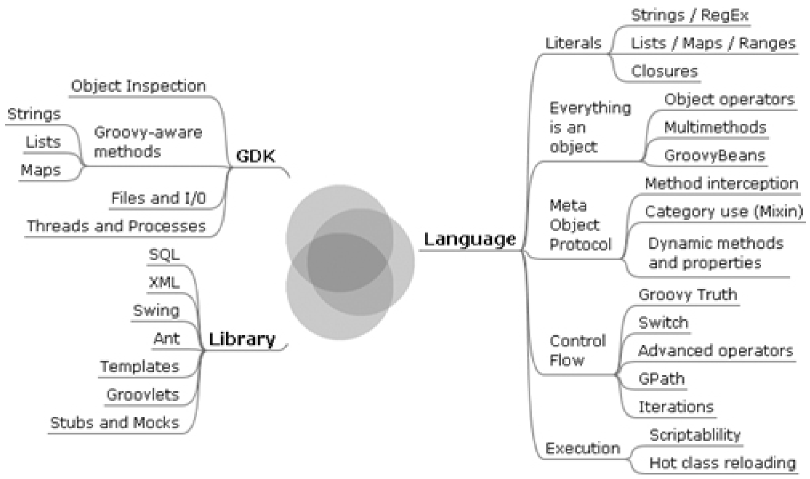


Figure 1.4 Many of the additional libraries and JDK enhancements in Groovy build on the new language features. The combination of the three forms a "sweet spot" for clear and powerful code.

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

LISTING A FILE: CLOSURES AND I/O ADDITIONS

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

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

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

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

which prints

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

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

PRINTING A LIST: COLLECTION LITERALS AND SIMPLIFIED PROPERTY ACCESS

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

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

Here's an example using these two features to print the package for each of a list of classes. Note that the word *clazz* is not *class* because that would be a Groovy keyword--exactly like in Java. Although Java would allow a similar first line to declare an array, we're using a real list here--elements could be added or removed with no extra work:

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

which prints

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

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

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

to produce the output

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

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

XML HANDLING THE GROOVY WAY: GPATH WITH DYNAMIC PROPERTIES

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

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

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

You can print out all the corporate customers with their names and companies using just the following code.

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

which prints

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

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

SCRIPTING THE WEB

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

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

The term *scripting* refers to the ability to take a string of program code and evaluate it at runtime. That string may be given as user input, read from a database, or fetched from the web like above. The text we fetch happens to be so simple[3] 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 3 It is actually JavaScript Object Notation (JSON) format.

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

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

1.1.4 Community-driven but corporate-backed

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

The size of the user base is a second criterion. The larger the user base, the greater the chance of obtaining good support and sustainable development. Groovy's user base has grown beyond all expectations. Recent polls suggest that Groovy is used in the majority of all organizations that develop professionally with Java, much higher than any alternative language. Groovy is regularly covered in Java conferences and publications, and virtually any Java open-source project that allows scripting extensions supports Groovy. Groovy and Grails mailinglists are the most busy ones at codehaus. Groovy has become an important item in many developers CVs and job descriptions.

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

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

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

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

NOTE **Groovy**
"A situation or an activity that one enjoys or to which one is especially well suited (found his groove playing bass in a trio). A very pleasurable experience; enjoy oneself (just sitting around, grooving on the music). To be affected with pleasurable excitement. To react or interact harmoniously." (http://dict.leo.org)

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

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

1.2 What Groovy can do for you

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

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

1.2.1 Groovy for Java professionals

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

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

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

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

to print all filenames recursively.

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

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

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

Besides command-line availability and code beauty, Groovy allows you to bring dynamic behavior to Java applications, such as through expressing business rules that can be maintained while the application is running, allowing smart configurations, or even implementing *domain specific languages*.

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

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

1.2.2 Groovy for script programmers

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

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

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

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

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

1.2.3 Groovy for pragmatic programmers, extremos, and agilists

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

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

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

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

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

NOTE **Example**

Recently, Guillaume and I did a spike on *prime number disassembly.* [4] We started with a small Groovy solution that did the job cleanly but not efficiently. Using Groovy's interception capabilities, we unit-tested the solution and counted the number of operations. Because the code was clean, it was a breeze to optimize the solution and decrease the operation count. It would have been much more difficult to recognize the optimization potential in Java code. The final result can be used from Java as it stands, and although we certainly still have the option of porting the optimized solution to plain Java, which would give us another performance gain, we can defer the decision until the need arises.

Footnote 4 Every ordinal number N can be uniquely disassembled into factors that are prime numbers: N = 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.

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

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

Table 1.1 Commands to execute Groovy

Command	What it does
`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>
```

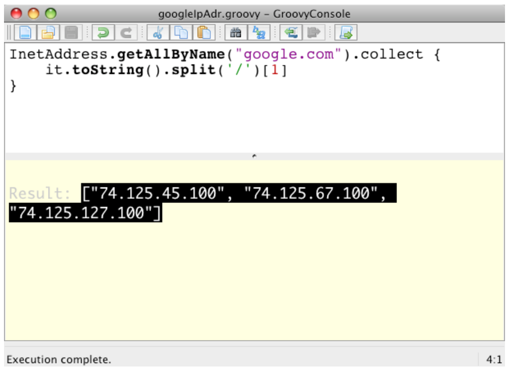
The shell is a good companion when you work on a remote server with only a text terminal being available. For the more common case that you work on a desktop or laptop machine, there are more comfortable options as we will see in a minute.

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]
}
```

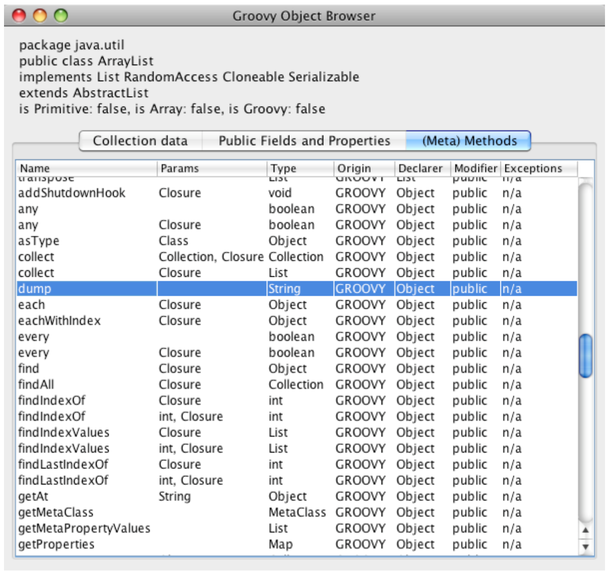
```
Result: ["74.125.45.100", "74.125.67.100",
"74.125.127.100"]
```

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

The `groovyConsole` comes with all the user interface goodness that you can expect from a Swing application.[5] Walk through the 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 5 Thanks to Romain Guy, the user interface expert and co-author of Filthy Rich Clients who supported the Groovy team here.

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



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
transpose		List	GROOVY	List	public	n/a
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.[6]

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

We calculate the golden ratio by narrowing down on the ratio of adjacent Fibonacci [7] 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 7 Leonardo Pisano (1170..1250), aka Fibonacci, was a mathematician from Pisa (now a town in Italy). He introduced this number sequence to describe the growth of an isolated rabbit population. Although this may be questionable from a biological point of view, his number sequence plays a role in many different areas of science and art. For more information, you can subscribe to the Fibonacci Quarterly.

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

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

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

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

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

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

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

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

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

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

The `groovy` command has many additional options that are useful for command-line scripting. For example, expressions can be executed by typing `groovy -e "println Math.PI"`, which prints `3.141592653589793` to the console. Section 12.3 will lead you through the full range of options, with numerous examples.

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

1.4 Compiling and running Groovy

So far, we have used Groovy in *direct*[8] mode, where our code is directly executed without producing any executable files. In this section, you will see a second way of using Groovy: compiling it to Java bytecode and running it as regular Java application code within a Java Virtual Machine (JVM). This is called *precompiled* mode. Both ways execute Groovy inside a JVM eventually, and both ways compile the Groovy code to Java bytecode. The major difference is *when* that compilation occurs and whether the resulting classes are used in memory or stored on disk.

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

1.4.1 Compiling Groovy with groovyc

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

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

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

For each script, `groovyc` generates a class that extends `groovy.lang.Script`, which contains a `main` method so that `java` can execute it. The name of the compiled class matches the name of the script being compiled. More classes may be generated, depending on the script code.

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

1.4.2 Running a compiled Groovy script with Java

Running a compiled Groovy program is identical to running a compiled Java program, with the added requirement of having the embeddable groovy-all-*.jar file in your JVM's classpath, which will ensure that all of Groovy's third-party dependencies will be resolved automatically at runtime. Make sure you add the directory in which your compiled program resides to the classpath, too. You then run the program in the same way you would run any other Java program, with the `java` command. [9]

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

```
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[10] may become available at runtime that the compiler cannot foresee. Therefore, it cannot shield you from mistyped method names. But the IDE can warn you. It can highlight unknown method names and even apply so-called type inference to give even better warnings and type-inferred code completion.

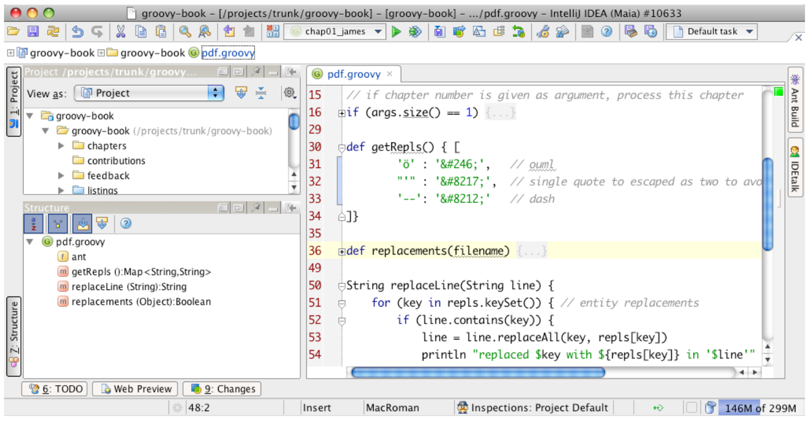
Footnote 10 This applies to more than just method names but we keep it short for the beginning.

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

1.5.1 IntelliJ IDEA plug-in

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

Listing all the features of JetGroovy would be a silly attempt. I wouldn't even know where to start. It may be enough to say that any Groovy code is so tightly integrated that the lines with Java begin to blur. The screenshot in figure 1.7 shows a Groovy script that produces this book from docbook format to PDF. Note that the method `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>`.



```
pdf.groovy
  ant
  getRepls ():Map<String,String>
  replaceLine (String):String
  replacements (Object):Boolean
```

```
// 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[11] blog and the quick-start guide[12].

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

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

1.5.3 Eclipse plug-in

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

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

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

1.5.4 Groovy support in other editors

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

The cross-platform *JEdit* editor comes with a plug-in for Groovy that supports executing Groovy scripts and code snippets. A syntax-highlighting configuration is available separately. More details are available here: http://groovy.codehaus.org/JEdit+Plugin.

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

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

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

1.6 Summary

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

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

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

Part 1
The Groovy language

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

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

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

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

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

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

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

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

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

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

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

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.[13]

Footnote 13 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![14]

Footnote 14 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.

NOTE **Real Life**

One real-life example of the value of assertions is in your hands right now (or on your screen). This book is constructed in a way that allows us to run the example code and the assertions it contains. This works as follows: There is a docbook XML version of this book that contains no code, but only placeholders that refer to files containing the code. With the help of a little Groovy script, all the listings are evaluated before the normal production process even begins. For instance, the assertions in listing 2.1 were evaluated and found to be correct during the substitution process. If an assertion fails, the process stops with an error message.

The fact that you are reading a production copy of this book, means the production process was not stopped and all assertions succeeded. This should give you confidence in the correctness of the Groovy examples we provide. For the first edition, we did the same with MS-Word using Scriptom (chapter 15) to control MS-Word and AntBuilder (chapter 8) to help with the building side--as we said 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.

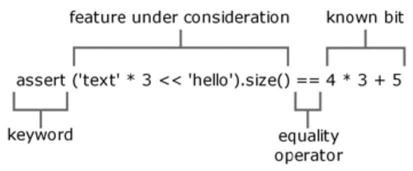

feature under consideration
known bit
assert ('text' * 3 << 'hello').size() == 4 * 3 + 5
keyword
equality operator

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.[15]

Footnote 15 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.

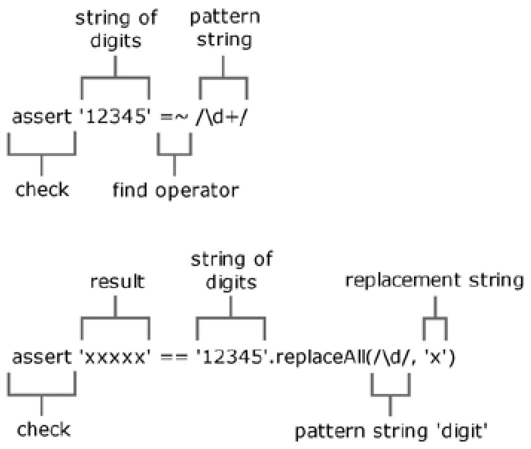


Figure 2.2 Regular expression support in Groovy through operators and slashy strings

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.

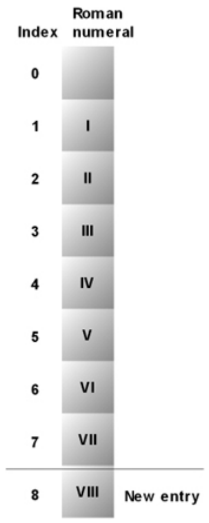


Figure 2.3 An example list where the content for each index is the Roman numeral for that index

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'] // #1 List of Roman numer
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 [16] return codes in a map, as depicted in figure 2.4.

Footnote 16 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.

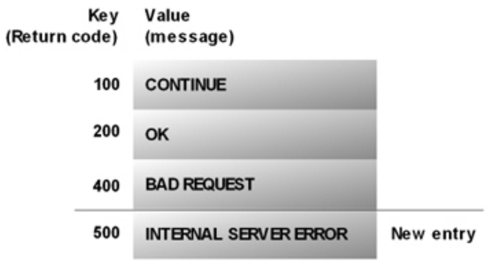


Figure 2.4 An example map where HTTP return codes map to their respective messages

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]`.

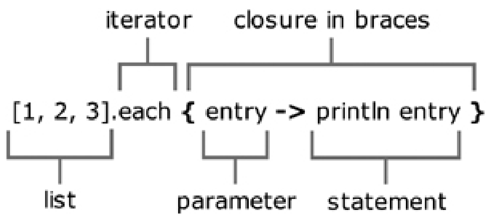


Figure 2.5 A simple example of a closure that prints the numbers 1, 2 and 3

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?[17] Figure 2.6 sketches this question for five people, where each line represents one clink.

Footnote 17 Or, in computer terms: What is the maximum number of distinct connections in a dense network of *n* components?

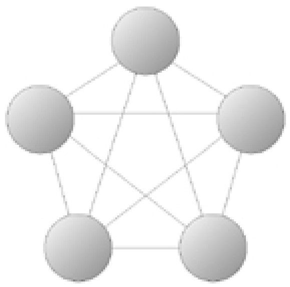


Figure 2.6 Five elements and their distinct connections,

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 [18] for this problem--with 100 people, there should be 4,950 clinks.

Footnote 18 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.[19]

Footnote 19 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.[20]

Footnote 20 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*.

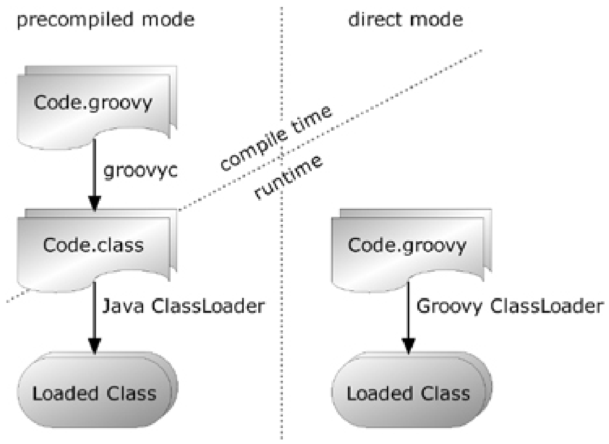


Figure 2.7 Groovy code can be compiled using groovyc and then loaded with the normal Java classloader, or loaded directly with the Groovy classloader

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[21], and it provides significant benefits in consistency, power, and expressiveness.

Footnote 21 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.

NOTE **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.

Table 1.1 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.

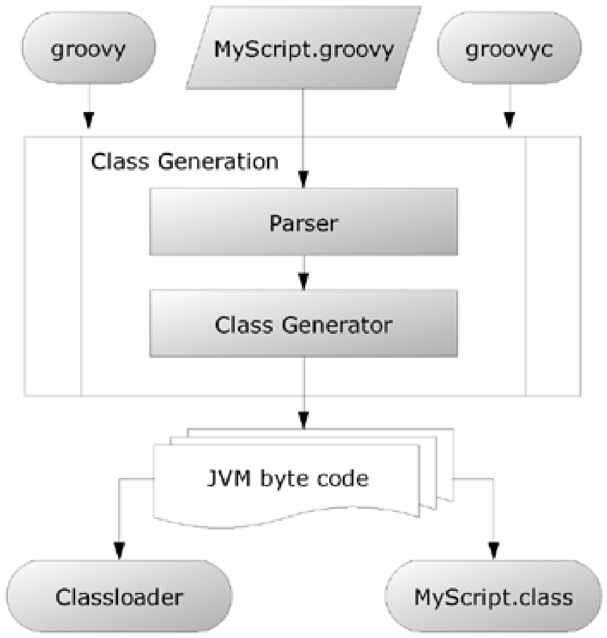


Figure 2.8 Flow chart of the Groovy bytecode generation process when executed in the runtime environment or compiled into class files. Different options for executing Groovy code involve different targets for the bytecode produced, but the parser and class generator are the same in each case.

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. [22]

Footnote 22 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[23]

Footnote 23 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.

NOTE **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.[24] 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 24 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[25] reference types, but only for primitive types (with the notable exception of `+`, which is also supported for strings).

Footnote 25 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.

Table 1.1 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.

Table 1.2 Numeric literals in Groovy

Type	**Example literals**
`java.lang.Integer`	`15, 0x1234ffff`
`java.lang.Long`	`100L, 200l` 26
`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.

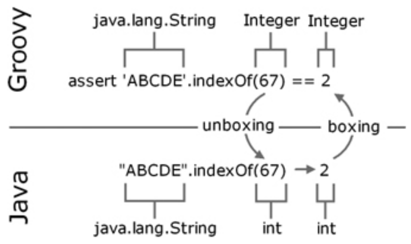


Figure 3.1 Autoboxing in action: An Integer parameter is unboxed to an int for the Java method call, and an int return value is boxed into an Integer for use in Groovy.

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[27] the second `Integer` object as an argument. The method call returns an `Integer` object of value `2`.

Footnote 27 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.

Table 1.3 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[28], 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 28 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 `ClassCastExceptions` and other runtime typing errors.

All this explains why the Groovy compiler[29] takes no issue with

Footnote 29 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*[30] of the `returnsObject()` reference. We hope that at runtime it will yield an `Integer`, which becomes its *runtime type*[31]. The Groovy code

Footnote 30 This is usually also called the "static" type but we avoid this term here to avoid further confusion.

Footnote 31 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:

NOTE **Groovy types are NOT DYNAMIC, they NEVER CHANGE**
If I could make the ink blink, I would! The word "dynamic" does not mean that the type of a reference, once declared, can ever change. Once we've declared `Integer myInt` we cannot execute `myInt = new Object()`. This will throw a `GroovyCastException`. We can only assign a value which Groovy can cast to an `Integer`.

As you see, the phrase "dynamic typing" can be misleading and is best avoided.

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.

NOTE **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.

Table 1.4 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()`[32]
`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

NOTE **The case of equals**

Nothing is easier than determine whether `a==b` is true, right? Well, only at first sight if you want this to be a useful equality check. First, if both are null, they should count as equal. Second, if they reference the same object they are equal without the need for 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 that `a==b`, right? But this may impose a conflict if we have a `Comparable` object that doesn't implement `equals` consistently. This is why Groovy only looks at the `compareTo` method for `Comparable` objects when doing the equality check and ignores the `equals` method in this case. You find the full logic implemented in the Groovy runtime under `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*[33]pattern. We allow values of the same currency to be summed, but do not support multicurrency addition.

Footnote 33 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.

NOTE 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*[34], and it helps to avoid duplicated, asymmetric, possibly inconsistent code. Let's reverse our previous example and consider `Integer.plus (BigDecimal operand)`.

Footnote 34 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*[35] operators. Test rigorously, and beware of endless cycles.

Footnote 35 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.

Table 1.5 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[36]

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[37]--at which point life is slightly harder, because specifying `\u` involves using a GString or specifying the Unicode escape sequence for a backslash.

Footnote 37 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.

Table 1.6 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` 38	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.[39] 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 39 `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 GStrinc
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'
```

NOTE **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 [40] 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 40 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`![41] 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 41 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 **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.

Table 1.7 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.

NOTE **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.

Table 1.8 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

NOTE **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.[42]

Footnote 42 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.

NOTE **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.

NOTE **What is a match?**

A match is the occurrence of a regular expression pattern in a string. It is therefore a string: a substring of the original string. When the pattern contains groupings like in `/begin(.*?)end/`, we need to know more information: not just the string matching the whole pattern, but also what part of that string matched each group. Therefore, the match becomes a list of strings, containing the whole match at position 0 with group matches being available as `match[` *n*`]` where *n* is group number *n*. Groups are numbered by the sequence of their opening 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.

NOTE **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.

NOTE **Use whitespace wisely**

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

It goes without saying that being explicit with whitespace is good programming style, even when the meaning is unambiguous for the parser. Do it for the next 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']
```

NOTE **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.

Table 1.9 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.

Table 1.10 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.

WARNING **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.

NOTE **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 `enums` 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]`.

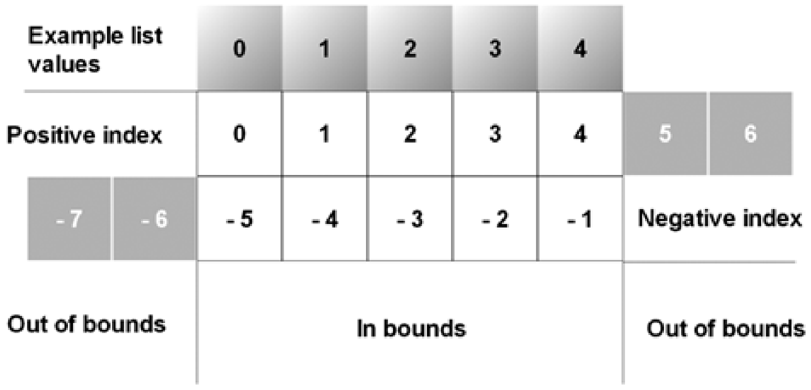


Figure 4.1 Positive and negative indexes of a list of length five, with “in bounds” and “out of bounds” classification for indexes

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.

NOTE 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 even comes into play. This can lead to surprises when mixing positive left and negative right 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- `nulls`--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 [43] 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 43 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.

NOTE **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.[44]

Footnote 44 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[45] types for keys, you need to obey the rules as outlined in the Javadoc for `java.util.Map`.

Footnote 45 *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 `SortedMap`s)
- `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 !!
```

NOTE *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* [46] object).

Footnote 46 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.

5 Working with closures

- A gentle introductions to closures from a Java programmer's perspective
- Declaring, passing, and calling closures
- Closure use cases

I wouldn't like to build a tool that could only do what I had been able to imagine for it.

-- Bjarne Stroustrup

Closures are important. Very important. They're arguably one of the most useful features of Groovy--but at the same time they can be a strange concept until you fully understand them. In order to get the best out of Groovy, or to understand anyone else's Groovy code, you're going to have to be comfortable with them. Not just "met them once at a wedding" comfortable, but "invite them over for a barbecue on the weekend" comfortable.

Closures aren't *hard*, though--they're just different to anything you might be used to. In a way, this is strange, because one of the chief tenets of object-orientation is that objects have behavior as well as data. Closures are objects whose main purpose in life is their behavior--that's almost all there is to them.

In the past few chapters, you've seen a few uses of closures, so you might already have a good idea of what they're about. Please forgive us if we seem to be going over the same ground again--it's so important, we'd rather repeat ourselves than leave you without a good grasp of the basic principles.

In this chapter, we will introduce the fundamental concept of closures (again), explain their benefits, and then show how they can be declared and called. After this basic treatment, we will look in a bit more depth at other methods available on closures and the *scope* of a closure--that is, the data and members that can be accessed within it--as well as considering what it means to return from a closure. We end the chapter with a discussion of how closures can be used to implement many common design patterns and how they alleviate the need for some others by solving the problem in a different manner.

So, let's take a look at what closures really are in the first place.

5.1 A gentle introduction to closures

Let's start with a simple definition of closures, and then we'll expand on it with an example. A *closure* is a piece of code wrapped up as an object. It acts like a method in that it can take parameters and it can return a value. It's a normal object in that you can pass a reference to it around just as you can a reference to any other object. Don't forget that the JVM has no idea you're running Groovy code, so there's nothing particularly odd that you *could* be doing with a closure object. It's just an object. Groovy provides a very easy way of creating closure objects and enables some very smart behavior.

If it helps you to think in terms of real-world analogies, consider an envelope with a piece of paper in it. For other objects, the paper might have the values of variables on it: “x=5, y=10” and so on. For a closure, the paper would have a list of instructions. You can give that envelope to someone, and that person might decide to follow the instructions on the piece of paper, or they might give the envelope to someone else. They might decide to follow the instructions lots of times, with a different context each time. For instance, the piece of paper might say, “Send a letter to the person you're thinking of,” and the person might flip through the pages of their address book thinking of every person listed in it, following the instructions over and over again, once for each contact in that address book.

The Groovy equivalent of that example would be something like this:

```
Closure envelope = { person -> new Letter(person).send() }
addressBook.each (envelope)
```

That's a fairly long-winded way of going about it, but it shows the distinction between the closure itself (in this case, the value of the `envelope` variable) and its use (as a parameter to the `each` method). Part of what makes closures unfamiliar when coming to them for the first time is that they're usually used in an abbreviated form. Groovy makes them very concise because they're so frequently used--but that brevity can be detrimental to the learning process. Just for the comparison, here's the previous code written using the shorthand Groovy provides. When you see this shorthand, it's often worth mentally separating it out into the longer form:

```
addressBook.each { new Letter(it).send() }
```

It's still a method call passing a closure as the single parameter, but that's all hidden--passing a closure to a method is sufficiently common in Groovy that there are special rules for it. Similarly, if the closure needs to take only a single parameter to work on, Groovy provides a default name--`it`--so that you don't need to declare it specifically. That's how our example ends up so short when we use all the Groovy shortcuts.

Now, before going into details with the various ways of declaring a closure, let's think about why we would want to have closures in the first place. Just keep remembering: They're objects that enclose some code, and Groovy provides neat syntax for them.

5.2 The case for closures

Java as a *platform* is great: it's portable, stable, scalable, and it performs reasonably well. Java as a *language* has a lot of advantages but unfortunately also some shortcomings.

Some of those deficiencies can be addressed in Groovy through the use of closures. We'll look at two particular areas that benefit from closures: performing everyday tasks with collections, and using resources in a safe manner. In these two common situations, you need to be able to perform some logic that is the same for every case and execute arbitrary code to do the actual work. In the case of collections, that code is the body of the iterator; in the case of resource handling, it's the use of the resource after it's been acquired and before it's been released. In general terms, such a mechanism uses a *callback* to execute the work. Closures are Groovy's way of providing transparent callback targets as first-class citizens.

5.2.1 Using iterators

A typical construction in Java code is traversing a collection with an iterator:

```
// since Java 5
for (ItemType item : list) {
    // do something with item
}
```

The syntax may not be ideal[47]--the Java 5 designers were constrained in terms of adding keywords--but it gets the job done, right? Clearly it's useful to have a `for` loop that iterates through every item in a collection--otherwise Groovy wouldn't have it, for starters. (Groovy's `for` statement is somewhat broader in scope than Java 5's, however; see chapter 6 for more details.) It's useful, but it's not everything we could wish for. There are common patterns for *why* we want to iterate through a collection, such as

- finding whether a particular condition is met by any or every element,
- finding *all* elements met by a condition, or
- transforming each element into another, thereby creating a new collection.

Footnote 47 Groovy supports the Java 5 colon notation for compatibility but encourages using `in` instead as that's clearer in terms of revealing the meaning of the expression and the role of the operands in use.

It would be madness to have a specialized syntax for all of those patterns. Making a language too smart in a non-extensible way ends up like a road through the jungle--it's fine when you're doing something anticipated by the designers, but as soon as you stray off the path, life is tough. So, without direct language support for all those patterns, what's left? Each of the patterns relies on executing a particular piece of code again and again, once for each element of the collection. Java has no concept of “a particular piece of code” unless it's buried in a method. That method can be part of an interface implementation, but at that point each piece of code needs its own (possibly anonymous) class, and life gets very messy.

Groovy uses closures to specify the code to be executed each time and adds the extra methods (`each`, `any`, `every` `find`, `findAll`, `collect` and so forth) to the collection classes to make them readily available. Those methods aren't magic, though--they're simple Groovy, because closures allow the *controlling* logic (the process of iterating over the collection) to be separated from the code to execute for every element. If you find yourself wanting a similar construct that isn't already covered by Groovy, you can add it easily as we will see below.

Separating iteration logic from what to do on each iteration is not the only reason for introducing the closure concept. A second reason, perhaps even more important than the first, is the use of closures to handle resources in a safe and convenient manner.

5.2.2 Handling resources

How many times have you seen code that opens a stream but calls `close` at the end of the method, overlooking the fact that the `close` statement may never be reached when an exception occurs while processing? So, it needs to be protected with a `try-catch` block? No--wait--that should be `try-finally`, or should it? And inside the `finally` block, `close` can throw another exception that needs to be handled. There are too many details to remember, and so resource handling is often implemented incorrectly. With Groovy's closure support, you can put that logic in one place and use it like this:

```
new File('myfile.txt').eachLine { println it }
```

The `eachLine` method of `File` now takes care of opening and closing the file input stream properly. This guards you from accidentally producing a resource leak of file handles.

Streams are just the most obvious tip of the resource-handling iceberg.

- Database connections,
- transactions,
- native handles such as graphic resources,
- network connections,
- threads from a thread pool,

even your GUI is a resource that needs to be managed (that is, repainted correctly at the right time), and observers and event listeners need to be removed when the time comes, to avoid memory leaks.

Forgetting to clean up correctly in all situations *ought* to be a problem that only affects neophyte Java programmers, but because the language provides little help beyond `try-catch-finally`, even experienced developers end up making mistakes. It is possible to code around this in an orderly manner, but Java leads inexperienced programmers away from centralized resource handling. Code structures are duplicated many times, and the probability of not-so-perfect implementations rises with the number of duplicates.

Resource-handling code is often tested poorly. Projects that measure their test coverage typically struggle to fully cover this area. That is because duplicated, widespread resource handling is difficult to test and eats up precious development time. Testing centralized handlers is easy and requires only a single set of tests.

Let's see what resource handling solutions Java provides and why they are not used often, and then we'll show the corresponding Groovy solutions.

A COMMON JAVA APPROACH: USE INNER CLASSES

In order to do centralized resource handling, you need to pass resource-using code to the handler. This should sound familiar by now--it's essentially the same problem we encountered when considering collections: The handler needs to know how to call that code, and therefore it must implement some known interface. In Java, this is frequently implemented by an inner class for two reasons: First, it allows the resource-using code to be close to the calling code (which is often useful for readability); and second, it allows the resource-using code to interact with the context of the calling code, using local variables, calling methods on the relevant objects, and so on.

NOTE **By the Way**
JUnit, one of the most prominent Java packages outside the JDK, follows this strategy by using the `Runnable` interface with its `runProtected` method.

Anonymous inner classes are almost solely used for this kind of pattern--if Java had closures, it's possible that anonymous inner classes might never have been invented. The rules and restrictions that come with them (and with plain inner classes) make it obvious what a wart the whole “feature” really is on the skin of what is otherwise an elegant and simple language. As soon as you have to start typing code like `MyClass.this.doSomething`, you know something is wrong--and that's aside from the amount of distracting clutter required around your code just to create it in the first place. The interaction with the context of the calling code is limited, with rules such as local variables having to be final in order to be used making life awkward.

In some ways, it's the right approach, but it looks ugly, especially when used often. Java's limitations get in the way too much to make it an elegant solution. The following example uses a `Resource` that it gets from a `ResourceHandler`, which is responsible for its proper construction and destruction. Only the boldface code is really needed for doing the job:

```
// Java
interface ResourceUser {
    void use (Resource resource);
}
resourceHandler.handle (new ResourceUser(){
        public void use (Resource resource) {
            resource.doSomething()
        }
});
```

The Groovy equivalent of this code reveals all necessary information without any waste:

```

resourceHandler.handle { resource->
    resource.doSomething()
}
```

Groovy's scoping is also significantly more flexible and powerful, while removing the “code mess” that inner classes introduce.

AN ALTERNATIVE JAVA APPROACH: THE TEMPLATE METHOD PATTERN

Another strategy to centralize resource handling in Java is to do it in a superclass and let the resource-using code live in a subclass. This is the typical implementation of the Template Method [GOF] pattern.

The downside here is that you either end up with a proliferation of subclasses or use (possibly anonymous) inner subclasses... which brings us back to the drawbacks discussed earlier. It also introduces penalties in terms of code clarity and freedom of implementation, both of which tend to suffer when inheritance is involved. This leads us to take a close look at the dangers of abstraction proliferation.

If there were only *one* interface that could be used for the purpose of passing logic around, like our imaginary `ResourceUser` interface from the previous example, then things would not be too bad. But in Java there is no such beast--no single `ResourceUser` interface that serves all purposes. The signature of the callback method `use` needs to adapt to the purpose: the number and type of parameters, the number and type of declared exceptions, and the return type.

Therefore a variety of interfaces has evolved over time: Runnable, Observer, Listener, Visitor, Comparator, Strategy, Command, Controller, and so on. This makes their use more complicated, because with every new interface, there also is a new abstraction or concept that needs to be understood.

In comparison, Groovy closures can handle any method signature, and the behavior of the controlling logic can even change depending on the signature of the closure provided to it, as you'll see later.

These two examples of pain-points in Java that can be addressed with closures are just that--examples. If they were the only problems made easier by closures, closures would still be worth having, but reality is much richer. It turns out that closures enable many patterns of programming that would be unthinkable without them.

Before you can live your dreams, however, you need to learn more about the basics of closures. Let's start with how we declare them in the first place.

5.3 Declaring closures

So far, we have used the simple abbreviated syntax of closures: After a method call, put your code in curly braces with parameters delimited from the closure body by an arrow.

Let's start by adding to your knowledge about the simple abbreviated syntax, and then we'll look at two more ways to declare a closure: by using them in assignments and by referring to a method.

5.3.1 The simple declaration

Listing 5.1 shows the simple closure syntax plus a new convenience feature. When there is only one parameter passed into the closure, its declaration is optional. The magic variable `it` can be used instead. Listing 5.1 shows two equivalent closure declarations.

Listing 5.1 Simple abbreviated closure declaration

```
def log = ''
(1..10).each { counter -> log += counter }
assert log == '12345678910'

log = ''
(1..10).each { log += it }
assert log == '12345678910'
```

Note that unlike `counter`, the magic variable `it` needs no declaration.

This syntax is an abbreviation because the closure object as declared by the curly braces is the last parameter of the method and would normally appear within the method call's parentheses. As you will see, it is equally valid to put it inside parentheses like any other parameter, although it is hardly ever used this way[48]:

Footnote 48 Although I have seen long-time Java programmers that preferred this style and kept it over a number of weeks before getting comfortable with more idiomatic Groovy.

```
def log = ''
(1..10).each({ log += it })
assert log == '12345678910'
```

This syntax is simple because it uses only one parameter, the implicit parameter `it`. Multiple parameters can be declared in sequence, delimited by commas. A default value can optionally be assigned to parameters, in case no value is passed from the method to the closure. We will see examples of this in section 5.4.

NOTE **Tip**
Think of the arrow as an indication that parameters are passed from the method on the left into the closure body on the right.

5.3.2 Using assignments for declaration

A second way of declaring a closure is to directly assign it to a variable:

```
Closure printer = { line -> println line }
```

The closure is declared inside the curly braces and assigned to the `printer` variable.

NOTE **Tip**
Whenever you see the curly braces of a closure, think: `new Closure(){}`.

There is also a natural kind of assignment to the return value of a method:

```
def Closure getPrinter() {
    return{ line -> println line }
}
```

Again, the curly braces denote the construction of a new closure object. This object is returned from the method call.

NOTE **Tip**
Curly braces can denote the construction of a new closure object or a Groovy *block*. Blocks can be class, interface, static or object initializers, or method bodies; or can appear with the Groovy keywords `if`, `else`, `synchronized`, `for`, `while`, `switch`, `try`, `catch`, and `finally`. All other occurrences are closures.

As you can see, closures are objects. They can be stored in variables, they can be passed around, and, as you probably guessed, you can call methods on them. Being objects, you can also return a closure from a method.

5.3.3 Referring to methods as closures

The third way of declaring a closure is to reuse something that is already declared: a method. Methods have a body, optionally return values, can take parameters, and can be called. The similarities with closures are obvious, so Groovy lets you reuse the code you already have in methods, but as a closure. Referencing a method as a closure is performed using the *reference*`.&` operator. The reference is used to specify which instance should be used when the closure is called, just like a normal method call to *reference*`.`*someMethod*`()`. Figure 5.1 shows an assignment using a method closure, breaking the statement up into its constituent parts.

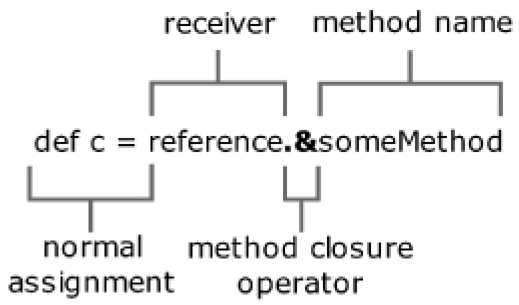


Figure 5.1 The anatomy of a simple method closure assignment statement

Listing 5.2 demonstrates method closures in action, showing two different instances being used to give two different closures, even though the same method is invoked in both cases.

Listing 5.2 Simple method closures in action

```
class SizeFilter {
    Integer limit
    boolean filter (String value) {
        return value.length() <= limit
    }
}

SizeFilter six  = new SizeFilter(limit:6)  //#1 GroovyBean constructor
SizeFilter five = new SizeFilter(limit:5)  //#1 calls

Closure smallerSix = six.&filter           //#2 Method closure assignment

def words = ['long string', 'medium', 'short', 'tiny']

assert 'medium' == words.find (smallerSix)     //#3 Calling with closure
assert 'short'  == words.find (five.&filter)   //#4 Passing a method closure directl
```

Each instance (created at ❶) has a separate idea of how long a string it will deem to be valid in the `filter` method. We create a reference to that method with `six.&filter` at ❷ and `five.&filter`, showing that the reference can be assigned to a variable which is then passed (at ❸) or passed as a parameter to the `find` method at ❹ . We use a sample list of words to check that the closures are doing what we expect them to.

Method closures are limited to instance methods, but they do have another interesting feature--runtime overload resolution, also known as *multimethods*. You will find out more about multimethods in chapter 7, but listing 5.3 gives a taste.

Listing 5.3 Multimethod closures--the same method name called with different parameters is used to call different implementations

```
class MultiMethodSample {

    int mysteryMethod (String value) {
        return value.length()
    }
    int mysteryMethod (List list) {
        return list.size()
    }
    int mysteryMethod (int x, int y) {
        return x+y
    }
}

MultiMethodSample instance = new MultiMethodSample()
Closure multi = instance.&mysteryMethod       //#1 Only a single closure is created

assert 10 == multi ('string arg')             //#2 Different implementations
assert  3 == multi (['list', 'of', 'values']) //#2 are called based on
assert 14 == multi (6, 8)                     //#2 argument types
```

Here a single instance is used, and indeed a single closure (at ❶)--but each time it's called, a different method implementation is invoked, at ❷ . We don't want to rush ahead of ourselves, but you'll see a lot more of this kind of dynamic behavior in chapter 7.

With the topic presented so far, you should be able to understand a construction that the Grails framework uses pervasively: properties of type `Closure`. The interesting effect of this construction is that you can change a property value at runtime, and consequently, you can assign a new closure to a property just as well. Listing 0.0 mimics a Grails controller where we change the list “action” at runtime.

Listing 5.4 ClosureProperty that looks like a method but can be redefined at runtime

```
class MultiMethodSample {

    int mysteryMethod (String value) {
        return value.length()
    }
    int mysteryMethod (List list) {
        return list.size()
    }
    int mysteryMethod (int x, int y) {
        return x+y
    }
}

MultiMethodSample instance = new MultiMethodSample()
Closure multi = instance.&mysteryMethod       //#1 Only a single closure is created

assert 10 == multi ('string arg')             //#2 Different implementations
assert  3 == multi (['list', 'of', 'values']) //#2 are called based on
assert 14 == multi (6, 8)                     //#2 argument types
```

What makes ClosureProperties attractive is that calling the closure looks exactly like a method call but the ability to assign a new closure object yields possibilities that would otherwise only be available through metaprogramming: redefining execution logic at runtime on a per-instance basis.

Now that you've seen all the ways of declaring a closure, it's worth pausing for a moment and seeing them all together, performing the same function, just with different declaration styles.

5.3.4 Comparing the available options

Listing 5.4 shows all of these ways of creating and using closures: through simple declaration, assignment to variables, and method closures. In each case, we call the `each` method on a simple map, providing a closure that doubles a single value. By the time we've finished, we've doubled each value three times.

Listing 5.5 List of closure declaration examples

```
Map map = ['a':1, 'b':2]
map.each{ key, value -> map[key] = value * 2 }          //#1 Parameter sequence with
assert map == ['a':2, 'b':4]

Closure doubler = {key, value -> map[key] = value * 2 } //#2 Assign and then call
map.each(doubler)                                       //#2 a closure reference
assert map == ['a':4, 'b':8]

def doubleMethod (entry){                               //|#3 A usual method
    entry.value = entry.value * 2                       //|#3 declaration
}
doubler = this.&doubleMethod                            //#4 Reference and call
map.each(doubler)                                       //#4 a method as a closure
assert map == ['a':8, 'b':16]
```

In ❶ , we pass the closure as the parameter directly. This is the form you've seen most commonly so far.

The declaration of the closure in ❷ is disconnected from its immediate use. The curly braces are Groovy's way to declare a closure, so we assign a closure object to the variable `doubler`. Some people incorrectly interpret this line as assigning the *result* of a closure call to a variable. Don't fall into that trap! This line creates a `Closure` object, but doesn't call it. The `each` method calls it instead. You can see that passing the closure as an argument to the method via a reference is exactly the same as declaring it *in-place*, the style that we followed in all the previous examples.

The method declared in ❸ is a perfectly ordinary method. There is no trace of our intention to use it as a closure.

In ❹ , the `reference.&` operator is used for referencing a method name as a closure. Again, the method is not immediately called; the execution of the method occurs as part of the next line. This is just like ❷ . The closure is passed to the `each` method, which calls it back for each entry in the map.

Typing [49] is optional in Groovy, and consequently it is optional for closure parameters. Just like for methods, if you choose to specify explicit type markers for closure parameters, Groovy guarantees that those parameters will be the right time at runtime.

Footnote 49 The word *typing* has two meanings: declaring object types and typing keystrokes. Although Groovy provides optional typing, you still have to key in your program code.

In order to fully understand how closures work and how to use them within your code, you need to find out how to invoke them. That is the topic of the next section.

5.4 Using closures

So far, you have seen how to create a closure for the purpose of passing it into a method such as `each`. But what happens inside the `each` method? How does it call your closure? If you knew this, you could come up with equally smart implementations. First we'll look at how simple it is to call a closure and then move on to explore some advanced methods that the `Closure` type has to offer.

5.4.1 Calling a closure

Suppose we have a reference `x` pointing to a closure; we can call it with `x.call()` or simply `x()`. You have probably guessed that any arguments to the closure call go between the parentheses.

We start with a simple example. Listing 5.5 shows the same closure being called both ways.

Listing 5.6 Calling closures

```
def adder = { x, y -> return x+y }

assert adder(4, 3) == 7
assert adder.call(2, 6) == 8
```

We start off by declaring pretty much the simplest possible closure--a piece of code that returns the sum of the two arguments it is passed. Then we call the closure both directly and using the `call` method. Both ways of calling the closure achieve exactly the same effect.

Now let's try something more involved. In listing 5.6, we demonstrate calling a closure from within a method body and how the closure gets passed into that method in the first place. The example measures the execution time of the closure.

Listing 5.7 Calling closures

```
def benchmark(int repeat, Closure worker) {      //#1 Put closures last
    def start = System.nanoTime()                //#2 Some pre-work
    repeat.times { worker(it) }                  //#3 Call closure a given number of

    def stop = System.nanoTime()                 //|#4 Some post-work
    return stop - start                          //|#4
}
def slow = benchmark(10000) { (int) it / 2 }    //|#5 Pass different closures
def fast = benchmark(10000) { it.intdiv(2) }    //|#5 for benchmarking
assert fast * 15 < slow                          //|#5
```

Do you remember our performance investigation for regular expression patterns in listing 3.7? We needed to duplicate the benchmarking logic because we had no means to declare how to benchmark *something*. Now you know how. You can pass a closure into the `benchmark` method, where some pre- and post-work takes control of timing it appropriately.

We put the closure parameter at the end of the parameter list in ❶ to allow the simple abbreviated syntax when calling the method. In the example, we declare the type of the parameter to be a closure. This is only to make things more obvious--the `Closure` type is optional.

We effectively start timing the benchmark at ❷ . From a general point of view, this is arbitrary pre-work like opening a file or connecting to a database. It just so happens that our resource is time.

At ❸ , we call the given closure as many times as our `repeat` parameter demands. We pass the current count to the closure to make things more interesting. From a general point of view, a resource is passed to the closure.

We stop timing at ❹ and calculate the time taken by the closure. This is where the post-work logic belongs: closing files, flushing buffers, returning connections to the pool, and so on.

The payoff comes at ❺ . We can now pass *logic* to the `benchmark` method. Note that we use the simple abbreviated syntax and use the magic `it` to refer to the current count. As a side effect, we learn that the general number division takes more than 15 times longer than the optimized `intdiv` method.

NOTE **By the Way**
This kind of benchmarking should not be taken too seriously. There are all kinds of effects that can heavily influence such wall-clock based measurements: machine characteristics, operating system, current machine load, JDK version, Just-In-time compiler and Hotspot settings, and so on.

Figure 5.2 shows the UML sequence diagram for the general calling scheme of the declaring object that creates the closure, the `method` invocation on the caller, and the caller's callback to the given closure.

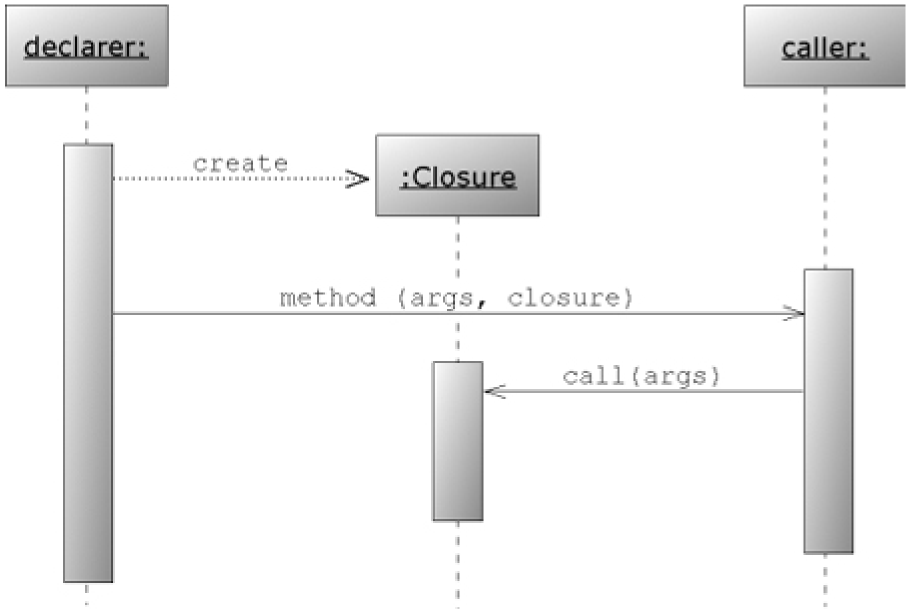


Figure 5.2 UML sequence diagram of the typical sequence of method calls when a declarer creates a closure and attaches it to a method call on the caller, which in turn calls that closure's call method

When calling a closure, you need to pass exactly as many arguments to the closure as it expects to receive, unless the closure defines default values for its parameters. A default value is used when you omit the corresponding argument. The following is a variant of the addition closure as used in listing 5.5, with a default value for the second parameter and two calls--one that passes two arguments, and one that relies on the default:

```
def adder = { x, y=5 -> return x+y }

assert adder(4, 3) == 7
assert adder.call(7) == 12
```

The same rules apply for default parameters in closures as they do for methods. Also, closures can be used with a variable length argument list in the same way as methods. We will cover this in section 7.1.2.

At this point, you should be comfortable with passing closures to methods and have a solid understanding of how the callback is executed. Whenever you pass a closure to a method, you can be sure that a callback will be executed one way or the other (perhaps conditionally), depending on that method's logic. Closures are capable of more than just being called, though. In the next section, you see what else they have to offer.

5.4.2 More closure methods

The class `groovy.lang.Closure` is an ordinary class, albeit one with extraordinary power and extra language support. It has various methods available beyond `call`. We will present the most the important ones--even though you will usually just declare and call closures, it's nice to know there's some extra power available when you need it.

REACTING ON THE PARAMETER COUNT

A simple example of how useful it is to change behavior based on the parameter count of a closure is map's `each` method, which we discussed in section 4.3.2. It passes either a `Map.Entry` object or key and value separately into the given closure, depending on whether the closure takes one argument or two. You can retrieve the information about the expected parameter count (and types, if declared) by calling the closure's `getParameterTypes` method:

```
def caller (Closure closure) {
    closure.getParameterTypes().size()
}

assert caller { one -> }      == 1
assert caller { one, two -> } == 2
```

As in the `Map.each` example, this allows for the luxury of supporting closures with different parameter styles, adapted to the caller's needs.

HOW TO CURRY FAVOR WITH A CLOSURE

Currying is a technique invented by Moses Schnfinkel and Gottlob Frege, and named after the logician Haskell Brooks Curry (1900..1982), a pioneer in *functional programming*. (Unsurprisingly, the functional language Haskell is also named after Curry.) The basic idea is to take a function with multiple parameters and transform it into a function with fewer parameters by fixing some of the values. A classic example is to choose some arbitrary value *n* and transform a function that sums two parameters into a function that takes a single parameter and adds *n* to it.

In Groovy, `Closure`'s `curry` method returns a clone of the current closure, having bound one or more parameters to a given value. Parameters are bound to `curry`'s arguments from left to right. Listing 5.7 gives an implementation of the addition example.

Listing 5.8 A simple currying example

```
def adder  = { x, y -> return x+y }
def addOne = adder.curry(1)
assert addOne(5) == 6
```

We reuse the same closure you've seen a couple of times now for general summation. We call the `curry` method on it to create a new closure, which acts like a simple adder, but with the value of the first parameter always fixed as 1. Finally, we check our results.

If you're new to closures or currying, now might be a good time to take a break--and pick the book up again back at the start of the currying discussion, to read it again. It's a deceptively simple concept to describe mechanically, but it can be quite difficult to internalize. Just take it slowly, and you'll be fine.

The real power of currying comes when the closure's parameters are themselves closures. This is a common construction in functional programming, but it does take a little getting used to.

For an example, suppose you are implementing a logging facility. It should support customized formatting and appending to an output device. The idea is to provide a single closure for a customized version of each activity, while still allowing you to implement the overall pattern of when to do the formatting and appending in one place. Listing 0.0 uses currying to inject the customized activities into that pattern:

Listing 5.9 Using the `curry` method to configure formatting and appending of log entries

```
def logger = { formatter, appender, line ->         //#1 General logging device
    appender(formatter(line))
}
def rightFormatter = { line -> line.padLeft(25) + "n" }    //|#2 Tool
def out = new StringBuilder()                               //|#2 details
def stringAppender = { line -> out << line }                //|#2

def myLog  = logger.curry(rightFormatter, stringAppender)    //#3 Custom device

myLog 'here is some log message'

assert out.toString().contains('log message')
```

Closure ❶ is like a recipe: given any output formatter, appender, and a line to log, that line must first be formatted and then appended--easy. The short closures in ❷ are the specific ingredients in the recipe. They could be specified every time, but we're always going to use the same ingredients. Currying (at ❸) allows us to remember just one object rather than each of the individual parts. To continue the recipe analogy, we've put all the ingredients together, and the result needs to be put in the oven whenever we want to do some logging.

Currying is a strategy that the functional programming community needed to develop since in a true functional world, you have no objects to carry *mutable state* . But in Groovy, we do have objects and can use them together with closure properties as a curry surrogate. Listing 0.0 puts the formatting and appending logic into closures that are held by the `GeneralLogger` as properties.

Listing 5.10 Property closures as a curry surrogate

```
class GeneralLogger {
    Closure formatter = { line -> line }    //|#1 Closure properties
    Closure appender  = { }                 //|#1
    Closure logger    = { line ->
        appender(formatter(line))
    }
}
def out = new StringBuilder()
def myLog = new GeneralLogger(
    formatter : { line -> line.padLeft(25) + "n" },
    appender  : { line -> out << line }
).logger

myLog 'here is some log message'

assert out.toString().contains('log message')
```

This is another compelling example where closure properties demonstrate their flexibility over methods with fixed implementations. Whatever style you prefer, Groovy gives you the freedom of choice.

CLASSIFICATION VIA THE ISCASE METHOD

Closures implement the `isCase` method to make closures work as classifiers with `in` `grep` and `switch`. In that case, the respective argument is passed into the closure, and calling the closure needs to evaluate to a Groovy Boolean value (see section 6.1). For example, consider this code:

```
def odd = { it % 2 == 1}

assert [1,2,3].grep(odd) == [1, 3]
```

```
switch(10) {
    case odd : assert false
}

if (2 in odd) assert false
```

You can see that the switch statement allows us to classify values with arbitrary logic. Again, this is only possible because closures are objects.

REMAINING METHODS

For the sake of completeness, it needs to be said that closures support the `clone` method in the usual Java sense.

The `asWriteable` method returns a clone of the current closure that has an additional `writeTo(Writer)` method to write the result of a closure call directly into the given `Writer`.

Finally, there are a getter and setter methods for the *delegate* of the closure. We will explore the topic of what a delegate is and how it is used inside a closure when investigating a closure's scoping rules in the next section.

5.5 Understanding scoping

You have seen how to *create* closures when they are needed for a method call and how to *work* with closures when they are passed to your method. This is very powerful while still simple to use.

In this section we'll look under the hood to get a deeper understanding of what happens when you use this simple construction. We explore what data and methods you can access from a closure, what the `this` reference means within a closure body, and how to put your knowledge to the test with a classic example designed to test any language's expressiveness.

This is a bit of a technical section, and you can safely skip it on your first read. However, at some point you may want to come back to it and learn how Groovy can provide all those clever tricks. Knowing the details may help you to come up with particularly elegant solutions yourself.

The environment available inside a closure is called its *scope*. The scope defines

- Which local variables are accessible
- What `this` (the current object) refers to
- Which fields and methods are accessible

We'll start with an explanation of the behavior that you have seen so far, revisiting a piece of code that does *something* 10 times:

```
def x= 0
10.times {
    x++
}
assert x== 10
```

Clearly the closure that is passed into the `times` method can access variable `x`, which is locally accessible when the closure is *declared*. Remember: The curly braces show the *declaration* time of the closure, not the *execution* time. The closure can access `x` for both reading and writing at *declaration* time.

This leads to a second thought: The closure surely needs to also access `x` at execution time. How could it increment it otherwise? But the closure is passed to the `times` method, a method that is called on the `Integer 10` object. That method, in turn, calls back to our closure. But the `times` method has no chance of knowing about `x`! So it cannot pass it to the closure, and it surely has no means of finding out what the closure is doing with it.

The only way in which this can possibly work is if the closure somehow remembers the context at the point of its declaration and carries it along throughout its lifetime. That way, it can work on that original context whenever the situation calls for it.

This is called the *birthday context* of the closures. It needs to be a reference, not a copy: if we merely copied the values of the fields (as Java does for anonymous inner classes), there would be no way of changing the original from inside the closure. But our example clearly does change the value of `x`--otherwise the assertion would fail. Therefore, the birthday context must be a reference.

5.5.1 The simple variable scope

Figure 5.3 depicts your current understanding of which objects are involved in the `times` example and how they reference each other.

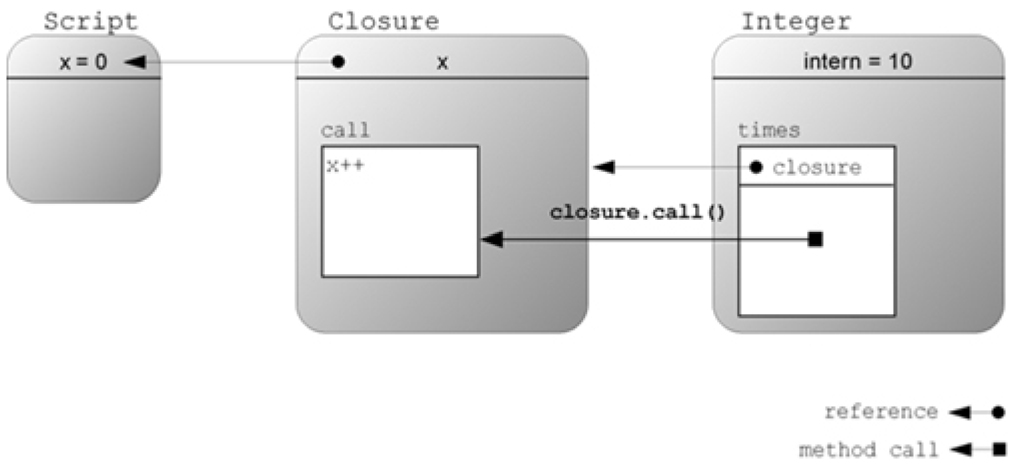


Figure 5.3 Conceptual view of object references and method calls between a calling script, an Integer object of value 10 that is used in the script, and the closure that is attached to the Integer's times method for defining something that has to be done 10 times

The `Script` creates a `Closure` that has a back reference to `x`, which is in the local scope of its declarer. `Script` calls the `times` method on the `Integer 10` object, passing the declared closure as a parameter. In other words, when `times` is executed, a reference to the closure object lies on the stack. The `times` method uses this reference to execute `Closure`'s `call` method, passing its local variable `count` to it. In this specific example, the `count` parameter is not used within `Closure.call`. Instead, `Closure.call` only works on the `x` reference that it holds to the local variable `x` in `Script`.

Through analysis, you can see that local variables are bound as a reference to the closure at declaration time.

5.5.2 The general closure scope

It would not be surprising if other scope elements were treated the same as local variables: the value of `this`, fields, methods, local variables, and parameters.

This generalization is correct, but the `this` reference is a special case. Inside a closure, you could legitimately assume that `this` would refer to the current object, which is the closure object itself. On the other hand, it should make no difference whether you use `this.`*`reference`* or plain *`reference`* for locally accessible references. Therefore, the Groovy team has decided that inside a closure `this` doesn't refer to the closure object but to its *owner*: the object in whose scope the closure was declared.

A reference to that *owner* object is held in a special variable called `owner` within the `Closure` class. Listing 5.8 extends the initial example to reveal the remaining scope elements.

We implement a small class `Mother` that should give `birth` to a closure through a method with that name. The class has a property and another method; the `birth` method has a parameter and a local variable that we can study. The closure should return a map of all elements that are in the current context (scope). Behind the scenes, these elements will be bound at declaration time but not evaluated until the closure is called. Let's investigate the result of such a call.

Listing 5.11 Investigating the closure scope

```
class Mother {
    int prop = 1
    int method() { return 2 }
    Closure birth (param) {                           //#1 This method creates and r
        def local = 3
        def closure = { caller ->
            [ self  : this,   prop : prop , method: method(),
              local : local,  param: param,
              caller: caller, owner: owner, delegate: delegate]
        }
        return closure
    }
}

Mother julia = new Mother()

def closure = julia.birth(4)                           //#2 Let a Mother give birth to

def context = closure.call(this)                       //#3 Call the closure

assert context.self     == julia                      //#4 'this' is owner
assert context.prop     == 1                          //|#5 No surprises?
assert context.method   == 2                          //|#5
assert context.local    == 3                          //|#5
assert context.param    == 4                          //|#5

assert context.owner    == julia                      //#6 Declaring object
assert context.caller   == this                       //#7 Calling object

firstClosure  = julia.birth(4)                        //#8 Closure brace are
secondClosure = julia.birth(4)                        //#8 like 'new'
assert firstClosure != secondClosure                  //#8
```

We've added the optional return type to the method declaration in ❶ to point out that this method returns a closure object. A method that returns a closure is not the most common usage of closures, but every now and then it comes in handy. Note that the method only *declares* the closure - it doesn't execute it. The map that the closure will return when called doesn't exist yet.

After constructing a new instance of `Mother`, we call its `birth` method at ❷ to retrieve a newly born closure object. Even now, the closure hasn't been called. The list of elements is not yet constructed.

Rubber meets road at ❸ . Now we call the closure using the explicit call syntax to make it stand out. The closure constructs its list of elements from what it remembers about its birth. We store that list in a variable for further inspection. Notice that we pass ourselves as a parameter into the closure in order to make it available inside the closure as the `caller`.

At ❹ we assert that inside the closure, `this` refers to the object that constructed it.

The property `prop`, the result of calling `method()`, the local variable `local`, and the parameter `param` all have the expected values, as demonstrated in ❺ . This is the birthday recall that we expected.

But `prop` and `foo()` are a bit more involved as they appear. Since they are not local references (local variables or parameters) and have no receiver specified (like ***`this`***`.foo()`), they are considered *vanilla references*[50]. Resolving a vanilla reference should *by default* work as if it was prefixed with `this.`*`reference`*. In other words, they are resolved against the `owner` ❻ . Since the owner is julia, everything works as if the enclosed code ran in the *birthday context* of the closure. We will see later that the resolution mechanism is very flexible and even allows vanilla references to be resolved against totally unrelated objects.

Footnote 50 References to *classes* are also vanilla references and if Groovy can resolve the reference name to a class on the classpath, then this resolution has precedence over any other possible resolution.

Passing the caller explicitly into the closure is the way to make it accessible inside. We demonstrate this at ❼ . Throughout all previous closure examples in this book, the *caller* and the *owner* have been identical. Therefore, we could easily apply side effects on it. You may have thought we were side-effecting the *caller* while we were working on the *owner*. If this sounds totally crazy to you, don't worry. Getting used to the scope available within a closure takes some time. It's worth revisiting a simple example like `times` and experimenting with it to get a better feel for what's happening. It'll become second nature soon enough.

At ❽ you can see that every call to `birth` constructs a *new* closure. Think of the curly braces in a closure declaration as if the word *new* appeared before them. Behind this observation is a fundamental difference between closures and methods: Methods are constructed exactly once at class-generation time. Closures are objects and thus constructed at runtime, and there may be any number of them constructed from the same lines of code if that code happens to be called multiple times.

Figure 5.4 shows who refers to whom in listing 5.8.

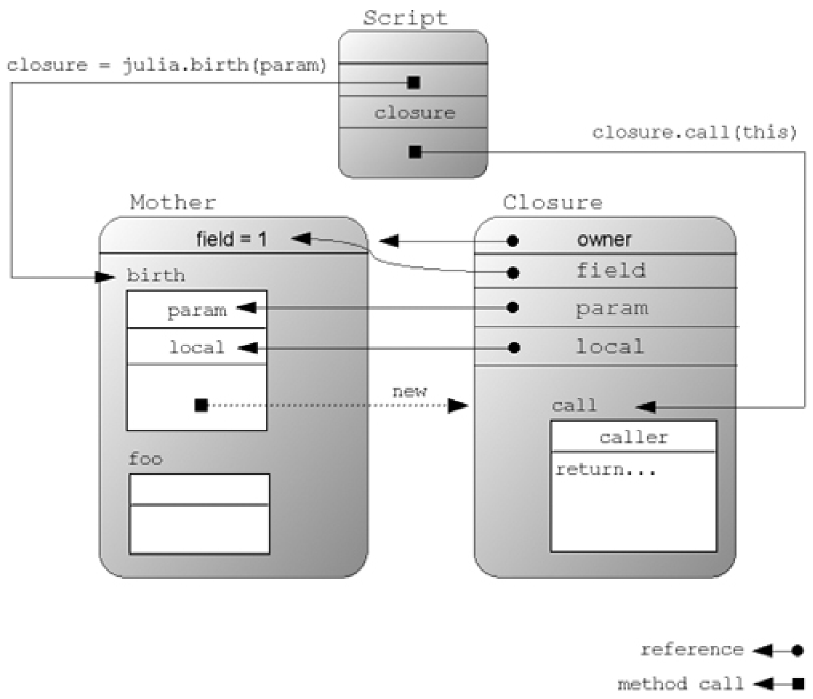


Figure 5.4 Conceptual view of object references and method calls for the general scoping example in listing 5.8, revealing the calls to the julia instance of Mother for creating a closure that is called in the trailing Script code to return all values in the current scope

This is all a bit unfamiliar for many developers but the net effect of this construction is that closures become fairly easy and straightforward to use. For the vast majority of cases, there is no need to think about the scoping rules too deeply.

There are use cases of closures, though, that are bit more involved. Listing 0.0 introduces us to the concept of a *delegate* that a closure can consider in addition to the owner for resolving vanilla references. The GDK provides a `with` method that takes a closure argument and resolves all vanilla references inside that closure against the receiver of `with`, which results in code that reads almost like Visual Basic.

Listing 5.12 Closure delegate for the `with` method

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

list.with {
    add 1
    add 2
    assert delegate == expected
}
```

The line `add 1` becomes a shorthand for `list.add 1` since inside the *with* closure, the `add` reference is resolved against `list`.

The `delegate` is a property of the closure itself and represents the receiver object that the `with` method was called on, while `expected` is resolved against our script--the owner of the closure.

Who is responsible for all this? Well, in the example above, it is the implementation of the `with` method that looks conceptually like this:

```
def with(Closure doit) {  // fake implementation
    doit.delegate = list
    doit()
}
```

Now we have three possible candidates to resolve a vanilla reference against:

- the closure itself (for `owner`, `delegate`)
- the owner (for `expected`)
- the delegate (for `add`)

There may be name clashes between those three and we need precedence rules for the lookup. The closure itself always comes first. What can be resolved against the closure always has precedence. The resolution rules between *owner* and *delegate* are determined by the closure's *resolveStrategy*, which are:

- `Closure.OWNER_FIRST`, which is the *default* and tries first the owner, then the delegate.
- `Closure.DELEGATE_FIRST`, which tries the delegate first, then the owner.
- `Closure.OWNER_ONLY`
- `Closure.DELEGATE_ONLY`
- `Closure.TO_SELF`, which neither looks at owner nor delegate but only the (meta-) methods and properties of `Closure`.

The `with` method actually sets the resolve strategy of their closure parameter to `DELEGATE_FIRST`. It is also a good example of the reason for having an adaptable resolution strategy: to allow simplified access to an existing API, which is an integral part of *domain specific languages* (see XREF ch_dsl) and builders (see XREF ch_builder).

Lectures about lexical scoping and closures from other languages such as Lisp, Smalltalk, Perl, Ruby, and Python typically end with some mind-boggling examples about variables with identical names, mutually overriding references, and mystic rebirth of supposed-to-be foregone contexts. These examples are like puzzles. They make for an entertaining pastime on a long winter evening, but they have no practical relevance. We will not provide any of those, because they can easily undermine your carefully built confidence in the scoping rules.

Our intention is to provide a reasonable introduction to Groovy's closures. This should give you the basic understanding that you need when hunting for more complex examples in mailing lists and on the Web. Instead of giving a deliberately obscure example, however, we *will* provide one that shows how closure scopes can make an otherwise complex task straightforward.

5.5.3 Scoping at work: the classic accumulator test

There is a classic example to compare the power of languages by the way they support closures. One of the things it highlights is the power of the scoping rules for those languages as they apply to closures. Paul Graham first proposed this test in his excellent article "Revenge of the Nerds" (http://www.paulgraham.com/icad.html). Beside the test, his article is an interesting and informative to read. It talks about the difference a language can make.

In some languages, this test leads to a brain-teasing solution. Not so in Groovy. The Groovy solution is exceptionally obvious and straightforward to achieve.

Here is the original requirement statement:

> "We want to write a function that generates accumulators--a function that takes a number `n`, and returns a function that takes another number `i` and returns `n` incremented by `i`".
>
> --

The following are proposed solutions for other languages:

In Lisp:

```
(defun foo (n) (lambda (i) (incf n i)))
```

In Perl 5:

```
sub foo { my ($n) = @_; sub {$n += shift} }
```

In Smalltalk:

```
foo: n |s| s := n. ^[:i| s := s+i. ]
```

The following steps lead to a Groovy solution, as shown in listing 5.9:

1. We need a function that returns a closure. In Groovy, we don't have functions, but methods. (Actually, we have not only methods, but also closures. But let's keep it simple.) We use `def` to declare such a method. It has only one line, which after creates a new closure and returns it. We will call this method `foo` to make the solutions comparable in size. The name `createAccumulator` would reflect the purpose more clearly, of course.
2. Our method takes an initial value `n` as required.
3. Because `n` is a parameter to the method that *declares* the closure, it gets bound to the closure scope. We can use it inside the closure body to calculate the incremented value.
4. The incremented value is not only calculated but also assigned to `n` as the new value. That way we have a true accumulation.

We add a few assertions to verify our solution and reveal how the accumulator is supposed to be used. Listing 5.9 shows the full code.

Listing 5.13 The accumulator problem in Groovy

```
def foo(n) {
    return { n += it }
}

def accumulator = foo(1)
assert accumulator(2) == 3
assert accumulator(1) == 4
```

All the steps that led to the solution are straightforward applications of what you've learned about closures.

In comparison to the other languages, the Groovy solution is not only short but also surprisingly clear. Groovy has passed this language test exceptionally well.

Is this test of any practical relevance? Maybe not in the sense that we would ever need an accumulator generator, but it is in a different sense. Passing this test means that the language is able to dynamically put logic in an object and manage the context that this object lives in. This is an indication of how powerful abstractions in that language can be.

5.6 Returning from closures

So far, you have seen how to declare closures and how to call them. However, there is one crucial topic that we haven't touched yet: how to return from a closure.

In principle, there are two ways of returning:

- The last expression of the closure has been evaluated, and the result of this evaluation is returned. This is called *end return*. Using the `return` keyword in front of the last expression is optional.
- The `return` keyword can also be used to return from the closure *prematurely*.

This means the following ways of doubling the entries of a list have the very same effect:

```
[1, 2, 3].collect { it * 2 }
[1, 2, 3].collect { return it * 2 }
```

A premature return can be used to, for example, double only the even entries:

```
[1, 2, 3].collect {
    if ( it % 2 == 0) return it * 2
    return it
}
```

This behavior of the `return` keyword inside closures is simple and straightforward. It sounds like it shouldn't cause any problems, but there is one aspect which often catches people out.

WARNING **Where to return from**
There is a difference between using the `return` keyword inside and outside of a closure.

Outside a closure, any occurrence of `return` leaves the current method. When used inside a closure, it only ends the current evaluation of the *closure*, which is a much more localized effect. For example, when using `List.each`, returning early from the closure doesn't return early from the `each` method--the closure will still be called again with the next element in the list.

As we progress further through the book, we will hit this issue again and explore more ways of dealing with it. Section 13.1.8 summarizes the topic.

5.7 Support for design patterns

Design patterns are widely used by developers to enhance the quality of their designs. Each design pattern presents a typical problem that occurs in object-oriented programming along with a corresponding well-tested solution. Let's take a closer look at the way the availability of closures affects how, which, and when patterns are used.

If you've never seen design patterns before, we suggest you look at the classic book Design Patterns: Elements of Reusable Object-Oriented Software by Gamma et al, or one of the more recent ones such as Head First Design Patterns by Freeman et al or Refactoring to Patterns by Joshua Kerievsky, or search for “patterns repository” or “patterns catalog” using your favorite search engine.

Although many design patterns are broadly applicable and apply to any language, some of them are particularly well-suited to solving issues that occur when using programming languages such as C++ and Java. Often these involve implementing new abstractions and new classes to make the original programs more flexible or maintainable. With Groovy, some of the restrictions that face C++ and Java do not apply, and the design patterns are either of less value or are directly supported using language features rather than introducing new classes. We'll pick two examples to show the difference: the *Visitor* and *Builder* patterns. As you'll see, closures and dynamic typing are the key differentiators in Groovy that facilitate easier pattern usage.

5.7.1 Relationship to the Visitor pattern

The Visitor pattern is particularly useful when you wish to perform some complex business functionality on a composite collection (such as a tree or list) of existing simple classes. Rather than altering the existing simple classes to contain the desired business functionality, a `Visitor` object is introduced. The `Visitor` knows how to traverse the composite collection and knows how to perform the business functionality for different kinds of a simple class. If the composite changes or the business functionality changes over time, typically only the `Visitor` class needs to change.

Listing 5.10 shows how simple the Visitor pattern can look in Groovy; the composite traversal code is in the `accept` method of the `Drawing` class, whereas the business functionality (in our case performing some calculations involving the area of a shape) is contained in two closures, which are passed as parameters to the appropriate `accept` methods. There is no need for a separate `Visitor` class in this simple case.

Listing 5.14 The Visitor pattern in Groovy

```
class Drawing {
    List shapes
    def accept(Closure yield) { shapes.each{it.accept(yield)} }
}
class Shape {
    def accept(Closure yield) { yield(this) }
}
class Square extends Shape {
    def width
    def area() { width**2 }
}
class Circle extends Shape {
    def radius
    def area() { Math.PI * radius**2 }
}

def picture = new Drawing(shapes: [new Square(width:1), new Circle(radius:1)])

def total = 0
picture.accept { total += it.area() }
println "The shapes in this drawing cover an area of $total units."
println 'The individual contributions are: '
picture.accept { println it.class.name + ":" + it.area() }
```

5.7.2 Relationship to the Builder pattern

The Builder pattern serves to encapsulate the logic associated with constructing a product from its constituent parts. When using the pattern, you normally create a `Builder` class, which contains logic determining what builder methods to call and in which sequence to call them to ensure proper assembly of the product. For each product, you must supply the appropriate logic for each relevant builder method used by the `Builder` class; each builder method typically returns one of the constituent parts.

Coding Java solutions based on the Builder pattern is not hard, but the Java code tends to be cumbersome and verbose and doesn't highlight the structure of the assembled product. For that reason, the Builder pattern is rarely used in Java; instead developers use unstructured or replicated builder-type logic mixed in with their other code. This is a shame, because the Builder pattern is so powerful.

Groovy's builders provide a solution using nested closures to conveniently specify even very complex products. Such a specification is easy to read, because the appearance of the code reflects the product structure. Groovy has built-in library classes based on the Builder pattern that allow you to easily build arbitrarily nested node structures, produce markup like HTML or XML, define GUIs in Swing or other widget toolkits, and even access the wide range of functionality in Ant. You will see lots of examples in chapter 8, and we explain how to write your own builders in section 8.6.

5.7.3 Relationship to other patterns

Almost all patterns are easier to implement in Groovy than in Java. This is often because Groovy supports more lightweight solutions that make the patterns less of a necessity--mostly because of closures and dynamic typing. In addition, when patterns are required, Groovy often makes expressing them more succinct and simpler to set up.

We discuss a number of patterns in other sections of this book, patterns such as Strategy (see 9.1.1 and 9.1.3), Observer (see 13.2.3), and Command (see 9.1.1) benefit from using closures instead of implementing new classes. Patterns such as Adapter and Decorator (see 7.5.3) benefit from dynamic typing and method lookup. We also briefly discuss patterns such as Template Method (see section 5.2.2), the Value Object pattern (see 3.3.2), the incomplete library class smell (see 7.5.3), MVC (see 8.5.6), and the DTO and DAO patterns (see chapter 10). Just by existing, closures can completely replace the Method Object pattern.

Groovy provides plenty of support for using patterns within your own programs. Its libraries embody pattern practices throughout. Higher-level frameworks such as Grails take it one step further. Grails provides you with a framework built on top of Groovy's libraries and patterns support. Because using such frameworks saves you from having to deal with many pattern issues directly--you just use the framework--you will automatically end up using patterns without needing to understand the details in most cases. Even then, it is useful to know about some of the patterns we have touched upon so that you can leverage the maximum benefit from whichever frameworks you use.

5.8 Summary

You have seen that closures follow our theme of *everything is an object*. They capture a piece of logic, making it possible to pass it around for execution, return it from a method call, or store it for later usage.

Closures encourage centralized resource handling, thus making your code more reliable. This doesn't come at any expense. In fact, the codebase is relieved from structural duplication, enhancing expressiveness and maintainability.

Defining and using closures is surprisingly simple because all the difficult tasks such as keeping track of references and relaying method calls back to the delegating owner are done transparently. If you don't care about the scoping rules, everything falls into place naturally. If you want to hook into the mechanics and perform tasks such as redirecting the calls to the delegate, you can. Of course, such an advanced usage needs more care. You also need to be careful when returning from a delegate, particularly when using one in a situation where in other languages you might use a `for` loop or a similar construct. This has surprised more than one new Groovy developer, although the behavior is logical when examined closely. Re-read section 5.6 when in doubt.

Closures open the door to several ways of doing things that may be new to many developers. Some of these, such as currying, can appear daunting at first sight but allow a great deal of power to be wielded with remarkably little code. Additionally, closures can make familiar design patterns simpler to use or even unnecessary.

Although you now have a good understanding of Groovy's datatypes and closures, you still need a way of controlling the flow of execution through your program. This is achieved with control structures, which form the topic of the next chapter.

6 Groovy control structures

The pursuit of truth and beauty is a sphere of activity in which we are permitted to remain children all our lives.

-- Albert Einstein

At the hardware level, computer systems use simple arithmetic and logical operations, such as jumping to a new location if a memory value equals zero. Any complex flow of logic executed by a computer can always be expressed in terms of these simple operations. Fortunately, languages such as Java raise the abstraction level available in programs we write so that we can express the flow of logic in terms of higher-level constructs--for example, looping through all of the elements in an array or processing characters until we reach the end of a file.

In this chapter, we explore the constructs Groovy provides to describe logic flow in ways that are even simpler and more expressive than Java. Before we look at the constructs themselves, however, we have to examine Groovy's answer to that age-old philosophical question: What is truth? [51]

Footnote 51 Groovy has no opinion as to what beauty is. We're sure that if it did, however, it would involve expressive minimalism. Closures too, probably.

6.1 The Groovy truth

In order to understand how Groovy will handle control structures such as `if` and `while`, you need to know how it evaluates boolean expressions. Many of the control structures we examine in this chapter rely on the result of a *boolean test* --an expression that is first evaluated and then considered as being either true or false. The outcome of this determines which path is then followed in the code. In Java, the consideration involved is usually trivial, because Java requires the expression to be one resulting in the primitive `boolean` type to start with. Groovy is more relaxed about this, allowing simpler code at the slight expense of language simplicity. We'll examine Groovy's rules for boolean tests and give some advice to avoid falling into an age-old trap.

6.1.1 Evaluating boolean tests

The expression of a boolean test can be of any (non-void) type. It can apply to any object. Groovy decides whether to consider the expression as being true or false by asking the object for the result of its `asBoolean()` method.

You can implement this method yourself or rely on the default implementations that Groovy provides as part of the GDK. Table 6.1 summarises these defaults; you can also easily look up the details in

```
org.codehaus.groovy.runtime.DefaultGroovyMethods
```

Look for the various `asBoolean(obj)` implementations.

Note that choosing the appropriate `asBoolean` implementation is subject to the standard Groovy method dispatch rules. The method can be inherited, it can be overridden, and it can be modified by the means of metaprogramming (which we will encounter later).

Table 1.1 Default implementations of asBoolean() that implement Groovy's meaning of truth.

Runtime type	Evaluation criterion required for truth
`java.lang.Object`	The object reference is non-null
`java.lang.Boolean`	Corresponding Boolean value is true
`java.util.regex.Matcher`	The matcher has a match
`Collections, Arrays`	The collection is non-empty
`Iterator, Enumeration`	There are more elements
`java.util.Map`	The map is non-empty
`CharSequence, String, GString`	The sequence is non-empty
`Number, Character`	The value is nonzero

Listing 6.1 shows these rules in action, using the boolean negation operator `!` to assert that expressions which ought to evaluate to `false` really do so.

```
assert true                //|#1 Boolean values
assert !false              //|#1 are trivial

assert ('a' =~ /./)        //|#2 Matcher must
assert !('a' =~ /b/)       //|#2 match

assert [1]                 //|#3 Collections must
assert ![]                 //|#3 be non-empty

Iterator iter = [1].iterator()
assert iter               //|#4 Iterators
```

```
iter.next()              //|#4 must have
assert ! iter            //|#4 next element

assert ['a':1]            //|#5 Maps must be
assert ![:]               //|#5 non-empty

assert 'a'                //|#6 Strings must be
assert !''                //|#6 non-empty

assert 1                  //|#7 Numbers
assert 1.1                //|#7 (any type)
assert 1.2f               //|#7 must be
assert 1.3g               //|#7 non-zero
assert 2L                 //|#7
assert 3G                 //|#7
assert !0                 //|#7

assert ! null              //|#8 Objects must be
assert new Object()        //|#8 non-null

class AlwaysFalse {
    boolean asBoolean() { false } //#9 Custom truth
}
assert ! new AlwaysFalse() //#10 Calls asBoolean()
```

The observant reader may spot some commonality between the `asBoolean` and `isCase` method since both allow objects to specify their behavior in Groovy control structures. This is entirely deliberate!

In fact, the `isCase` method also plays a role in boolean tests as it allows objects to specify their behavior with respect to the *in* operator. That is, in a statement like

```
if (candidate in classifier) { ... }
```

the *if* path will be executed if

```
classifier.isCase(candidate) == true
```

Since `isCase` is available for many types in Groovy, this allows for elegant usages like these:

```
assert 1 in [0, 1, 2]        // list
assert 1 in 0..3             // range
assert 'Hello' in String     // class
assert 'Hello' in ~/H.*/     // pattern
```

We will see more examples of `isCase` when talking about the *switch* control structure.

Evaluating objects as boolean values can make testing for “truth” simpler and easier to read. However, they come with a price, as you're about to discover.

6.1.2 Assignments within boolean tests

Before we get into the meat of the chapter, we have a warning to point out. Just like Java, Groovy allows the expression used for a boolean test to be an assignment--and the value of an assignment expression is the value assigned. Unlike Java, the type of a boolean test is not restricted to `booleans`, which means that a problem you might have thought was ancient history reappears, albeit in an alleviated manner. Namely, an equality operator `==` incorrectly entered as an assignment operator `=` is valid code with a drastically different effect than the intended one. Groovy shields you from falling into this trap for the most common appearance of this error: when it's used as a top-level expression in an `if` statement. However, it can still arise in less common cases.

Listing 6.2 leads you through some typical variations of this topic.

```
def x = 1

if (x == 2) {           //#1 Normal comparison
    assert false
}
/*******************
if (x =  2) {           //#2 Not allowed! Compiler error!
   println x
}
********************/
if ((x = 3)) {          //#3 Assign and test in nested expression
    println x
}
assert x == 3

def store = []
while (x = x - 1) {     //#4 Deliberate assign and test in while
    store << x
}
assert store == [2, 1]

while (x =  2) {        //#5 Ouch! This will print 2!
    println x
    break
}
```

The equality comparison in ❶ is fine and would be valid in Java. In ❷ , an equality comparison was intended, but one of the equal signs was left out. This raises a Groovy compiler error, because an assignment is not allowed as a top-level expression in an `if` test.

However, boolean tests can be nested inside expressions in arbitrary depth; the simplest one is shown at ❸ , where extra parentheses around the assignment make it a subexpression, and therefore the assignment becomes compliant with the Groovy language. The value `3` will be assigned to `x`, and `x` will be tested for truth. Because `3` is considered `true`, the value `3` gets printed. This use of parentheses to please the compiler can even be used as a trick to spare an extra line of assignment. The unusual appearance of the extra parentheses then serves as a warning sign for the reader.

The restriction of assignments from being used in top-level boolean expressions applies only to `if` and not to other control structures such as `while`. This is because doing assignment and testing in one expression are often used with `while` in the style shown at ❹ . This style tends to appear with classical usages like processing tokens retrieved from a parser or reading data from a stream. Although this is convenient, it leaves us with the potential coding pitfall shown at ❺ , where `x` is assigned the value `1` and the loop would never stop if there weren't a break statement. [52]

Footnote 52 Remember that the code in this book has been executed. If we didn't have the `break` statement, the book would have taken literally forever to produce.

This potential cause of bugs has given rise to the idiom in other languages (such as C and C++, which suffer from the same problem to a worse degree) of putting constants on the left side of the equality operator when you wish to perform a comparison with one. This would lead to the last `while` statement in the previous listing (still with a typo) being

```
while(1 =  x) { // should be ==
    println x
}
```

This would raise an error, as you can't assign a value to a constant. We're back to safety--so long as constants are involved. Unfortunately, not only does this fail when both sides of the comparison are variables, it also reduces readability. Whether it is a natural occurrence, a quirk of human languages, or conditioning, most people find `while (x==3)` significantly simpler to read than `while (3==x)`. Although neither is going to cause confusion, the latter tends to slow people down or interrupt their train of thought. In this book, we have favored readability over safety--but our situation is somewhat different than that of normal development. You will have to decide for yourself which convention suits you and your team better.

Now that we have examined which expressions Groovy will consider to be true and which are false, we can start looking at the control structures themselves.

6.2 Conditional execution structures

Our first set of control structures deals with conditional execution. They all evaluate a boolean test and make a choice about what to do next based on whether the result was true or false. None of these structures should come as a completely new experience to any Java developer, but of course Groovy adds some twists of its own. We will cover `if` statements, the conditional operator, `switch` statements, and assertions.

6.2.1 The humble if statement

Our first two structures act *exactly* the same way in Groovy as they do in Java, apart from the evaluation of the boolean test itself. We start with `if` and `if/else` statements.

Just as in Java, the boolean test expression must be enclosed in parentheses. The conditional block is normally enclosed in curly braces. These braces are optional if the block consists of only one statement. [53]

Footnote 53 Even though the braces are optional, many coding conventions insist on them in order to avoid errors that can occur through careless modification when they're not used.

A special application of the “no braces needed for single statements” rule is the sequential use of `else if`. In this case, the logical indentation of the code is often flattened; that is, all `else if` lines have the same indentation although their meaning is nested. The indentation makes no difference to Groovy and is only of aesthetic relevance.

Listing 6.3 gives some examples, using `assert true` to show the blocks of code that will be executed and `assert false` to show the blocks that won't be executed.

There should be no surprises in the listing, although it might still look slightly odd to you that non-boolean expressions such as strings and lists can be used for boolean tests. Don't worry--it becomes natural over time.

```
if (true)         assert true
```

```
else             assert false

if (1) {
    assert true
} else {
    assert false
}

if ('non-empty') assert true
else if (['x'])  assert false
else             assert false

if (0)           assert false
else if ([])     assert false
else             assert true
```

Finally, there is a notable speciality of *if* in Groovy: it plays well with the optional *return* statement. If your last expression of a method or closure is an *if* statement, then it is evaluated like an expression. Note, how the following code needs no return statements:

```
def mac() {
    if (System.properties.'os.name'.contains('Mac'))
        "We're on Mac." // no 'return'
    else
        "Oh, well ..."  // no 'return'
}
println mac()
```

6.2.2 The conditional ?: operator

Groovy also supports the ternary conditional `?:` operator for small inline tests, as shown in listing 6.4. This operator returns the object that results from evaluating the expression to the left or right of the colon, depending on the test before the question mark. If the first expression evaluates to `true`, the middle expression is evaluated. Otherwise, the last expression is evaluated. Just as in Java, whichever of the last two expressions isn't used as the result isn't evaluated at all.

```
def result = (1==1) ? 'ok' : 'failed'
assert result == 'ok'

result = 'some string' ? 10 : ['x']
assert result == 10
```

Again, notice how the boolean test (the first expression) can be of any type. Also note that because everything is an object in Groovy, the middle and last expressions can be of radically different types.

Opinions about the ternary conditional operator vary wildly. Some people find it extremely convenient and use it often. Others find it too Perl-ish. You may well find that you use it less often in Groovy because there are features that make its typical applications obsolete--for example, GStrings (covered in section 3.4.2) allow the dynamic creation of strings that would be constructed in Java using the conditional operator.

A common use case for the ternary conditional operator is checking whether a reference is has a non-empty value, and using a default otherwise:

```
value ? value : default
```

This code has two issues: first, it contains duplication; second, if accessing the value is expensive, doing it twice may take to long. That's why Groovy has introduced the *Elvis operator* to achieve the above as:

```
value ?: default
```

If you turn your head sideways and look at the operator as a smiley emoticon, you will recognize "The King". Elvis makes sense in Groovy, since we have the Groovy truth, which allows us to use *value* as an object and inside a boolean test at the same time. This feature was proposed for Java 7 but then withdrawn when people realized it makes no sense if you can only assign boolean values.

So far, so Java-like. Things change significantly when we consider `switch` statements.

6.2.3 The switch statement

On a recent train ride, I (Dierk) spoke with a teammate about Groovy, mentioning the oh-so-cool `switch` capabilities. He wouldn't even let me get started, waving his hands and saying, “I never use `switch`!” I was put off at first, because I lost my momentum in the discussion; but after more thought, I agreed that I don't use it either-- *in Java*.

The `switch` statement in Java is very restrictive. You can only switch on an `int` type, with `byte`, `char`, and `short` automatically being promoted to `int`.[54] With this restriction, its applicability is bound to either low-level tasks or to some kind of dispatching on a *type code*. In object-oriented languages, the use of type codes is considered smelly.[55]

Footnote 54 As of Java 5, `enum` types can also be switched on, due to some compiler trickery, and with Java 7 we may be able to switch on Strings.

Footnote 55 See “Replace Conditional with Polymorphism” in Refactoring by Martin Fowler (Addison Wesley, 2000).

THE SWITCH STRUCTURE

The general appearance of the `switch` construct is just like in Java, and its logic is identical in the sense that the handling logic falls through to the next `case` unless it is exited explicitly. We will explore exiting options in section 6.4.

Listing 6.5 shows the general appearance.

```
def a = 1
def log = ''
switch (a) {
    case 0  : log += '0' //|#1 fall
    case 1  : log += '1' //|#1 through
    case 2  : log += '2' ; break
    default : log += 'default'
}
assert log == '12'
```

Although the *fallthrough* is supported in Groovy, there are few cases where this feature really enhances the readability of the code. It usually does more harm than good (and this applies to Java, too). As a general rule, putting a break at the end of each case is good style.

Just like the *if* statement in Groovy, *switch* can be used without *return* if it is the last statement of a method or closure:

```
def mac() {
    switch(System.properties.'os.name') {
        case 'Mac OS X': "We're on Mac."; break // no 'return'
        default:         "Oh, well ..."         // no 'return'
    }
}
println mac()
```

SWITCH WITH CLASSIFIERS

You have seen the Groovy `switch` used for classification in section 3.5.5 and when working through the datatypes. A *classifier* is eligible as a `switch` case if it implements the `isCase` method. In other words, a Groovy `switch` like

```
switch (candidate) {
    case classifier1: handle1() ; break
    case classifier2: handle2() ; break
    default: handleDefault() }
```

is roughly equivalent (beside the fallthrough and exit handling) to

```
if (classifier1.isCase(candidate)) handle1()
else if (classifier2.isCase (candidate)) handle2()
else handleDefault()
```

This allows expressive classifications and even some unconventional usages with mixed classifiers. Unlike Java's constant cases, the candidate may match more than one classifier. This means that the order of cases is important in Groovy, whereas it cannot affect behavior in Java. Listing 6.6 gives an example of multiple types of classifiers. After checking that our number `10` is not zero, not in range `0..9`, not in list `[8,9,11]`, not of type `Float`, and not an integral multiple of `3` , we finally find it to be made of two characters.

```
switch (10) {
    case 0          : assert false ; break
    case 0..9       : assert false ; break
    case [8,9,11]   : assert false ; break
    case Float      : assert false ; break     //#1 Class case
    case {it%3 == 0}: assert false ; break     //#2 Closure case
    case ~/../      : assert true  ; break     //#3 Pattern case
    default         : assert false ; break
}
```

The new feature in ❶ is that we can classify by type. `Float` is of type `java.lang.Class`, and the GDK enhances `Class` by adding an `isCase` method that tests the candidate with `isInstance`.

The `isCase` method on closures at ❷ passes the candidate into the closure and returns the result of the closure call coerced to a `Boolean`.

The final classification ❸ as a two-digit number works because `~/../` is a `Pattern` and the `isCase` method on patterns applies its test to the `toString` value of the argument.

In order to leverage the power of the `switch` construct, it is essential to know the available `isCase` implementations. It is not possible to provide an exhaustive list, because any custom type in your code or in a library can implement it, but table 6.2 has the list of known implementations in the GDK.

Table 1.2 Implementations of isCase for switch

Class	**`a.isCase(b)` implemented as**
`Object`	`a.equals(b)`
`Class`	`a.isInstance(b)`
`Collection`	`a.contains(b)`
`Range`	`a.contains(b)`
`Pattern`	`a.matcher(b.toString()).matches()`
`String`	`(a==null && b==null) || a.equals(b)`
`Closure`	`a.call(b)`

NOTE **Recall**
The `isCase` method is also used with `grep` on collections such that *`collection`* `.grep(` *`classifier`* `)` returns a collection of all items that are a case of that classifier.

The same logic applies when using the *in* operator in boolean tests.

Using the Groovy `switch` in the sense of a classifier is a big step forward. It adds much to the readability of the code. The reader sees a simple classification instead of a tangled, nested construction of `if` statements. Again, you are able to reveal *what* the code does rather than *how* it does it.

As pointed out in section 4.1.2, the switch classification on ranges is particularly convenient for modeling business rules that tend to prefer discrete classification to continuous functions. The resulting code reads almost like a specification.

It's worth actively looking through your code for places to implement `isCase`. A characteristic sign of looming classifiers is lengthy `else if` constructions.

NOTE **Advanced Topic**
It is possible to overload the `isCase` method to support different kinds of classification logic depending on the type of the candidate. For example, if you provide both `isCase(String candidate)` and `isCase(Integer candidate)`, then `switch ('1')` can behave differently than `switch(1)` with your object as a classifier.

Our next topic, *assertions,* may not look particularly important at first glance. However, although assertions don't change the business capabilities of the code, they do make the code more robust in production. Moreover, they do something even better: enhance the development team's confidence in their code as well as their ability to remain agile during additional enhancements and ongoing maintenance.

6.2.4 Sanity checking with assertions

This book contains several hundred assertion statements--and indeed, you've already seen a number of them. Now it's time to go into some extra detail. We will look at producing meaningful error messages from failed assertions, reflect on reasonable uses of this keyword, and show how to use it for inline unit tests. We will also quickly compare the Groovy solution to Java's `assert` keyword and assertions as used in unit test cases.

PRODUCING INFORMATIVE FAILURE MESSAGES

When an assertion fails, it produces a stacktrace and a message. Put the code

```
def a = 1
assert a == 2
```

in a file called FailingAssert.groovy, and let it run via

```
> groovy FailingAssert
```

It is expected to fail, and it does so with the message

```
Caught: Assertion failed:

assert a == 2
       | |
       1 false

 at FailingAssert.run(FailingAssert.groovy:2)
```

You can see that when it fails, the assertion prints out the failed expression as it appears in the code plus the value of the variables in that expression together with the value of all subexpressions. The trailing stack-trace reveals the location of the failed assertion and the sequence of method calls that led to the error; as of Groovy 1.7 the stack trace is "sanitized" to skip over Groovy internals.

This is a exactly the kind of information that is needed to locate and understand the error in most cases, but not always. Let's try another example that tries to protect file-reading code from being executed if the file doesn't exist or cannot be read. [56]

Footnote 56 Perl programmers will see the analogy to `or die`.

```
try {
    input = new File('no such file')
    assert input.exists()
    assert input.canRead()
    println input.text
} catch (AssertionError error) {
    assert "n" + error.message == '''
assert input.exists()
       |     |
       |     false
       no such file'''
}
```

This error message shows that the file named "no such file" was not available. That is a good first indication but often, we need more information like the directory where the file was expected or the absolute path that was used. We can give the assertion a second argument to reveal this information in case of failure.

```
try {
    input = new File('no such file')
    assert input.exists(), "cannot find: $input.absolutePath"
    assert input.canRead()
    println input.text
} catch (AssertionError error) {
    assert error.message ==
'cannot find: ' +
'/projects/trunk/groovy-book/listings/chap06/no such file.' +
'Expression: input.exists()'
}
```

This is the information we need. However, assertions are not always needed.

Given that without the assertion, we run into a meaningful exception anyway, we can also leave them out:

```
def input = new File('no such file')
println input.text
```

The result is the following error message that may often be sufficient:

```
FileNotFoundException: no such file (The system cannot find the file specified)
```

This leads to the following best practices with assertions:

- Before writing an assertion, let your code fail, and see whether any other thrown exception is good enough.
- When you write an assertion, let it fail once to see whether the default message is sufficient. If it isn't, specify a more useful one then let it fail again to check that it now gives you all the information you'd need in a production situation.
- If you feel you need an assertion to clarify or protect your code, add it regardless of the previous rules.
- If you feel you need a message to clarify the meaning or purpose of your assertion, add it regardless of the previous rules.

INSURE CODE WITH INLINE UNIT TESTS

Finally, there is a potentially controversial use of assertions as unit tests that live right inside production code and get executed with it. Listing 6.7 shows this strategy with a nontrivial regular expression that extracts a hostname from a URL. The pattern is first constructed and then applied to some assertions before being put to action. We also implement a simple method `assertHost` to make it easy to assert a match grouping. [57]

Footnote 57 Please note that we're only using regexes to show the value of assertions. If we needed to find the hostname of a URL in real code, we would use `candidate.toURL().host`.

```
def host = ///([a-zA-Z0-9-]+(.[a-zA-Z0-9-])*?)(:|/)/ //#1 Regular expression matching host

assertHost 'http://a.b.c:8080/bla',     host, 'a.b.c'
assertHost 'http://a.b.c/bla',          host, 'a.b.c'
assertHost 'http://127.0.0.1:8080/bla', host, '127.0.0.1'
assertHost 'http://t-online.de/bla',    host, 't-online.de'
assertHost 'http://T-online.de/bla',    host, 'T-online.de'

def assertHost (candidate, regex, expected){
    candidate.eachMatch(regex){ assert it[1] == expected }
}
```

```
//#2 Trailing code goes here
```

Reading this code with and without assertions, their value becomes obvious. Seeing the example matches in the assertions reveals what the code is doing and verifies our assumptions at the same time. Traditionally, these examples would live inside a test harness or perhaps only within a comment. This is better than nothing, but experience shows that comments go out of date and the reader cannot *really* be sure that the code works as indicated. Tests in external test harnesses also often drift away from the code. Some tests break, they are commented out of a test suite under the pressures of meeting schedules, and eventually they are no longer run at all.

Some developers may fear the impact on performance of this style of inline unit tests. The best answer is to use a profiler and investigate where performance is really relevant. Our assertions in listing 6.7 run in a few milliseconds and should not normally be an issue. When performance is important, one possibility would be to put inline unit tests where they are executed only once per loaded class: in a static initializer.

RELATIONSHIPS TO OTHER ASSERTIONS

Java has had an `assert` keyword since JDK 1.4. It differs from Groovy assertions in that it has a slightly different syntax (colon instead of comma to separate the boolean test from the message) and that it can be enabled and disabled--and assertions are even disabled by default! Java's assertion feature is not as powerful, because it only works on a Java boolean test, whereas the Groovy assert takes a full Groovy conditional (see section 6.1).

The JDK documentation has a long chapter on assertions that talks about the disabling feature for assertions and its impact on compiling, starting the VM, and resulting design issues. Although this is fine and the design rationale behind Java assertions is clear, we feel that the disabling feature is the biggest stumbling block for using assertions in Java. You can never be sure that your assertions are really executed.

Some people claim that for performance reasons, assertions should be disabled in production, after the code has been tested with assertions enabled. On this issue, Bertrand Meyer, [58] the father of *design by contract*, pointed out that it is like learning to swim with a swimming belt and then taking it off when leaving the pool and heading for the ocean.

Footnote 58 See Object Oriented Software Construction, 2nd ed., by Bertrand Meyer (Prentice Hall, 1997).

In Groovy, your assertions are always executed.

Assertions also play a central role in unit tests. Groovy comes with a version of JUnit included automatically. JUnit makes a lot of specialized assertions available to its `TestCases`, and Groovy adds even more of them. Full coverage of these assertions is given in chapter 14. The information that Groovy provides when assertions fail makes them very convenient when writing unit tests, because it relieves the tester from writing lots of messages.

Assertions can make a big difference to your personal programming style and even more to the culture of a development team, regardless of whether they are used inline or in separated unit tests. Asserting your assumptions not only makes your code more reliable, but it also makes it easier to understand and easier to work with.

That's it for conditional execution structures. They are the basis for every kind of logical branching and a prerequisite to allow looping--the language feature that makes your computer do all the repetitive work for you. The next two sections cover the `while` and `for` looping structures.

6.3 Looping

The structures you've seen so far have evaluated a boolean test *once* and changed the path of execution based on the result of the condition. Looping, on the other hand, repeats the execution of a block of code multiple times. The loops available in Groovy are `while` and `for`, both of which we cover here.

6.3.1 Looping with while

The `while` construct is like its Java counterpart. The only difference is the one you've seen already--the power of Groovy boolean test expressions. To summarize very briefly, the boolean test is evaluated, and if it's true, the body of the loop is then executed. The test is then re-evaluated, and so forth. Only when the test becomes false does control proceed past the `while` loop. Listing 6.8 shows an example that removes all entries from a list. We visited this problem in chapter 3, where you discovered that you can't use `each` for that purpose. The second example adds the values again in a one-liner body without the optional braces.

Listing 6.8 Example while loops

```
def list = [1, 2, 3]
while (list) {
    list.remove(0)
}
assert list == []

while (list.size() < 3) list << list.size() + 1
assert list == [1, 2, 3]
```

Again, there should be no surprises in this code, with the exception of using just `list` as the boolean test in the first loop.

Note that there is no `do {} while(`***`condition`***`)` or `repeat {} until (`***`condition`***`)` construct in Groovy.

6.3.2 Looping with for

Considering it is probably the most commonly used type of loop, the `for` loop in Java is relatively hard to use, when you examine it closely. Through familiarity, people who have used a language with a similar structure (and there are many such languages) grow to find it easy to use, but that is solely due to frequent use, not due to good design. Although the nature of the traditional `for` loop is powerful, it is rarely used in a way that can't be more simply expressed in terms of iterating through a collection-like data structure. Groovy embraces this simplicity, leading to probably the biggest difference in control structures between Java and Groovy.

Groovy `for` loops follow this structure:

```
for (variable in iterable ) {
    body
}
```

where *variable* may optionally have a declared type. The Groovy `for` loop iterates over the *iterable*. Frequently used iterables are ranges, collections, maps, arrays, iterators, and enumerations. In fact, any object can be an iterable. Groovy applies the same logic as for *object iteration*, described in chapter 9.

Curly braces around the body are optional if it consists of only one statement. Listing 6.9 shows some of the possible combinations.

Listing 6.9 Multiple `for` loop examples

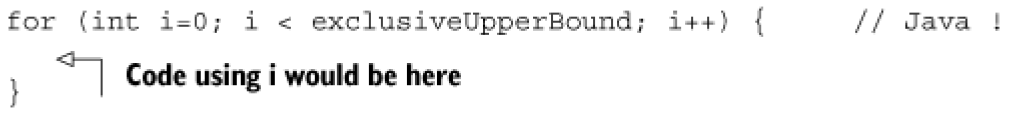
```
def store = ''                               //|#1 Statically typed, over String
for (String i in 'a'..'c') store += i        //|#1 range, no braces
assert store == 'abc'                        //|#1

store = ''                                   //|#2 Dynamically typed, over
for (i in [1, 2, 3]) {                       //|#2 list as collection,
    store += i                               //|#2 braces
}                                            //|#2
```

```
assert store == '123'                     //|#2

def myString = 'Equivalent to Java'       //|#3 Dynamically typed,
store = ''                                //|#3 over half-exclusive
for (i in 0 ..< myString.size()) {        //|#3 IntRange, braces
    store += myString[i]                  //|#3
}                                         //|#3
assert store == myString                  //|#3

store = ''                                //|#4 Dynamically typed,
for (i in myString) {                     //|#4 over String as collection,
    store += i                            //|#4 braces
}                                         //|#4
assert store == myString                  //|#4
```

Example ❶ uses explicit typing for `i` and no braces for a loop body of a single statement. The looping is done on a range of strings.

The usual `for` loop appearance when working on a collection is shown in ❷. Recall that thanks to the *autoboxing*, this also works for arrays.

Looping on a half-exclusive integer range as shown in ❸ is equivalent to the Java construction



```
for (int i=0; i < exclusiveUpperBound; i++) {      // Java !
}
```

Code using i would be here

Figure 6.1

which is referred to as the classic `for` loop, which is also available in Groovy. The remaining difference is that Groovy does not allow the Java style of multiple, comma-separated initialization and increment statements.

```
// Java and Groovy alike:
for (int i; i < 10; i++) { /* ... */ }
// Java, but NOT Groovy:
for (int i, int j; i < 10; i++, j++) { /* ... */ }
```

Example ❹ is provided to make it clear that ❸ is not the typical Groovy style of code for working on strings. It is more Groovy to treat a string as a collection of characters.

Using the `for` loop with *object iteration* as described in section 9.1.3 provides some very powerful combinations.

You can use it to print a file line-by-line via

```
def file = new File('myFileName.txt')
for (line in file) println line
```

or to print all one-digit matches of a regular expression:

```
def matcher = '12xy3'=~/\d/
for (match in matcher) println match
```

If the *iterable* object is null, no iteration will occur:

```
for (x in null) println 'This will not be printed!'
```

If Groovy cannot make the *iterable* object iterate by any means, the fallback solution is to do an iteration that contains only the *iterable* object itself:

```
for (x in new Object()) println "Printed once for object $x"
```

Object iteration makes the Groovy `for` loop a sophisticated control structure. It is a valid counterpart to using methods that iterate over an object with closures, such as using `Collection`'s `each` method.

The main difference is that the body of a `for` loop is not a closure! That means this body is a block:

```
for (x in 0..9) { println x }
```

whereas this body is a closure:

```
(0..9).each { println it }
```

Even though they look similar, they are very different in construction.

A closure is an object of its own and has all the features that you saw in chapter 5. It can be constructed in a different place and passed to the `each` method.

The body of the `for` loop, in contrast, is directly generated as bytecode at its point of appearance. No special scoping rules apply.

This distinction is even more important when it comes to managing exit handling from the body, as we'll see in the next section.

6.4 Exiting blocks and methods

Although it's nice to have code that reads as a simple list of instructions with no jumping around, it's often vital that control is passed from the current block or method to the enclosing block or the calling method--or sometimes even further up the call stack. Just like in Java, Groovy allows this to happen in an expected, orderly fashion with `return`, `break`, and `continue` statements, and in emergency situations with exceptions. Let's take a closer look.

6.4.1 Normal termination: return/break/continue

The general logic of `return`, `break`, and `continue` is similar to Java. One difference is that the `return` keyword is optional for the last expression in a method or closure. If it is omitted, the return value is that of the last expression. Methods with an explicit return type of `void` do not return a value, whereas closures always return a value. [59]

Footnote 59 You may be wondering what happes if the last evaluated expression of a closure is a void method call. In this case, the closure returns `null`.

Listing 6.10 shows how the current loop is shortcut with `continue` and prematurely ended with `break`. Like Java, there is an optional `label`.

```
def a = 1
while (true) {           //#1 Do forever
    a++
    break                //#2 Forever is over now
}
assert a == 2

for (i in 0..10) {
    if (i==0)  continue //#3 Proceed with 1
    a++
    if (i > 0) break    //#4 Premature loop end
}
assert a==3
```

Using `break` and `continue` is sometimes considered smelly. However, they can be useful for controlling the workflow in services that run in an endless loop, or to break apart complex conditional logic, such as this:

```
for(i in whatever){
    if (filterA) continue // skip if filter matches
    if (conditionB) break // exit loop if condition matches
    // normal case here
}
```

Similarly, returning from multiple points in the method is frowned upon in some circles, but other people find it can greatly increase the readability of methods that might be able to return a result early. We encourage you to figure out what you find most readable and discuss it with whoever else is going to be reading your code--consistency is as important as anything else.

As a final note on return handling, remember that when closures are used with iteration methods such as `each`, a `return` statement within the closure returns from the closure rather than the method, as explained in section 5.6. (Yes, we know we've mentioned it several times already... but we'd be surprised if you didn't get caught out by it at least once.)

6.4.2 Exceptions: throw/try-catch-finally

Exception handling is exactly the same as in Java and follows the same logic. Just as in Java, you can specify a complete `try-catch-finally` sequence of blocks, or just `try-catch`, or just `try-finally`. Note that unlike various other control structures, braces are required around the block bodies whether or not they contain more than one statement. The only difference between Java and Groovy in terms of exceptions is that declarations of exceptions in the method signature are optional, even for checked exceptions. Listing 6.11 shows the usual behavior.

```
def myMethod() {
    throw new IllegalArgumentException()
}

def log = []
try {
    myMethod()
} catch (Exception e) {
    log << e.toString()
} finally {
    log << 'finally'
}
assert log.size() == 2
```

In accordance with optional typing in the rest of Groovy, a type declaration is optional in the `catch` expression. And like in Java, you can declare multiple catches.

There are no compile-time or runtime warnings from Groovy when checked exceptions are not declared. When a checked exception is not handled, it is propagated up the execution stack like a `RuntimeException` in Java.

We cover integration between Java and Groovy in more detail in chapter 11; however, it is worth noting an issue relating to exceptions here. When using a Groovy class from Java, you need to be careful--the Groovy methods will not declare that they throw any checked exceptions unless you've explicitly added the declaration, even though they might throw checked exceptions at runtime.

Unfortunately, the Java compiler attempts to be clever and will complain if you try to catch a checked exception in Java when it believes there's no way that the exception can be thrown. If you run into this and need to explicitly catch a checked exception generated in Groovy code, you may need to add a `throws` declaration to the Groovy code, just to keep `javac` happy.

6.5 Summary

This chapter was our tour through Groovy's control structures: conditionally executing code, looping, and exiting blocks and methods early. We haven't seen any big surprises: everything turned out to be like Java, enriched with a bit of Groovy flavor. The only structural difference is the `for` loop. Exception handling is very similar to Java, except without the requirement to declare checked exceptions. [60]

Footnote 60 Checked exceptions are regarded by many as an experiment that was worth performing but which proved not to be as useful as had been hoped.

Groovy's handling of boolean tests is consistent between conditional execution structures and loops. We examined the differences between Java and Groovy in determining when a boolean test is considered to be true. This is a crucial area to understand, because idiomatic Groovy will often use tests that are not simple boolean expressions.

The `switch` keyword, the *in* operator and their use as a general classifiers bring a new object-oriented quality to conditionals. The interplay with the `isCase` method allows objects to control how they are treated inside that conditional. Although the use of `switch` is often discouraged in object-oriented languages, the new power given to it by Groovy gives it a new sense of purpose.

In the overall picture, *assertions* find their place as the bread-and-butter tool for the mindful developer. They belong in the toolbox of every programmer who cares about their craft.

With what you learned in the tour, you have all the means to do any kind of procedural programming. Of course, you should have higher goals and want to master object-oriented programming. The next chapter will teach you how.

8 Dynamic Programming with Groovy

"Until real software engineering is developed, the next best practice is to develop with a dynamic system that has extreme late binding in all aspects."

- Alan Kay

This chapter covers

- How Groovy supports dynamic programming
- Explanation of the Meta-Object-Protocol (MOP)
- How to utilize the MOP for your own purposes

We're going to start our journey with some general considerations about dynamic programming, how it differs from conventional object-oriented approaches, and why we want to have it in our toolbox. We will experience how the Meta-Object-Protocol (MOP) serves as the central hub that provides us with dynamic programming capabilities. Groovy comes with dynamic features out-of-the-box but you can also add your own. There are various ways of achieving this and we will start with the simpler ones and slowly move on to the more advanced use cases. As you will see, there is no reason to be scared about words like 'dynamic' or 'meta'. If by the end of this chapter you say: "Well, it isn't so magical after all", then we have achieved our goal.

If you seek perfection in completeness, designing and implementing an object-oriented system becomes hard. It may well be impossible.

Imagine you are responsible for `java.lang.Integer`. You are of course aware that this class will be used for counting, indexing, calculations, and so on but you cannot possibly anticipate all use cases.

Not before long, somebody will come along and would like to use it with a `times` method like in `3.times { println it }` , which you haven't foreseen, or calculate dates as in `2.weeks.from.today` but you haven't provided a `getWeeks()` method on Integer as would be needed to make the above possible. On another occasion, the user of your class may prefer having an exception being thrown on `Integer.MAX_VALUE + 1` rather than returning a negative number.

A third user would like to optimize an algorithm and to this end count the number of modulo operations on any integer that happens when his algorithm is executed. You are very unlikely to have anticipated such a requirement.

The good news is: dynamic programming allows adding such *features* later - without even touching the original! And the original can even be a Java class as long as it is called from Groovy.

Changes to such a ubiquitously used class as `Integer` are better only applied to the scope where you actually need them or you risk interference with seemingly unrelated parts of your codebase. Therefore, dynamic programming allows using such a feature only temporarily: adding and removing it at runtime, limiting its use to a given piece of code, to a class or only single instances, or even confine it to the current thread of execution.

Dynamic programming has a wide range of applicability, including

- designing domain specific languages (DSLs, see chapter 18),
- implementing builders (see chapter 10),
- advanced logging, tracing, debugging, and profiling,
- automated testing even where testing seems "impossible" (see chapter 16),
- putting lipstick on existing APIs, e.g. by eliminating the smell "incomplete library class"[1], to make them more complete, coherent, and accessible, and finally
- organizing the codebase such that *features* are kept in one place even if their behavior involves the collaboration of multiple classes. For example, you need abstractions for date, time, and duration working together to provide the date-calculation feature.

The last point is particularly interesting. You can observe it in Grails (xxx reference) where the *persistency feature* is dynamically available in all domain classes. On a domain class like `Person` you can call `Person.findAllByFirstName('Dierk')` to find all people in the database that share my first name, even though this method does not exist!

Note that such an approach has one quality that static code generation never achieves: since the code is not materialized anywhere, you cannot introduce errors in it! Also, your code is kept as clean as possible and you never have to read through code that was generated!

In this chapter, we will go through the various means of dynamic programming in Groovy and provide examples of the use cases above. Now, let's start with looking at what mechanics make our programming *dynamic*.

8.1 What is dynamic programming?

In classic object-oriented systems, every class has a well-known set of states, captured in the fields of that class, and well-known behavior, defined by its methods. Neither the set of states nor the behavior ever changes after compilation and it is identical for all instances of a class.

Dynamic programming breaks this limit by allowing the introduction of a new state but even more importantly, allowing the addition of a new behavior or modification of an existing one.

And what is "meta"?

"Meta" means applying a concept onto itself, e.g. meta-information is information about information. Likewise, since programming is "writing code", meta-programming means writing code that writes code. This includes source-code generation (e.g. producing a long String that is then evaluated as a Script), byte-code generation as explained in the next chapter, and pretending or synthesizing methods. The latter is part of dynamic programming and we will encounter it further down.

The use of "meta" as a qualifier in the Groovy runtime system is in many places debatable. Anyway, it is not only there for historical reasons, it also suggests that we are working on an elevated level of abstraction whenever this word is used.

How can we possibly add new state and behavior at runtime, when we are working on the JVM and the Java object model provides no such means? As the saying goes: "Every problem in computer science can be

[1] "Refactoring", Martin Fowler, Kent Beck

solved with a layer of indirection."[2] (beside the problem of too many layers of indirection). Enter the Meta-Object-Protocol.

8.2 The Meta-Object-Protocol

The approach is actually rather straightforward. Whenever "Groovy calls a method" it doesn't call it directly but asks an intermediate layer to do so on his behalf. The intermediate layer provides hooks that allow us to influence its inner workings.

A protocol is a collection of rules and formats. The Meta-Object-Protocol (MOP) is the collection of rules of how a request for a method call is handled by the Groovy runtime system and how to control the intermediate layer. The "format" of the protocol is defined by the respective APIs, which we will walk through in the course of this chapter.

An important part for understanding the mechanics is what it means when we say "Groovy calls a method". When writing Groovy source code, the Groovy compiler generates bytecode that calls into the MOP.

As an illustration, assume that our Groovy source code contains the statement

```
println 'Hello' // Groovy
```

Then the resulting bytecode that Groovy produces is roughly equivalent to the following Java code

```
InvokerHelper.invokeMethod(this, "println", {"Hello"}); // Java
```

When executed, the `InvokerHelper` as part of the MOP looks for the method named "`println`" with a `String` argument, finds that the Groovy runtime has registered such a method for `java.lang.Object`, and calls that implementation. This is a very shallow description of what actually happens but one that explains the principle and one that we can start with.

Takeaway

Every innocent method call that you write in Groovy is actually a call into the MOP, regardless of whether the call target has been compiled with Groovy or Java. This applies equally to static and instance method calls, constructor calls, and property access, even if the target is the same object as the caller.

Figure 8.1 shows how the MOP works like a filter for all method calls, which originate from code that was compiled by Groovy. The MOP is like a pair of rainbow-colored glasses that make all objects appear rich and powerful.

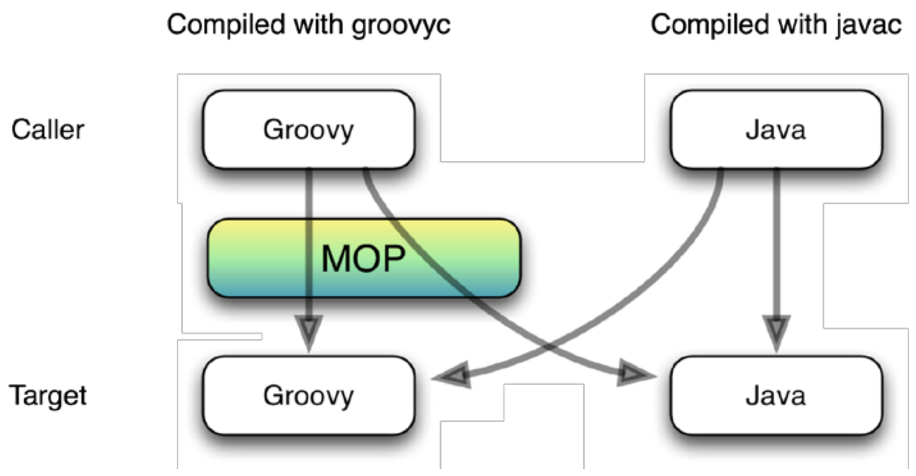


Figure 8.1 Every method call from a Groovy class or object into either Groovy or Java automatically goes through the Meta-Object-Protocol. Method calls from Java to both, Groovy and Java targets, do not use the MOP per default.

[2] Attribution is unclear, maybe David Wheeler or Andrew Koenig.

Figure 8.1 shows what happens by default. Of course, you can also call into the MOP from Java but this requires calling `InvokerHelper.invokeMethod()` explicitly. By default, Java classes only see what is in the bytecode of a class and not what the MOP adds to it - even if the target class was compiled by Groovy.

There is no spoon

Relating to the 'Matrix' motion picture[3], there is no such thing as a "Groovy class". You may have noticed that I avoid the wording of Java class versus Groovy class. That is because classes are classes are classes, regardless who compiled them. They have the exact same format and constraints. Of course, they differ in content but so do all classes anyway.

The MOP needs a lot of information in order to find the right call target for each method call that it serves. This information is stored in so-called meta classes. These meta classes are not fixed. One part of dynamic programming is changing the content of meta classes and replacing one meta class with another. We will explore this in section 8.4.

But even with the default meta class being in place - which does nothing fancy beside providing the GDK methods and doing some very advanced performance optimizations - the MOP knows about some special methods that allow the first degree of dynamic behavior. We call them "hooks methods".

8.3 Customizing the MOP with hook methods

Core of the MOP responsibilities is finding and selecting the right target method and handling the case when the requested method cannot be found. The first hook method that we will look at allows customizing the "missing method" case. A second hook method covers the case that a property access fails to find a property of the requested name. Then we explore the effects of combining hook methods with closure properties to allow instance-specific hooks that can even change at runtime. We finish up with custom logic for methods that objects need to provide if they implement the `GroovyObject` interface.

8.3.1 Customizing methodMissing

Whenever a method cannot be found in the target object, the MOP tries looks for the hook method

```
Object methodMissing(String name, Object arguments)
```

and invokes this method with the requested method name and arguments.

Listing 8.1 uses this hook in a `Pretender` class to merely return a String that shows what had been requested. The `Pretender` only *pretends*[4] to have the method `hello(String)` while in the bytecode of the class, there is no such method.

Listing 8.1 Bouncing when a missing method is called

```
class Pretender {
    def methodMissing(String name, Object args) {
        "called $name with $args"
    }
}
def bounce = new Pretender()
assert bounce.hello('world') == 'called hello with [world]'
```

The target is absolutely free in what it does inside `missingMethod`. It may provide a more sophisticated error handling than merely throwing a `MissingMethodException` (which is the default), delegate all calls to a collaborating object, or inspect method name and arguments to derive what needs to be done.

[3] The main character sees a child playing with a "dynamic spoon object" and realizes that it is not a truly physical object. Dynamic programming is kind of like that.

[4] Some call this a *synthesized* method but I feel this suggests that it somehow materializes, which it doesn't.

The last use case above goes into the direction that Grails provides with its dynamic finder methods in the Groovy object-relational mapping (GORM). Listing 8.2 outlines the approach. A method name like `findByXXX` is analyzed to select the search criterion.

Listing 8.2 Using missingMethod to simulate a miniature GORM

#1 Extract criterion from method name
#2 Setting up test data
#3 Call with pretended methods

Of course, a full implementation of GORM dynamic finder methods is more complex, but the principle and the mechanics are the same.

For the geeks

You can share the implementation of `missingMethod` by various means. One is to put it in a base superclass. Later, in section 8.4, we will see how methods can be injected into a class without even having access to the code of that class! In other words, you can even pretend having hook methods.

The `missingMethod` hook is quite simple to understand and use. Yet, it is very versatile and covers the vast majority of use cases for dynamic programming with DSLs. It is a very good entry point into dynamic programming and a good default choice when deciding upon which means of dynamic programming to apply. It comes with a counterpart that does to property access what `methodMissing` does to method calls.

8.3.2 Customizing propertyMissing

What `missingMethod` is for method calls is `propertyMissing` for property access. You implement

```
Object propertyMissing(String name)
```

to catch all access to non-existing properties. All the rest is exactly analogous to `methodMissing` such that listing 8.3 should be rather self-explanatory. We try to access the `hello` property, which is not in the bytecode.

Listing 8.3 Bouncing when a missing property is accessed

```
class PropPretender {
    def propertyMissing(String name) {
        "accessed $name"
    }
}
def bounce = new PropPretender()
assert bounce.hello == 'accessed hello'
```

This hook is a specialization of `methodMissing`: if you pretend the respective getter Method, you achieve the same effect. Anyway, having this more specialized hook is sometimes convenient. In listing 8.4 we use this hook as a method of the `Script` class to implement an easy way to calculate with binary numbers. This actually feels like a DSL.

The idea is quite simple. We would like to use symbols like `IOOI` to specify a positive integer of value 9 in its binary form. Now, simply using this symbol would throw us a `MissingPropertyException`. By providing a `propertyMissing` hook we can do the translation from a String into an Integer.

Listing 8.4 Using propertyMissing to calculate with binary numbers in DSL style

```
def propertyMissing(String name) {
    int result = 0
    name.each {
        result <<= 1
        if (it == 'I') result++
    }
    return result
}

assert IIOI +
        IOI ==
      I00I0
```

In case you have difficulties with the String-to-Integer translation logic above, don't worry, it is an implementation detail. The main point to take away is how to use the hook method.

Where there is specialization, there may also be generalization and actually, there is. But before we come to that in section 8.3.4, we enter a new dimension of dynamicity.

8.3.3 Using closures for dynamic hooks

By now, you may have the impression that MOP hook methods are very conventional. And in a way they are. They are just ordinary methods.

But if you think that this means that their behavior is guaranteed to be identical for all instances of your class, then this is not quite so in Groovy. In fact, if you wish so, you can even change the hook logic during the lifetime of an object!

Hook methods are not static. They are instance methods. Being that, they can work with the object's state. This state can include parameters for the hook logic. If these properties are of type `Closure`, then we have another example of "parameterization with logic" (cp. chapter 5 "Closures").

Listing 8.5 maintains a `whatToDo` property of type `Closure` that is called from inside a hook method. This allows changing the hook logic at runtime - and (not shown) having multiple instances of `DynamicPretender` using different closures.

Listing 8.5 Using the closure property pattern to change hook logic at runtime

#1 Closure property with default logic
#2 Delegating to the closure
#3 Change hook behavior at runtime

In classic Java programming, the behavior of a class never changes and the behavior is the same for all objects of the class. At best, you can use a Strategy Pattern[5] to switch between objects that behave differently. The above pattern of using a closure property to customize behavior of an object has a dynamic touch in itself, even though it is totally independent of the MOP. But in combination with the MOP, it adds a new dimension to the solution space.

The closure property pattern

All features of dynamic programming that are explained in this chapter can be combined with closure properties to open another dimension of versatility.

The hook methods that we have talked about so far apply regardless whether the call target is compiled by Groovy or Java. The next section will be about some more specific handling that the MOP applies to Groovy targets.

8.3.4 Customizing GroovyObject methods

All classes that are compiled by Groovy implement the `GroovyObject` interface, which looks like this:

```
public interface GroovyObject {
    public Object    invokeMethod(String methodName, Object args);
    public Object    getProperty(String propertyName);
    public void      setProperty(String propertyName, Object newValue);
    public MetaClass getMetaClass();
    public void      setMetaClass(MetaClass metaClass);
}
```

Again, you are free to implement any of such methods in your Groovy class to your liking. If you don't, then the Groovy compiler will insert a default implementation for you. This default implementation is the same as if you would inherit from `GroovyObjectSupport`, which basically relates all calls to the meta class. It roughly looks like this (excerpt):

[5] "Design Patterns: Elements of Reusable Object-Oriented Software", Gamma et al., 1994

```
public abstract class GroovyObjectSupport implements GroovyObject {

    public Object invokeMethod(String name, Object args) {
        return getMetaClass().invokeMethod(this, name, args);
    }
    public Object getProperty(String property) {
        return getMetaClass().getProperty(this, property);
    }
    public void setProperty(String property, Object newValue) {
        getMetaClass().setProperty(this, property, newValue);
    }
    // more here...
}
```

We defer the explanation of the meta class handling to the next section. For the moment, it is just a device that we can use for calling into the MOP.

Java can be Groovy

We can fool the MOP into thinking that a class that was actually compiled by Java was compiled by Groovy. We only need to implement the `GroovyObject` interface or - more conveniently - extend `GroovyObjectSupport`.

As soon as class implements `GroovyObject`, the following rules apply:

- Every access to a property calls the `getProperty()` method[6].
- Every modification of a property calls the `setProperty()` method.
- Every call to an unknown method calls `invokeMethod()`. If the method is known, `invokeMethod()` is only called if the class implements `GroovyObject` and the marker interface `GroovyInterceptable`.

Let's use this newly acquired knowledge to play with the Groovy language rules. In Groovy, parentheses for method calls are optional for top-level statements - but only if there is at least one argument. This is needed to distinguish method calls from property access. We cannot call `toString()` without the parentheses since `toString` would refer the property of the name `toString`. Listing 8.6 allows us to go around this limitation. We implement `getProperty()` such that if the property exists, we return its value, if not, we assume that the parameterless method shall be executed. Such a feature can be interesting when designing DSLs.

Listing 8.6 Using the getProperty hook to allow calling parameterless methods without parentheses

```
class NoParens {
    def getProperty(String propertyName) {
        if (metaClass.hasProperty(this, propertyName)) { //#1
            return metaClass.getProperty(this, propertyName)
        }
       invokeMethod propertyName, null  //#2
    }
}

class PropUser extends NoParens {       //#3
    boolean existingProperty = true
}

def user = new PropUser()
assert user.existingProperty
assert user.toString() == user.toString //#4
```

#1 Properties have priority
#2 Dynamic invocation

[6] There is a special handling for maps in the default meta class that makes sure that even though Map is not a GroovyObject, every property access on a map is relayed to the respective MapEntry.

#3 Subclass for feature sharing
#4 Look Ma! No parens!

This example uses the meta class and so leads us slowly into the topic of the next section where we will explore this concept in more details.

When we (#1) check whether a known property is requested, we ask our `metaClass` (i.e. we call the `getMetaClass()` method) if it has such a property. In case it has, we ask the `metaClass` for its value. Note that we cannot simply use `this."$propertyName"` since this would call `getProperty()` again, leading to endless recursion.

In order to (#2) eventually execute the method, we call the default implementation of the `invokeMethod()` hook, which relays the call to the meta class.

We see in (#3) that subclasses can share this "no-parens" feature. Subclassing is generally not a good way of sharing features but it works. We will discuss this further down and provide better alternatives.

Finally, we (#4) assert that omitting parentheses really works with selecting the ubiquitously available `toString()` method as our test candidate. Existing properties remain untouched.

Implementing `get/setProperty` can often improve the elegance of an API. Just consider Groovy maps. They relay property access like `map.a` to map content access like `map['a']` and you can do the equivalent with your own objects.

Note: getProperty() shadows propertyMissing()

Once you have implemented getProperty(), every property will be found and thus propertyMissing() will no longer be called.

So far, we have seen means of dynamic programming that require access to the source code of the target class and the possibility to apply modifications to it. We call this approach *intrusive*. You may be glad to hear that there also is a *non-intrusive* approach, which is the topic of our next section.

8.4 Modifying behavior through the meta class

By now you should feel at ease with the situation that all method calls that originate from Groovy code are routed through the MOP. If the last sentence still sounds odd to you, consider re-reading section 8.2 and doing some more experiments around the provided examples until you gained enough confidence to proceed.

With `methodMissing` and `propertyMissing` we have seen examples of hook methods that the MOP invokes when it cannot find the requested method or property. In this section, we will explore how Groovy tries to locate those and how we can use the lookup mechanism for our purposes of customizing the object's behavior.

8.4.1 MetaClass knows it all

For every class A in the class loader, Groovy maintains a meta class - an object of type `MetaClass`. This meta class maintains the collection of all methods and properties of A, starting with the bytecode information of A and adding additional methods that Groovy knows about per default (`DefaultGroovyMethods`).

Generally, all instances of class A share the same meta class. However, Groovy also supports having per-instance meta classes, i.e. different instances of A may refer to different meta classes. We will revisit this situation further down.

We can easily ask any meta class for its information and actually, we have seen this information already in the very beginning of this book in figure 1.6, which displayed the Groovy Object Browser.

Listing 8.7 inspects the capabilities of `MetaClass` by asking `String` for its meta class and calling various methods on it. We inspect the availability of methods with `respondsTo`, list all `properties`, list all `methods` from the bytecode, list all `metaMethods` that Groovy added dynamically, and we call `invokeMethod`, `invokeStaticMethod`, and `invokeConstructor` to show dynamic invocation.

Listing 8.7 Capabilities of MetaClass make the core of the Groovy reflection and dynamic method invocation

```
MetaClass mc = String.metaClass
final Object[] NO_ARGS = []
assert   1  == mc.respondsTo("toString", NO_ARGS).size()
assert   3  == mc.properties.size()
assert  75  == mc.methods.size()
assert 152  == mc.metaMethods.size()
assert ""   == mc.invokeMethod("","toString", NO_ARGS)
assert null == mc.invokeStaticMethod(String, "println", NO_ARGS)
assert ""   == mc.invokeConstructor(NO_ARGS)
```

There are more methods and more variants of the above in `MetaClass` but they give a good overview of what it does in general: providing means of *reflection* and *dynamic invocation*.

Calling a method means calling the meta class

As a rule of thumb you can assume that Groovy never calls methods directly in the bytecode but always through the object's meta class. At least, this is how it looks for you as a programmer.

Behind the scenes there are optimizations going on that technically circumvent the meta class but only when it is safe to do so.

Even the MOP hook methods that we have seen in earlier sections make no exception. If you provide your own implementation of let's say `invokeMethod`, then this method is added to your object's meta class at class loading time and later invoked from there.

All this should look to you as a pretty simple rule and you may ask what is so special about it. The trick is that a meta class can change at runtime and that an object may also change its meta class. Let's first investigate how Groovy finds meta classes.

8.4.2 How to find the meta class and invoke methods

We have seen that all `GroovyObjects` have a `metaClass` property (`setMetaClass` and `getMetaClass` methods). That makes it easy to find the meta class for them. We simply ask the object with `obj.metaClass.`

If we do not provide a custom implementation of the `metaClass` property accessor methods, the default implementation looks up the meta class in the so-called `MetaClassRegistry`. The registry maintains a map of classes and their meta classes. Figure 8.2 displays the connection between `GroovyObject`, `MetaClass`, and `MetaClassRegistry`.

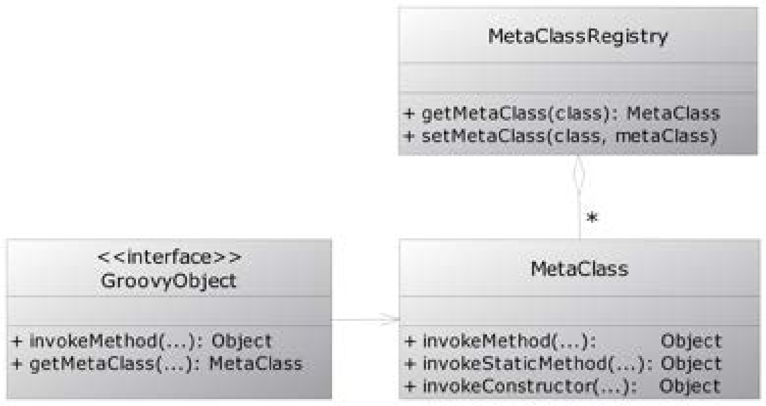

MetaClassRegistry
+ getMetaClass(class): MetaClass
+ setMetaClass(class, metaClass)
*
<<interface>>
GroovyObject
+ invokeMethod(...): Object
+ getMetaClass(...): MetaClass
MetaClass
+ invokeMethod(...): Object
+ invokeStaticMethod(...): Object
+ invokeConstructor(...): Object

Figure 8.2 UML class diagram of the GroovyObject interface that refers to an instance of class MetaClass, where MetaClass objects are also aggregated by the MetaClassRegistry to allow class-based retrieval of MetaClasses in addition to GroovyObject's potentially object-based retrieval

Objects that do not inherit from `GroovyObject` are not asked for the `metaClass` property. Their meta class is retrieved from the `MetaClassRegistry`.

For the geeks

The default meta class can actually be changed from the outside without touching any application code. Let's assume you have a class "Custom" in package "custom". Then you can change it's default meta class by putting a meta class with the name "groovy.runtime.metaclass.custom.CustomMetaClass" on the classpath.

This device has been proven useful when inspecting large Groovy codebases in production.

Putting all this together is a bit of a challenge. Below is a sketch in pseudo-code to keep the level of detail manageable while still revealing the core of the logic. Important methods from the meta class are shown in ***bold italics***, hook methods as underlined. Note that `invokeMethod` appears twice: with two parameters as a hook method and with three parameters in `MetaClass`.

At the very beginning, we decide whether we have a `GroovyObject` and if not, look for the meta class in the registry and use it to invoke the requested method.

```
// MOP pseudo code
def mopInvoke(Object obj, String method, Object args) {
    if (obj instanceof GroovyObject) {
        return groovyObjectInvoke(obj, method, args)
    }
    registry.getMetaClass(obj.class).invokeMethod(obj, method, args)
}
```

If we have a `GroovyObject`, we use the `metaClass` property to find the meta class but we also have to care for the special handling around `GroovyInterceptable` and unknown methods (see section 8.2).

```
def groovyObjectInvoke(Object obj, String method, Object args){
    if (obj instanceof GroovyInterceptable) {
        return obj.metaClass.invokeMethod(method, args)
    }
    if (! obj.metaClass.respondsTo(method, args)) {
        return obj.metaClass.invokeMethod(method, args)
    }
    obj.metaClass.invokeMethod(obj, method, args)
}
```

You may ask why `methodMissing` does not appear in the code above. This case is handled in the default meta class:

```
// Default meta class pseudo code
def invokeMethod(Object obj, String method, Object args) {
    if (obj.metaClass.respondsTo(method, args)) {
        return methodCall(obj, method, args)
    }
    if (methodMissingAvailable(obj)) {
        return obj.metaClass.methodMissing(method, args)
    }
    throw new MissingMethodException()
}
```

Don't forget that all the above is pseudo-code. That actual implementation differs quite a bit - mostly for performance reasons. Also, the code is supposed to have Java semantics, i.e. all method calls and property access are direct and do not go through the MOP itself. Otherwise, we would run into endless recursion.

The mechanics of the MOP may appear complex but for usual case you can simply assume that all method calls go through the meta class and the default meta class is in place. This raises the question what other meta classes are available and why we would want to use them.

8.4.3 Setting other meta classes

Groovy comes with a number of meta classes:

- the default meta class `MetaClassImpl`, which is used in the vast majority of cases,
- the `ExpandoMetaClass`, which can expand state and behavior,
- a `ProxyMetaClass`, which can decorate a meta class with interception capabilities, and
- more meta classes that are used internally and for testing purposes.

Let's look at `ProxyMetaClass` as an example of how to use a customized meta class. A `ProxyMetaClass` wraps an existing meta class that it relays all method calls to. When doing so, it provides the ability to execute customized logic before and after each method call. That customized logic is captured in a so-called `Interceptor`. With Groovy comes a `TracingInterceptor` that simply logs all method access to a Writer, effectively providing a trace of all method calls. Listing 8.8 configures such a `ProxyMetaClass` with a `TracingInterceptor` and assigns this meta class to an arbitrary Groovy object that should be subject to tracing.

Listing 8.8 Assigning a ProxyMetaClass to a GroovyObject for tracing method calls

#1 Setup
#2 Assigning a meta class
#3 Normal method call

Our self-testing code requires a small change to the default. In (#1) we set up the `TracingInterceptor` to not print to `System.out` but to use a `StringWriter` that we can later inspect for its content.

In (#2) we assign our meta class to the object under inspection. We do have a per-instance meta class this way.

When we call any method on our object under inspection as in (#3), then all method calls and returns are traced - even in private methods. Note that this does not require any change in `InspectMe` nor is that class in any way aware of the tracing. It is all controlled from the outside. This is what we call *non-intrusive*.

Interceptors are more than aspects

Interceptors may remind one or the other reader on aspect oriented programming (AOP) and the `TracingInterceptor` suggests this connotation. However, interceptors can do much more: they can redirect to a different method, change the arguments, suppress the method call, and even change the return value!

Oftentimes, you may want to use the proxy meta class only temporarily and set the meta class back to the original afterwards. In such a case can put the proxy-using code inside a closure and give it to the `use` method like

```
proxyMetaClass.use(inspectMe){
    inspectMe.outer() // proxy in use
}
// proxy is no longer in use
```

Manually setting the meta class of a Groovy object works as expected and working through the example has confirmed our understanding of the MOP. But Groovy wouldn't be Groovy if it would leave us behind with only the low-level devices.

In the next sections we will see ways of working with the MOP on a higher level of abstraction to make it more accessible, more flexible, and more convenient to work with for specialized use cases.

8.4.4 Expanding the meta class

Since its early days, Groovy has a class called `Expando`. It is a tiny class with few methods but one interesting characteristic: it can expand its state and behavior. Listing 8.9 uses an `Expando` as a boxer who can take some hits but will eventually fight back.

Listing 8.9 An Expando can extend state and behavior at runtime

```
def boxer = new Expando()

boxer.takeThis  = 'ouch!'
boxer.fightBack = { times -> takeThis * times  }

assert boxer.fightBack(3) == 'ouch!ouch!ouch!'
```

New state is assigned to not-yet-existing properties, analogous to what we have seen for maps.

New behavior is assigned to not-yet-existing properties as closures. After the assignment, it can be called as if it was a method.

The reason for explaining the `Expando` class here is that there is an `ExpandoMetaClass` in Groovy that - as you may have guessed - is a meta class that works like an `Expando`. You can register new state (properties) and new behavior (methods) in the meta class by simply using property assignments.

Listing 8.10 introduces the concept with an example that adds the a new method called `low()` to `java.lang.String`. It does the same as `toLowerCase()` but is shorter and the spelling is easier to remember. We do not need to set the `ExpandoMetaClass` explicitly. Groovy automatically replaces the default meta class with an `ExpandoMetaClass` when we apply any modification to it.

Listing 8.10 Adding the low() method to java.lang.String via ExpandoMetaClass

```
assert String.metaClass =~ /MetaClassImpl/
String.metaClass.low    = {-> delegate.toLowerCase() }
assert String.metaClass =~ /ExpandoMetaClass/

assert "DiErK".low() == "dierk"
```

Note that our closure uses the `delegate` reference to refer to the actual `String` instance that the closure is called upon. The closure must also have the right number of parameters. The usual rules for closure parameters apply, i.e. type markers are optional, you can use default values, varargs, etc. Since our method shall not have any parameters we use an empty parameter list `{-> ...}`.

Listing 8.11 adds a new property `myProp` and a new method `test` to the meta class of `MyGroovy1` - a class that is written in Groovy. Note that the dynamic `test` method refers to the dynamic property `myProp`. These dynamic features are only available for objects of type `MyGroovy1` that have been constructed *after* the meta class modification.

Listing 8.11 Modifying the meta class of a class (Groovy & Java)

```
class MyGroovy1 { }

def before = new MyGroovy1()

MyGroovy1.metaClass.myProp = "MyGroovy prop"
MyGroovy1.metaClass.test = {-> myProp }

try {
    before.test()                   //#1
    assert false, "should throw MME"
} catch(mme) { }

assert new MyGroovy1().test() == "MyGroovy prop"
```

#1 not available

Above, we have changed the meta class of a class and thus for all instances of that class. In listing 8.12 we do the very same but only on a single instance. Only the `myGroovy` instance gets the new dynamic features since we only modify a per-instance meta class.

Listing 8.12 Modifying the meta class of a Groovy instance

```
class MyGroovy2 { }

def myGroovy = new MyGroovy2()

myGroovy.metaClass.myProp = "MyGroovy prop"
myGroovy.metaClass.test = {-> myProp }

try {
    new MyGroovy2().test()          //#1
    assert false, "should throw MME"
} catch(mme) { }

assert myGroovy.test() == "MyGroovy prop"
```

#1 not available

Per-instance meta classes are very valuable since they allow fine-grained control over where and how dynamic features are added.

Imagine a large development team where accidentally two developers modify the same meta class with the same method names for different reasons. The last modification wins and may compromise the logic of the developer who did the first change.[7]

With per-instance meta classes such clashes are easier to avoid. Listing 8.13 uses per-instance meta classes even for such a ubiquitous Java object as a `String` while avoiding clashes.

Listing 8.13 Modifying the meta class of a Java instance

```
def myJava = new String()

myJava.metaClass.myProp = "MyJava prop"
myJava.metaClass.test = {-> myProp }

try {
    new String().test()             //#1
    assert false, "should throw MME"
} catch(mme) { }

assert myJava.test() == "MyJava prop"
```

#1 not available

So far, we have asked classes and objects for their meta class every single time when we did a modification. Listing 8.14 introduces a new so-called *builder* style for doing multiple changes at once. We use it to `encode` and `decode` Strings by moving every character up and down the alphabet with the respective methods, a meta class property to capture how many characters to `shift` up or down, and we also provide property accessor methods to work more conveniently with the `code` and the `original`.

If you have ever seen Stanley Kubrick's motion picture "A Space Odyssey", you may remember the super-intelligent computer "HAL". I turns out that this is an encoded version of "IBM". Well, things could have been worse for that company if the writer Arthur C. Clarke would have chosen a different shift distance for the encoding…

Listing 8.14 Decoding a space odyssey with a meta class builder

```
def move(string, distance) {
    string.collect { (it as char) + distance as char }.join()
}

String.metaClass {
```

[7] This situation is often called "monkey patching", referring to programmers that use programming constructs that they have seen elsewhere without fully understanding what they do: "monkey see - monkey do".

```
        shift = -1
        encode  {-> move delegate, shift  }
        decode  {-> move delegate, -shift }
        getCode {-> encode() }
        getOrig {-> decode() }
    }

    assert "IBM".encode() == "HAL"
    assert "HAL".orig     == "IBM"

    def ibm = "IBM"
    ibm.shift = 7
    assert ibm.code == "PIT"
```

Note that we can change the `shift` distance on a per-instance basis by setting the respective property.

Note

Modifying the meta class of the String class will affect all future String instances.

In all the examples above, we have added new instance methods to all instances of a class or to only a specific instance of a class. Listing 8.15 adds a static method to `java.lang.Integer` by using the `static` keyword. We can now ask the `Integer` class (as opposed to an `Integer` object) for the answer to "life, the universe, and everything".

Listing 8.15 Adding a static method to a class

```
Integer.metaClass.static.answer = {-> 42}

assert Integer.answer() == 42
```

When talking about objects, we also have to consider inheritance. Listing 8.16 adds a new method `toTable()` dynamically to a superclass and asserts that it is transparently available in its subclass.

We can even modify the meta class of interfaces and all classes that implement this interface share the new behavior. This feature isn't enabled by default, though. We have to enable it via `ExpandoMetaClass.enableGlobally()`.

Listing 8.16 Meta class changes for superclasses and interfaces

```
class MySuperGroovy { }
class MySubGroovy extends MySuperGroovy { }

MySuperGroovy.metaClass.added = {-> true }

assert new MySubGroovy().added()

//ExpandoMetaClass.enableGlobally() // xxx delete?

Map.metaClass.toTable = {->
    delegate.collect{ [it.key, it.value] }
}

assert [a:1, b:2].toTable() == [
       ['a', 1],
       ['b', 2]
]
```

Note that we call `toTable()` on a literally declared map, which is of type `LinkedHashMap`. Even though we have added our new method to the meta class of the `java.util.Map` interface, it is available for all instances of its subtypes.

We mop[8] the meta class topic up with an example that should illustrate that we can add any kind of method dynamically - even operator methods and MOP hook methods.

Listing 8.17 adds the >>> operator to `Strings` with the `rightShiftUnsigned` operator method to split the string by words and push them to the right. It then replaces names with nicknames by calling a method of the to-be-replaced name with the replacement as the argument. To make this possible for every conceivable name, it adds the `methodMissing` hook to `String`.

Listing 8.17 Meta class injection of operator and MOP hook methods

```
String.metaClass {
    rightShiftUnsigned = { prefix ->
        delegate.replaceAll(~/\w+/) { prefix + it }
    }
    methodMissing = { String name, args->
        delegate.replaceAll name, args[0]
    }
}

def people = "Dierk,Guillaume,Paul,Hamlet,Jon"
people  >>>= "\n    "
people     =  people.Dierk('Mittie').Guillaume('Mr.G')

assert people == '''
    Mittie,
    Mr.G,
    Paul,
    Hamlet,
    Jon'''
```

Finally, some takeaways and rules of thumb for meta classes:

- All method calls from Groovy code go through a meta class.
- Meta classes can change - for all instances of a class or per single instance.
- Meta class changes affect all future instances in all running threads.
- Meta classes allow non-intrusive changes to both Groovy and Java code as long as the caller is Groovy. We can even change access to final classes like `java.lang.String`.
- Meta class changes can well take the form of property accessors (pretending property access), operator methods, `GroovyObject` methods, or MOP hook methods.
- `ExpandoMetaClass` makes meta class modifications more convenient.
- Meta class changes are best applied only once - preferably at application startup time.

The last point directly leads us to another concept of dynamic programming in Groovy. `ExpandoMetaClass` is not designed for easily removing a once dynamically added method or un-doing any other change. For such temporary changes, Groovy provides category classes.

8.4.5 Temporary MOP modifications using category classes

Meta classes are the main workhorses for dynamic programming in Groovy but sometimes we do not need their full power and would prefer an alternative that is small and focused and confined to the current thread and a small piece of code. This is exactly what category classes are. We will look into how to use existing category classes, what benefits they bring, and how to write our own ones.

Using a category class is trivial. Groovy adds a `use` method to `java.lang.Object` that takes two parameters: a category class (or any number thereof) and a closure.

```
use CategoryClass, {
    // new methods are available
```

[8] pun intended

```
}
// new methods are no longer available
```

While the closure is executed, the MOP is modified as defined by the category. After the closure execution is finished, the MOP is reset to its old state.

Listing 8.18 leads us through two examples of using a category: a `TimeCategory` that is part of Groovy and the `java.util.Collections` class.

`TimeCategory` allows simplified working with date, time, and duration for both, easier definition and easier calculation. If you have an appointment in two weeks, you can find the date with `2.weeks.from.today`.

`Collections` is the unmodified class from the JDK. It contains a number of static helper methods.

Listing 8.18 How to use existing categories like TimeCategory and Collections

```
import groovy.time.TimeCategory

def janFirst1970 = new Date(0)
use TimeCategory, {
    Date   xMas = janFirst1970 + 1.year - 8.days
    assert xMas.month == Calendar.DECEMBER
    assert xMas.date  == 24
}

use Collections, {
    def list     = [0, 1, 2, 3]
    list.rotate 1
    assert list == [3, 0, 1, 2]
}
```

Inside the closures, we have new properties on numbers (`1.year`) new operator methods for calculating dates, and a new `rotate` method on lists. Outside the closures, no such feature is visible. Note that `janFirst1970` was constructed before the `use` closure.

Category classes are by no means special. Neither do they implement a certain interface nor do they inherit from a certain class. They are not configured or registered anywhere! They just happen to contain static methods with at least one parameter.

When a class is used as an argument to the `use` method, it becomes a category class and every static method like

static *ReturnType* *methodName*(*Receiver* self, *optionalArgs*) {...}

becomes available on the receiver as if the `Receiver` had an instance method like

ReturnType *methodName*(*optionalArgs*) {...}

An example says it better than any explanation. Listing 8.19 defines a class `Marshal` with static methods to `marshal` and `unMarshal` an integer to and from a string. The string version may be used for sending the integer to a remote machine. When we `use` the `Marshal` category class, we can call `marshal()` on an integer and `unMarshal()` on a string.

Listing 8.19 Running your own category to marshal/unmarshal integers to/from strings

```
class Marshal {
    static String marshal(Integer self) {
        self.toString()
    }
    static Integer unMarshal(String self) {
        self.toInteger()
    }
}

use Marshal, {
        assert   1.marshal()   == "1"
        assert "1".unMarshal() == 1
        [Integer.MIN_VALUE, -1, 0, Integer.MAX_VALUE].each {
            assert it.marshal().unMarshal() == it
        }
    }
```

Naming the receiver object `self` is just a convention. You can use any name you want. Groovy's design decision of using static methods to implement category behavior has some beneficial effects. First, we are much less likely to run into concurrency issues, since there is less shared state. Second, we can use a plethora of classes as categories even if they have been implemented without knowing about Groovy. `Collections` was just an example of many classes with static methods that reside in widely used helper libraries. Third, they can easily be created in Groovy, Java, or any other JVM language that produces classes and static methods.

Category classes are a good place to collect methods that work conjointly on different types, e.g. Integer and String, to accomplish a feature, e.g. marshalling.

Key characteristics of using category classes:

- The `use` method applies categories to the runtime scope of the closure (as opposed to the literal scope). That means you can extract code from the closure into a method and call the method from inside the closure.
- Category usage is confined to the current thread.
- Category use is non-intrusive.
- If the receiver type refers to a superclass or even an interface, then the method will be available in all subclasses/implementors without further configuration.
- Category method names can well take the form of property accessors (pretending property access), operator methods, and `GroovyObject` methods. MOP hook methods cannot be added through a category class[9].
- Category methods can override method definitions in the meta class.
- In places where performance is crucial, use categories with care and measure their influence.
- Categories cannot introduce new state in the receiver object, i.e. they cannot add new properties with a backing field.

The last point reveals that even though categories are a great tool for combining behavior into reusable features they do have their limitations when it comes to sharing state. For this, we have Mixins, which are the final topic in our dynamic programming tour.

8.4.6 Merging classes with Mixins

Have you ever noticed that in Java many interface names end with "-able"? `Appendable`, `Adjustable`, `Activatable`, `Callable`, `Cloneable`, `Closeable`, etc. make a really long list. That is because they refer to an *ability*.

An object may have many abilities and so its class may implement many interfaces but reusing implementations of any such ability is restricted to only one superclass. In Java and Groovy alike, you can only inherit once even though you can implement many interfaces.

Reuse by inheritance in questionable

Using inheritance for reuse of an ability implementation is often frowned upon. Instead of implementation reuse, it is considered good object-oriented design to only use inheritance if there is a true "is-a" relationship between subclass and superclass.

[9] This is a restriction as of Groovy 1.8. The feature may become available in later versions.

If you have a superclass A with a subclass B than any object of class B is not only a B, it also *is* an A! The definitions of A and B typically reside in different files[10]. The situation looks as if A and B would be merged when constructing an instance of B. They share both state and behavior.

This "class merging" by inheritance is pretty restricted in Java.

- You cannot use it when inheritance has already been used for other purposes.
- You cannot merge (i.e. inherit from) more than one class.
- It is intrusive, i.e. you have to change the class definition.
- You cannot do it with final classes.

Groovy provides a feature called Mixin that addresses exactly these limitations. Listing 8.20 uses the `@Mixin` class annotation to mix reusable state and behavior into a test case that uses inheritance in order to be recognized by the testing framework.

Listing 8.20 Mixing a feature into a test case by using the @Mixin annotation

```
@Mixin(MessageFeature)
class FirstTest extends GroovyTestCase {
    void testWithMixinUsage() {
        message = "Called from Test"
        assertMessage "Called from Test"
    }
}
class MessageFeature {
    def message
    void assertMessage(String msg) {
        assertEquals msg, message
    }
}
```

Note that you can execute listing 8.20 as a script and it will run the test case with the bundled JUnit. Test frameworks for both unit and functional tests tend to use inheritance a lot even though this is no longer considered good framework design. Inheritance makes it more difficult to nicely factor out common state and behavior. With Mixins, we can circumvent this restriction. They make a good companion for unit tests with JUnit and functional tests with Canoo WebTest.

Using the `@Mixin` annotation is intrusive. We have to change the code of the class that receives the new features. Listing 8.21 in contrast works non-intrusively. It calls the `mixin` method on the `List` interface to mix in two different features that "sieve" factors of 2 or any other number from a list of numbers. Such a feature is helpful when implementing the Sieve of Erastothenes[11] to efficiently find prime numbers.

Listing 8.21 Mixing-in multiple sieve features non-intrusively

```
class EvenSieve {
    def getNo2() {
        removeAll { it % 2 == 0}
        return this
    }
}
class MinusSieve {
    def minus(int num) {
        removeAll { it % num == 0}
        return this
    }
}
```

10 Because of Java's late binding, you cannot even be sure that the A that was available at compile time is the same A that is used at runtime. Java is much more dynamic than many might assume.

11 http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

```
List.mixin EvenSieve, MinusSieve

assert (0..10).toList().no2 - 3 - 5 == [1, 7]
```

You see that we can mix-in multiple classes (`EvenSieve`, `MinusSieve`) with property accessor methods (`getNo2`) and operator methods (`minus`) not only to concrete classes but also to interfaces (`List`).

The surprising part is how easily the sieve classes implement their feature methods as if they were of type `List` themselves, which they aren't. Even the return value "`this`" refers to the actual `ArrayList` instance when the method is called from the `ArrayList` - but not when you are looking at "`this`" from inside the feature method.

Mixins are often compared with multiple inheritance but they are of a different nature. In the first place, our `ArrayList` doesn't become a subtype of `MinusSieve`. Any `instanceof` test will fail. There is no "is-a" relationship and no polymorphism. You can use enforced type coercion with the "`as`" operator, though.

Unlike many models of multiple inheritance, the mixing-in of new features always happens in traceable sequence and in case of conflicts, the latest addition wins. Mixins work like meta class changes in that respect.

To sum it up, here are the important characteristics of Mixins:

- You can instantiate objects from a blend of many classes. The object's state and behavior encompasses all properties and methods of all mixed classes.
- There is an intrusive use with the `@Mixin` class annotation and a non-intrusive use with the `mixin` method on classes. Both alternatives happen at runtime (as opposed to compile time). `@Mixin` happens at class construction time in a static initializer.
- Mixins are visible in all threads.
- There are no restrictions what methods to mix in. Property accessors, operator methods, `GroovyObject` methods, and even MOP hook methods all work fine.
- You can mix into superclasses and interfaces.
- A Mixin can override a method of a previous Mixin but not methods in the meta class.
- There is no per-instance Mixin. You can only mix into classes and meta classes. To achieve the effect of a per-instance Mixin, you can mix into a per-instance meta class.
- Mixins cannot easily be un-done.

In general, Mixins are designed for sharing features while not modifying any existing behavior of the receiver. Features can build on top of each other and merge and blend with the receiver.

MOP priorities

It is always a good advice to keep things simple. With dynamic programming one can easily go overboard by doing too much, e.g. using category classed, meta class changes, and Mixins in combination. If you do anyway, then categories are looked at first, then the meta class, and finally the Mixins:

category class > meta class > mixin

But this only applies to methods that are defined for the same class and have the same parameter types. Otherwise, the rules for method dispatch by class/superclass/interface take precedence.

Latest wins

In case of multiple method definitions, a category class shadows a previously applied category class.

Changes to an ExpandoMetaClass override previously added methods in that meta class.

Later applied Mixins shadow previously applied Mixins.

That it was for the technical description of Groovy's dynamic programming devices. It was quite a number of different concepts to understand and remember. Their real value will become apparent when you use them in practice and the following use cases may give you some inspiration when and how to try some dynamic programming yourself.

8.5 Real-world dynamic programming in action

After having seen the various means of dynamic programming in Groovy you may ask yourself how this applies to real-world projects. If you haven't seen much dynamic programming in your career so far, you may even ask whether it is valuable at all since apparently, you have been able to live without it so far.

This section presents five scenarios that we have derived from working experience with Groovy. They are taken from real codebases with minor modifications. We will always start with explaining the task such that you can take it as an exercise to come up with your own solution. Then we will present a solution and talk about the design rationale. We start simple and proceed to the more complex.

8.5.1 Calculating with metrics

I always do silly mistakes when calculating with measurements that have a different orders of magnitude. How many nanoseconds are there in a second? Hm - I must concede that I would rather look it up than guessing.

But Groovy can help us. Let's take meters, centimeters and millimeters as a simple example. If we could simply write "`1m + 20.cm - 8.mm`", that would be much easier than calculating with 1192 millimeters.

The task is to make the above possible. Calculations shall be done in millimeters. The feature shall be ubiquitously available.

Listing 8.22 addresses the requirements by adding the respective property accessor methods.

Listing 8.22 Metric calculations that avoid common magnitude mistakes

```
Number.metaClass {
    getMm = { delegate          }
    getCm = { delegate *  10.mm }
    getM  = { delegate * 100.cm }
}

assert 1.m + 20.cm - 8.mm == 1.192.m
```

We chose a meta class modification as the vehicle to introduce the new getters for the remainder of the program. We add them to the `Number` interface to not only accommodate `Integers` but also `Doubles`, `Floats`, etc.

Note that from inside one new feature method we can call the others. Specifying that one meter is 100 cm is more obvious than trying to specify a meter in terms of millimeters.

A solution like the above can be found in many domain specific languages. You will find more examples in chapter 18.

8.5.2 Replacing constructors with factory methods

New objects are usually constructed by using the "`new`" keyword and a constructor as in "`new Integer(42)`". Many question this language design and you often hear the advice to favor factory methods over direct constructor calls.

The task is to change the Groovy language so that every class can be constructed by a static factory method called "`make`" with the same parameters as the respective constructor, e.g. "`Integer.make(42)`" shall replace "`new Integer(42)`".

Listing 8.23 goes for a solution that is essentially a one-liner, even though it is typeset on three lines for better reading. It tests itself with factory methods that take zero, one, and two parameters.

Listing 8.23 Introducing static factory methods to all classes

```
import java.awt.Dimension

Class.metaClass.make = { Object[] args ->
    delegate.metaClass.invokeConstructor(*args)
}

assert new HashMap()        == HashMap.make()
assert new Integer(42)      == Integer.make(42)
assert new Dimension(2, 3)  == Dimension.make(2, 3)
```

Quite obviously, we have to introduce the 'make' method on some meta class. But which one? Well, it shall be available on every class, i.e. on every instance of java.lang.Class. Therefore, we add it to the meta class of the Class class.[12]

Invoking the constructor is done dynamically, i.e. on the meta class of the current Class object, which we refer to as the delegate. To allow any number of parameters we use varargs (Object[]) in the closure parameter list and spread all arguments over the invokeConstructor argument list with the spread operator (*args).

This has been a tiny change and we have seemingly changed all classes in the system! That is the true power of dynamic programming. Try this with a static language!

Our example has a number of real-world usages. The Ruby language for example solely relies on this approach to constructing objects. Tammo Freese first explored the solution when he, Johannes Link, and I designed our "Groovy in a day" workshop.

8.5.3 Fooling IDEs for fun and profit

Imagine you had a set of components that you have to connect to each other. One component's output channel shall be connected to another component's input channel. Let's call the process of defining the connections "wiring".

We could do the wiring by maintaining a list of pairs where every pair reflects one connection between a source and a target component. However, we do not get much IDE support when we create such pairs.

The task is to allow an approach to wiring that gives us IDE support and checks for assignable types such that only channels of assignable types are wired. All components should remain untouched in the wiring process.

Listing 8.24 comes up with a solution that fools your IDE into thinking that there would be property assignments while we actually intercept the assignment and only register the call for the wiring. Depending on the quality of your IDE support, it will check the assignment statements for assignable types and will suggest only those.

Listing 8.24 Temporarily faking property assignments for configuration purposes

```
interface ChannelComponent {}
class Producer implements ChannelComponent {
    List<Integer> outChannel
}
class Adaptor implements ChannelComponent {
    List<Integer> inChannel
    List<String>  outChannel
}
class Printer implements ChannelComponent {
    List<String> inChannel
}

class WiringCategory {
    static connections = []
    static setInChannel(ChannelComponent self, value){ //#1
        connections << [target:self, source:value]
    }
    static getOutChannel(ChannelComponent self){
        self
    }
}
```

[12] If you have gone cross-eyed by now, don't worry. That is a healthy reaction. Re-reading and understanding the last paragraph will improve your nerd level at the possible risk of compromising your common sense.

```
Producer producer = new Producer()
Adaptor  adaptor  = new Adaptor()
Printer  printer  = new Printer()

use WiringCategory, {
    adaptor.inChannel = producer.outChannel //|#2
    printer.inChannel = adaptor.outChannel  //|#2
}

assert WiringCategory.connections == [
        [source: producer, target: adaptor],
        [source: adaptor,  target: printer]
]
```

#1 Intercept assignments
#2 Fake assignments

Since the components shall remain untouched, we use a category class for the scope of the wiring. The assignments are intercepted by overriding the respective property getter and setter methods non-intrusively on the common interface of all components.

The solution is a simplified version of the wiring in the PillarOne project (www.pillarone.org). PillarOne is an open-source project for risk calculation in the insurance industry. It makes heavy use of Groovy for specifying risk models made from wired components.

8.5.4 Undoing meta class modifications

Modifying a meta class is simple. Undoing such a modification can be a bit involved, though. The task is to try various approaches and to start with an experiment that modifies the `size()` method of `String` such that it returns twice the actual value by referring to the old implementation. Later we want to set the `size()` method back to the original behavior.

Listing 8.25 searches the meta class of `String` for the meta method of `size()` and stores it for later reference. A `MetaMethod` has an `invoke` method that takes the receiver object as the first parameter.

Listing 8.25 Method aliasing and undoing meta class modifications

```
MetaClass oldMetaClass = String.metaClass           //#1

MetaMethod alias = String.metaClass.metaMethods     //#2
                   .find { it.name == 'size' }
String.metaClass {
    oldSize = { -> alias.invoke delegate  }
    size    = { -> oldSize() * 2 }
}

assert "abc".size()    == 6
assert "abc".oldSize() == 3

if (oldMetaClass.is(String.metaClass)){
    String.metaClass {                              //#3
        size    = { -> alias.invoke delegate }
        oldSize = { -> throw new UnsupportedOperationException() }
    }
} else {
    String.metaClass = oldMetaClass                 //#4
}

assert "abc".size() == 3
```

#1 Store old meta class
#2 Store meta method
#3 Reverse modification
#4 Reset meta class

When overriding a method on the meta class, there is nothing like "super" that we would have used in subclasses to refer to an original implementation in a superclass. As a replacement, we introduce a new method `oldSize()` as an alias for the (#2) old method such that we can refer to it.

Undoing that modification comes in two flavors: doing a (#3) reverse modification or (#4) setting the meta class instance back to the (#1) original instance in case the instance has changed. If before the modification the default meta class was in use, then it was changed into an `ExpandoMetaClass` with the first modification and we can reset to the old meta class. Otherwise, we have already started with an `ExpandoMetaClass` and only modified that instance.

Resetting the meta class instance is the cleaner way but it is only available if they were no changes to the meta class of String before we started. The code above is again a simplified version of meta class handling in the PillarOne project.

8.5.5 The intercept/cache/invoke pattern

The `methodMissing` hook method is a cornerstone of the MOP. Some people even define dynamic programming by the availability of such a method. However, it comes at a cost. Since Groovy first tries all other possibilities of finding a suitable method before it finally calls `methodMissing`, this requires some time. It is also very common that a method that has been called once will be called again.

The task is to step into `methodMissing` at most once for every distinct method call. As an example we want to support methods of the form `findBy`*`<propertyName>`*`(`*`value`*`)` that searches any collective datatype for items that have a property of that name with the given value. We seek an optimized and non-intrusive version of listing 8.2.

Listing 8.26 searches a list of maps for planets with a given name or average distance from earth in astronomical units (rounded).

Listing 8.26 The intercept/cache/invoke pattern for finding-by-property-value

```
Object.metaClass.methodMissing = { String name, Object args ->
    assert name.startsWith("findBy")
    assert args.size() == 1
    Object.metaClass."$name" = { value ->       //#1
        delegate.find { it[name.toLowerCase()-'findby'] == value }
    }
    delegate."$name"(args[0])                  //#2
}

def data = [
    [name:'moon',    au: 0.0025],
    [name:'sun',     au: 1     ],
    [name:'neptune', au:30     ],
]

assert data.findByName('moon')     //#3
assert data.findByName('sun')     //#4
assert data.findByAu(1)
```

#1 Cache the method
#2 Invoke the method
#3 Intercepted call
#4 Cached call

We add the `methodMissing` hook to the meta class of `Object`. We choose type `Object` since we have no known restrictions. Whenever we enter the hook method we (#1) add a new method of the requested name to our meta class. For this new method, the missing method hook method will never be called again, since it is no longer "missing". We have *synthesized* a new method.

We also need to execute the synthesized method, which we do in #2.

The intercept/cache/invoke pattern was invented by Graeme Rocher, the project lead of the Grails web platform. It is a core part of the Grails infrastructure. The productive version is a bit more elaborate than our example, mainly to work nicely in highly concurrent environments, but the general approach is the same.

8.6 Summary

We hope that by the end of this chapter you have gained a good overview of the various concepts that allow dynamic programming with Groovy. These language capabilities may have been new to you and thus unfamiliar and maybe even daunting.

But even if they appear like magic, they are all easily explained by the fact that Groovy sees the world through the glasses of the meta object protocol. The MOP itself offers many alternatives for adapting it to new necessities.

We can use the MOP hook methods intrusively or apply non-intrusive changes by switching meta classes, modifying meta classes, using categories, or mixing-in new state and behavior. All these devices come in combination with the Groovy method dispatch, property handling, operator methods, `GroovyObject` methods, and inheritance. The pervasive use of closures adds another dimension of dynamically changing behavior at runtime.

Once you have experienced the merits of dynamic programming, you will find it unwieldy to go back to a static language.

You may be surprised to hear that the topic of dynamic programming isn't over, yet. What we have covered so far is the runtime aspect of it. But there are also compile-time aspects that we will explore in the next chapter.

9 Compile-Time Metaprogramming and AST Transformations

By Hamlet D'Arcy

“It is my firm belief that all successful languages are grown and not merely designed from first principles.”

– Bjarne Stroustrup, The Design and Evolution of C++

This chapter covers

- Removing redundancy and verbosity with Groovy's metaprogramming annotations
- Writing your own compiler extensions using the AST transformations feature
- Compile-time metaprogramming testing, tools, and pitfalls

In the last chapter we looked at dynamic programming with Java, where the behavior of a type or even an individual object can change while the program is executing. We don’t always need the behavior to vary that dynamically though – sometimes we just want to be able to apply common patterns in an expressive and efficient manner, once and for all when the class is compiled.

We'll start by briefly explaining what compile-time metaprogramming is, the concept of AST transformations in Groovy, and why they’re important. Then we'll explore most of the transformations Groovy ships with, such as @ToString, @EqualsAndHashCode, and @Lazy, and how these keep your code lean and clean. Next we'll dive into writing your own transformations using the *Local and Global Transformations* mechanism. We'll also show various ways to create an AST, and some tools available for viewing and testing it. We’ll round the chapter off with a discussion of common mistakes and limitations encountered with compile-time metaprogramming.

9.1 A Brief History

You may never have heard the term "compile-time metaprogramming". It has only recently entered the vocabulary of mainstream Groovy developers and some of the more daring Java developers. However, Java has had a long history of code generation: tools and frameworks that automatically create code in the hopes of reducing development time. In the good old days, when CORBA services were the standard remoting technology, it was common to have Java source code automatically generated as part of your build process. More modern applications still do similar things. The common "wsdl2java" and "wsimport" applications read WSDL interface documents and produce source code for projects using web services. This approach is so common that Maven even has a convention for dealing with the files: put them all in a folder called "generated".

9.1.1 Generating bytecode, not source code

The technologies listed so far share a common trait: they all generate source code as part of the build process. Like many other modern languages, Groovy takes a different approach to code generation. Instead of writing out source that the standard compiler can later read and convert to byte code, Groovy lets you, the programmer, get involved in the compilation process.

HOW ARE GETTERS AND SETTERS GENERATED?

In Groovy there is no need to write getters and setters for fields: they will be generated for you. This occurs without a separate source code file listing these getters and setters hidden on the disk somewhere. The Groovy compiler is smart enough to just read your source and write out the correct class definition in the .class file. These changes are all visible from Java or other languages calling your code, because they're part of the compilation process. As far as anything looking at the class is concerned, the getters and setters simply exist as if they'd been hand-written.

From the very beginning, Groovy has made life easier for programmers by manipulating what gets written into the final JVM .class file. The difficulty was that if you wanted a new feature in the language, then you needed to download the Groovy source code and write the feature yourself. But this all changed with the 1.6 release.

9.1.2 Putting the power of code generation in the hands of developers

Groovy 1.6 introduced a feature called *AST Transformations*. The AST part of this is an *Abstract Syntax Tree* – a representation of code as data. This feature allows you to modify the code being generated without ever needing a source code representation. For example, you can add new methods and fields to a class, or add code into to method bodies. Although no source code is generated, the bytecode is present in the final class file in an entirely ordinary way. This is important because it means Java objects calling your Groovy objects will see the new code, which is not the case for changes made through runtime-metaprogramming.

Compile-time metaprogramming is an exciting area of the language. There are many new libraries and frameworks for Groovy that generate verbose, boilerplate code directly into the .class files instead of forcing all the users to write extra source code. Code generation is no longer limited to those brave developers willing to download and build the source code for the Groovy compiler: it's available to anyone using Groovy. If you have a great idea for a new language feature then it's possible to write it today as a library. This powerful technique creates a living language, where you are allowed to extend the language in the direction best suited to your project. Many of Groovy's features are implemented on top of the AST transformation framework. For example, the @Delegate, @Immutable, and @Log annotations all hook into the compiler and affect the final .class file. @Bindable is the secret to UI property binding in the Griffon framework (or writing Groovy Swing in general). The Spock and GContracts libraries both leverage AST Transforms, providing useful and productivity boosting results. Compile-time metaprogramming is used by these libraries to produce more readable tests and more correct runtime behavior.

Before we start writing our own transformations, we'll look at some of the annotations which ship with Groovy, so you can get a feel for what's possible. It's worth bearing in mind any repetitive coding tasks you've recently had to perform – if they sound like the *kind* of work that these annotations help with, you may well be able to eliminate them soon.

9.2 Making Groovy Cleaner and Leaner

Groovy ships with many AST transformations that you can use today to get rid of those annoying bits of repetitive code in your classes. When applied properly, the annotations described here make your code less verbose, so that the bulk of the code expresses meaningful business logic to the *reader* instead of meaningful code templates to the *compiler*. AST transformations cover a wide range of functionality, from generating standard toString() methods, to easing object delegation, to cleaning up Java synchronization constructs, and more. You don't need to know anything about compilers or Groovy internals before using the annotations described in this section: just annotate a class or method and watch your standard code templates disappear.

For the purposes of this section, we've divided the existing AST transformations into six categories:

- Code Generation Transformations
- Class Design Annotations
- Logging Improvements
- Declarative Concurrency
- Easier Cloning and Externalizing
- Safer Scripting

Let's start by looking at some annotations that write code into your class so that you don't have to.

9.2.1 Code Generation Transformations

AST transformations often focus on automating the repetitive task of writing common methods like equals(Object), hashCode(), and constructors; generating the code for you so that you don't have to write it yourself. The built-in annotations in this category are @ToString, @EqualsAndHashCode, @Canonical, @Lazy, @IndexedProperty, @InheritConstructors, and @TupleConstructor.

@GROOVY.TRANSFORM.TOSTRING

Annotating a class with the @ToString annotation gives that class a standard toString() method. @ToString prints out the class name and, by default, all of the field values, as you can see in the simple example from listing 9.1.

Listing 9.1: Using @ToString to generate a toString() method.

```
@groovy.transform.ToString
class Person {
    String first, last
}
def p = new Person(first:'John', last:'Doe')
assert p.toString() == 'Person(John, Doe)'
```

You can use annotation parameters to control the information that toString() displays. For example, you can exclude certain properties, include properties from the super class, and include the property names if you wish. A full description of all the parameters for @ToString appears in Appendix E.

@groovy.transform.EqualsAndHashCode

Implementing the equals() and hashCode() methods correctly is repetitive and error-prone. Luckily the @EqualsAndHashCode annotation does it for you. The generated equals() method obeys the contract of Object.equals(), and hashCode() produces an integer based on the common principles of producing object hashes. Listing 9.2 shows using @EqualsAndHashCode in action on our Person class.

Listing 9.2: Using @EqualsAndHashCode to generate an equals() and hashCode() method.

```
@groovy.transform.EqualsAndHashCode
class Person {
    String first, last
}
def p1 = new Person(first:'John', last: 'Doe')
def p2 = new Person(first:'John', last: 'Doe')
assert p1 == p2
```

You can customize the equals() and hashCode() methods created using three different annotation parameters. You can easily exclude certain properties from the calculation, include properties from the super class, or even include fields in the calculation. A full description of all the available parameters appears in Appendix E.

@groovy.transform.TupleConstructor

Groovy has a flexible syntax for creating objects, such as named arguments and identity blocks. However, sometimes you want the object constructor to take all of the fields explicitly, especially when you're creating the Groovy object from Java code. The @TupleConstructor annotation adds this constructor onto the object, as you can see in listing 9.3.

Listing 9.3: Using @TupleConstructor to generate Java-style constructors.

```
@groovy.transform.TupleConstructor
class Person {
    String first, last
}
def p1 = new Person('John', 'Doe')
def p2 = new Person('John')
```

By default, the overloaded constructors use the declaration order of the properties in order to determine the order of the parameters. However, you can fine tune the exact behavior in a very flexible way. Appendix E provides a full explanation of all the annotation parameters.

@groovy.transform.Canonical

@ToString, @EqualsAndHashCode, and @TupleConstructor are commonly used together to create standard, or canonical, objects. Groovy provides @Canonical to make this a little easier. @Canonical is the combination of all three of these transformations. As you can see in listing 9.4, a canonical object has tuple constructors, equals() and hashCode() implementations, and a standard toString() representation.

Listing 9.4: Using @Canonical to generate equals(), hashCode(), toString(), and constructors.

```
@groovy.transform.Canonical
class Person {
    String first, last
}

def p1 = new Person('John', 'Doe')
def p2 = new Person('John')
assert p1.first == p2.first
assert p1.toString() == 'Person(John, Doe)'
```

Unlike the original annotations @Canonical is based on, the @Canonical annotation does not take any parameters. Instead it uses sensible defaults and let's you override the defaults by using the other annotation in conjunction with @Canonical. For example, if you want to use @Canonical but customize the @ToString behavior, then annotate the class with both @Canonical and @ToString. The @ToString definition and parameters takes precedence over @Canonical. And just what exactly is a sensible default? A complete listing of the default values for @Canonical are described in Appendix E.

@groovy.transform.Lazy

Lazy instantiation is a common idiom in Java. If a field is expensive to create, such as a database connection, then the field is initialized to null, and the actual connection is created only the first time the getter for that field is called. Typical in this idiom is a null check and instantiation within a getter method. But not only is this boilerplate code, it's also frequently done wrong. The @Lazy field annotation correctly delays field instantiation until the time when that field is first used. For example, listing 9.5 constructs a Person object, which would normally throw an exception if the connection field when initialized as part of the object initialization

Listing 9.5: Using @Lazy to delay property instantiation.

```
import java.sql.Connection
import static java.sql.DriverManager.*

class Person {
    @Lazy
    def connection = getConnection('jdbc:odbc:dummy', 'sa', '')
}
assert new Person()
```

Groovy will instantiate the connection the first time the connection property is referenced. If you mark a field as @Lazy but don't provide an initial value, then Groovy tries to create the @Lazy field using the type's default, no-arg constructor. For clarity, and to avoid a missing constructor exception, we recommend you supply an initial value.

Also, by default, this lazy initialization is not thread safe. If you want thread safe lazy initialization then mark the field as transient. The transient keyword is normally used to mark a field as not serializable, but here in the @Lazy annotation the keyword is treated differently. @Lazy transient fields will be initialized from within a synchronized double checked lock block. Double checked locking is an important idiom to write correctly, and using @Lazy with a transient field is an easy way to make sure your code is correct.

There is one parameter called 'soft' available on the annotation. The 'soft' parameter determines if the field should be a SoftReference, and therefore eligible for garbage collection. By default, the field is not a soft reference.

@groovy.transform.InheritConstructors

The @InheritConstructors annotation removes the boilerplate of writing matching constructors for a super class. For example, the java.io.PrintWriter class has eight constructors, and your subclass should probably provide the same set of creation options. @InheritConstructors to the rescue. The annotation creates matching constructors for every super class constructor, as you can see in listing 9.6.

Listing 9.6: Using @InheritConstructors to generate constructors matching the super class's constructors.

```
@groovy.transform.InheritConstructors
class MyPrintWriter extends PrintWriter { }

assert new MyPrintWriter(new File('out.txt'))
```

You can still write your own constructors, of course. If there is a conflict with a super class constructor then @InheritConstructors is smart enough to back off and not overwrite your implementation. A word of warning however: think about your subclass when using this annotation. If your subclass introduces required properties, then it is best to make those properties required in a constructor and not implement too many of the super class constructors. Plus, some Groovy features rely on the availability of a constructor without parameters, so having one on your class is typically a good idea.

That's the last of the code generation annotations in Groovy 1.8. Next up on the tour are some annotations that help you maintain a better designed and more object-oriented system.

9.2.2 Class Design Annotations

Some transformations focus on implementing common design patterns or best practice idioms. The goal is to make the right design decisions also the easiest design to implement. In Java, one of easiest ways to reuse existing code is with a parent class, but just because it's easy doesn't mean that it's the best approach. The annotations in this category are @Delegate, @Singleton, and @Immutable, and their goal is to making the right decision also making the easy decision.

@GROOVY.LANG.DELEGATE

A delegate is a 'has-a' relationship between two classes. Typically, one class will contain a reference to another class and then also share some of the API with that class. The example in listing 9.7 might explain this better.

Listing 9.7: Handwritten delegation.

```
class NoisySet implements Set {
    @Delegate
    Set delegate = new HashSet()

    @Override
    boolean add(i) {
        println "adding $i"
        delegate.add(i)
    }
```

```
        @Override
        boolean addAll(Collection i) {
            for(def x : i) { println "adding $x" }
            delegate.addAll(i)
        }
    }
```

In this example, we have a “noisy set” class that performs some basic println statements when elements are added to the set. A naïve way to code a noisy set is to simply subclass HashSet and override the two methods: but this approach is broken. If NoisySet subclassed HashSet, then every println statement would be called twice each time addAll was invoked. The problem? HashSet implements addAll by calling the add method. This is an implementation detail that is exposed if we choose to subclass instead of use delegation.

The alternative is delegation, our NoisySet has-a HashSet. The @Delegate transformation adds *all* of the public instance methods from the delegate onto your class, and automatically calls the delegate when those methods are invoked. This is how our NosiySet can implement Set yet only declare two methods instead of every method on the Set interface. By default, the owning class is also made to implement all of the interfaces defined by the delegate as well. There could be a conflict between the owner class and one of the delegate methods, or between two of the delegates methods. In that case, the first delegate's method will be delegated to, and the second delegate's method will not. Several annotation parameters are available to fine-tune the behavior, and they are fully described in Appendix E.

@groovy.lang.Singleton

The singleton pattern is intended to ensure that only one instance of a class exists within your system at a time. It requires that the class has a private static reference to this instance, a private constructor so that it cannot be instantiated outside the class, and a public static method to access the single instance. Listing 9.9 shows how to implement a singleton manually in Groovy.

Listing 9.9: Handwritten singleton pattern.

```
    class Zeus {
        static final Zeus instance = new Zeus()
        private Zeus() { }
    }

    assert Zeus.instance
```

This is not too much code to write, particularly as the code generation in Groovy already supplies the accessor method, but it can be simplified using the @Singleton annotation. The obvious advantage is less code, but it also means that invoking the private constructor results in an exception, as shown in listing 9.10.

Listing 9.10: Using @Singleton to enforce a single instance of an object.

```
    @Singleton
    class Zeus {
    }

    assert Zeus.instance
    try {  new Zeus()  }
    catch (RuntimeException e) {  }
```

An additional advantage is that you can use the 'lazy' annotation parameter to properly generate a lazily instantiated instance, which marks the instance variable as volatile and correctly performs double checked locking in the instantiation method. Table 9.7 describes the parameter.

NOTE

The singleton pattern can be useful, but it’s considered by some to be an anti-pattern. Singletons offer no layers of abstraction: it is a concrete type and cannot be extended or easily mocked or changed. Also, improper serialization or multiple classloaders can result in two instances of the Singleton object, and there are also thread safety complications. Singletons are useful, but be aware of the downsides.

@GROOVY.TRANSFORM.IMMUTABLE

Immutable types (such as String) permit no changes in state: when an instance has been created it can never altered. The main advantages of immutability is that the object is side-effect free and thread safe. There is almost no way to change an immutable object from within a method or any way to abuse an immutable object across threads (without resorting to using Reflection, that is). Also, there is never a need to make a defensive copy of an immutable object, or worry about what other objects may have references to your internal state. Working with immutable objects is highly recommended on the Java platform. Groovy provides the groovy.transform.Immutable transformation to help you easily create immutable objects, as shown in listing 9.11.

Listing 9.11: Using @Immutable to mark fields final and suppress setter methods.

```
@groovy.transform.Immutable
class Person {
    String firstName, lastName
}

new Person(firstName: 'Dierk', lastName: 'Koenig')
def p = new Person('Hamlet', "D'Arcy")
assert p.firstName == 'Hamlet'

try { p.firstName = 'John' }
catch (ReadOnlyPropertyException e) { }
```

The Person class has quite a lot of generated code:

- a Map based constructor
- a tuple constructor
- a getter for each property

The @Immutable annotation is very intelligent about which fields to handle. All fields in an @Immutable class must also be marked Immutable, or be of a know immutable type such as a primitive type, String, Color, or URI. Known 'effectively immutable' fields are also handled. Dates are defensively copied in the constructor and getters so that state cannot be changed, and List, Map, and Collection classes are converted to Immutable objects in the constructor. Also, the Immutable annotation shares similar behavior to @ToString and @EqualsAndHashCode: your class receives a nicely formatted toString() method and correct equals(Object) and hashCode() implementations. @Immutable uses sensible defaults for generating the toString(), equals(Object), and hashCode() methods, and they are fulyl described in Appendix E.

WARNING

The Groovy code-base contains two @Immutable annotations: groovy.lang.Immutable and groovy.transform.Immutable. The one in the groovy.lang package is deprecated. Please only use the new one in groovy.transform.

That's the end of the discussion of the class design annotations. Before moving on to concurrency and scripting annotations, let's see some of the new annotation based logging improvements in Groovy.

9.2.3 Logging Improvements

There's still a surprising amount of debate about the best way of logging errors and informative messages from Java, and new logging frameworks are still in development. The @Log family of annotations exists to simplify correct logging idioms from Groovy code. The family includes @Log, @Log4j, @Slf4j, and @Commons.

The annotation does more than just create a logger for you. To understand the power of @Log, consider listing 9.12 and ask yourself if the runLongDatabaseQuery() method will be executed.

Listing 9.12: Using @Log to inject a Logger object into an object.

```
@groovy.util.logging.Log
class Person {
    def method() {
        log.fine(runLongDatabaseQuery())
    }
}
```

```
    new Person().method()
```

From a Java background, the obvious answer would be ‘yes’ - because in Java method arguments are always evaluated before the method is called. There’s no way to avoid this. In Groovy, the answer is 'maybe': it depends on whether the FINE log level is enabled.

The @Log annotation first creates a logger based on the name of your class. It then wraps any logging method with a conditional checking whther that level is enabled before trying to execute the logging line. The result is equivalent to wrapping the logging call in a 'if (logger.isEnabled(LogLevel.FINE))' condition. The arguments to the method may never be evaluated depending on the logging configuration. The transformation is smart too; no check is made if the parameter is a constant such as a simple String or Integer. This improves the readability significantly – there’s no more need to include manual checks everywhere for the sake of performance. Groovy does the correct thing by default.

The @Log family of annotations all take one optional parameter: the name of the log variable. By default the log variable is called 'log', but you can change it to whatever you want. If you don't like how Groovy initializes the Logger object based on the current class name, then add your own logger field and update the annotation to refer to the field name. The four major logging frameworks are covered by Groovy, and each has its own annotation. The four annotations are detailed in table 9.9.

Table 9.9: The Four @Log Annotations

Name	Description
@groovy.util.logging.Log	Injects a static final java.util.logging.Logger into your class and initializes it using Logger.getLogger(class.name).
@groovy.util.logging.Commons	Injects a Apache Commons logger as a static final org.apache.commons.logging.Log into your class and initializes it using LogFactory.getLog(class).
@groovy.util.logging.Log4j	Injects a Log4j logger as a static final org.apache.log4j.Logger into your class and initializes it using Logger.getLogger(class).
@groovy.util.logging.Slf4j	Injects an Slf4j logger as a static final org.slf4j.Logger into your class and initializes it using org.slf4j.LoggerFactory.getLogger(class). The LogBack framework uses Slf4j as the underlying logger, so LogBack users should use @Slf4j.

It doesn't stop there though, because the @Log feature is extensible. You can use your own company's logger as well, as long as you implement one interface in order to define your new annotation. This extension mechanism is how the standard four @Log annotations are implemented, so there are four good examples in the Groovy code base. To implement the interface you need to define a new Logger object and instantiate it, determine if a method should be wrapped in a conditional check, and then wrap the log call in a guard. Writing the AST for this is not hard, but you'll need to understand the rest of the chapter before tackling the problem.

Next we’ll look at declarative concurrency. Groovy provides annotations to declare how your code is locked during multi-threaded access instead of writing the code that performs low level locking.

9.2.4 Declarative Concurrency

Synchronization and access to mutable state is hard to get right. Proper synchronization can leave your little branch of business logic hidden, surrounded by a forest of lock acquire and lock release code. The concurrency related annotations aim to remedy this problem: @Synchronized, @WithReadLock, and @WithWriteLock.

@GROOVY.TRANSFORM.SYNCHRONIZED

Code that is accessed from several threads at once often needs to be synchronized to avoid common concurrency problems. One problem with this is that correct concurrent code is hard: it’s all too easy to introduce one problem when trying to solve another. The easiest solution for Java developers is to add the 'synchronized' keyword to the method declaration. This is another instance where the *easiest* solution is not the *best* solution.

AVOID LOW LEVEL SYNCHRONIZATION

Java contains many fine primitives for working with concurrent code, such as the synchronized keyword and the contents of the java.util.concurrent package. However, these are mostly primitives, and not abstractions. The tools are low level and meant to serve as a foundation. GPars is a framework for parallelization that is built on top of these primitives. It provides many abstractions that shield you from low level coordination tasks. GPars is described fully in chapter 17Todo: add dynamic chapter ref.

The problem with method level synchronization is that it is very coarse grained and it's also part of the public API of the object. You're effectively locking on a publicly accessible references: the this reference. Some secure coding standards ban method level synchronization or synchronization on the 'this' reference because an attacker that has a reference to your object can interfere with your synchronization by synchronizing on it. It is best to declare a local, private lock and expose that lock to subclasses if classes need to coordinate locking. Doing this correctly is easy with the @Synchronized annotation, as seen in listing 9.15.

Listing 9.15: Declarative synchronization with @Synchronized

```
class Person {
  private final phoneNumbers = [:]

    @groovy.transform.Synchronized
    def getPhoneNumber(key) {
        phoneNumbers[key]
    }
    @groovy.transform.Synchronized
    def addPhoneNumber(key, value) {
        phoneNumbers[key] = value
    }
}
```

This annotation injects a lock object into your class. The object is a zero length Object array so that your class remains Serializable (which Object is not). And any method marked with the annotation has a synchronized block around it but without method synchronization. If you want to limit the scope of your synchronized block, then provide a name for the lock using the default annotation parameter and write the synchronized block yourself when needed, as shown in listing 9.16.

Listing 9.16: Mixing @Synchronized with custom synchronized block.

```
@groovy.util.logging.Log
class Person {
    private final phoneNumbers = [:]
    private final lock = new Object[0]

    @groovy.transform.Synchronized('lock')
    def getPhoneNumber(key) {
        phoneNumbers[key]
    }

    def addPhoneNumber(key, value) {
        log.info("Adding phone number $value")
        synchronized (lock) {
            phoneNumbers[key] = value
        }
    }
}
```

Synchronization is a low level, primitive operation. Java has some higher level locking mechanisms as well, and the following two annotations help make them easy to use.

@GROOVY.TRANSFORM.WITHREADLOCK AND @GROOVY.TRANSFORM.WITHWRITELOCK

Java 5 included the java.util.concurrent.locks.ReentrantReadWriteLock class as a tool to use when you need more control over locking than simply using synchronized blocks. A ReentrantReadWriteLock can guard against either read access or write access, where many readers are allowed concurrently, but only one writer is allowed. Although this is a very useful concurrency abstraction, acquiring and release a lock correctly is cumbersome, as you can see in listing 9.17.

Listing 9.17: Hand-written read and write locking.

```
import java.util.concurrent.locks.ReentrantReadWriteLock
class Person {
    private final phoneNumbers = [:]
    final private lock = new ReentrantReadWriteLock()
    def getPhoneNumber(key) {
        lock.readLock.acquire()
        try {
            phoneNumbers[key]
        } finally {
            lock.readLock.release()
        }
    }
    def addPhoneNumber(key, value) {
        lock.writeLock.acquire()
        try {
            phoneNumbers[key] = value
        } finally {
            lock.writeLock.release()
        }
    }
}
```

Phew, that's quite a bit of code. It *does* do the right thing: reading data is guarded with a read lock and writing data is guarded with a write lock. However, the code is much simpler when we use the @WithReadLock and @WithWriteLock annotations instead.

Listing 9.18: Declarative read and write locking using @WithReadLock and @WithWriteLock.

```
class Person {
    private final phoneNumbers = [:]

    @groovy.transform.WithReadLock
    def getPhoneNumber(key) {
        phoneNumbers[key]
    }
    @groovy.transform.WithWriteLock
    def addPhoneNumber(key, value) {
        phoneNumbers[key] = value
    }
}
```

This time the logic of the class stands out instead of being drowned in a sea of try/finally blocks, and you'll never forget to release a lock. Similar to @Synchronized, these annotations take a parameter for the lock name, and that lock will be used if it exists in the class.

These examples all show how annotations for AST transformations work. There are other ways to be thread safe as well. For instance, you could use a ConcurrentHashMap for thread safety or use immutable objects. The value in the Groovy annotation approach is that synchronization and safety are declarative. You don't explain how the synchronization works, you just declare that it exists and let Groovy do the rest.

In general, declarative solutions offer good abstractions, where you don't need to see the details and can focus on the more important parts of the code instead of the low level mechanics. The same is true for other areas where you traditionally end up with a lot of boilerplate code to write. Each *individual* bit of boilerplate is simple enough, but after you've written it enough times you're bound to make a subtle mistake – and it really impacts the readability of the class. The same idea extends to other operations we might wish to perform on our objects, too.

9.2.5 Easier Cloning and Externalizing

Implementing Cloneable and Externalizable correctly is not always simple. The @AutoClone annotation can give you a reasonable and configurable cloning strategy by adding just the annotation. In a similar vein, @AutoExternalize makes implementing Externalizable simpler by correctly creating default read and write methods.

@GROOVY.TRANSFORM.AUTOCLONE

Classes that implement Cloneable should provide a public clone method that creates a copy of the class. At its simplest, the @AutoClone annotation causes your class to implement Cloneable and provides a default and simple clone method implementation. The super class clone() method is invoked, followed by invoking clone() on each Cloneable field or property in the class. If a field or property is not Cloneable then it is simply copied in a bitwise fashion. If some properties don't support cloning, then a CloneNotSupportedException is thrown. Deep copies are left to the end user (you) to implement. For non-trivial objects, it is best to write your own clone method so that you can have finer control.

A popular alternative implementation for the clone() method is called the 'Copy Constructor' style. The clone method implementation is moved into the body of a constructor that takes a parameter of the same type as the class. Then calls to clone() simply return the result of calling this constructor. Marking a simple Person class with @AutoClone logically produces code similar to listing 9.19.

Listing 9.19: Handwritten 'Copy Constructor' style cloning.

```
class Person implements Cloneable {

    String firstName, lastName
    Date birthday

    protected Person(Person other) throws CloneNotSupportedException {
        first = other.first
        last = other.last
        birthday = other.birthday.clone()
    }
    Object clone() throws CloneNotSupportedException {
        new Person(this)
    }
}
```

One other style that is supported by @AutoClone is the Serialization style. If a class already implements Serializable, then that mechanism can be used to create a copy of the instance. This feature performs deep copy automatically, attempting to copy the entire tree of objects including array and list elements. The generated clone() method looks something like listing 9.20.

Listing 9.20: Handwritten 'Serialization' style cloning.

```
Object clone() throws CloneNotSupportedException {
     def baos = new ByteArrayOutputStream()
     baos.withObjectOutputStream{ it.writeObject(this) }
     def bais = new ByteArrayInputStream(baos.toByteArray())
     bais.withObjectInputStream(getClass().classLoader){ it.readObject() }
}
```

There are some downsides to the Serialization style. It's typically slower, doesn't allow final fields, and takes up more memory than the alternatives.

You can use several annotation parameters to fine tune AutoClone, and these parameters are described in Appendix E.

@groovy.transform.AutoExternalize

The Externalizable interface is similar to Serializable in that it is used to persists objects into a binary form. Externalizable was added to the JDK after Serializable. The new interface gives you more control over the persisted form than Serializable does, and it does not use reflection, which at one time was a performance bottleneck. Some performance sensitive applications prefer using Externalizable.

A class marked @AutoExternalize automatically implements the Externalizable interface, gaining two new method implementations: readExternal(ObjectInput) and writeExternal(ObjectOutput). Our Person example with three properties would have methods generated similar to those in listing 9.21.

Listing 9.21: Handwritten Externalizable implementation.

```
class Person implements Externalizable {
    String firstName, lastName
    Date birthday
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(firstName)
        out.writeObject(lastName)
        out.writeObject(birthday)
    }
    public void readExternal(ObjectInput oin) {
        firstName = oin.readObject()
        lastName = oin.readObject()
        birthday = oin.readObject()
    }
}
```

You can find the the @AutoExternalize behavior using the annotation parameters described in Appendix E.

That's all for the cloning and externalizing annotations. The next set of annotations we'll discuss exist to make using Groovy as a scripting language safe and secure.

9.2.6 Safer Scripting

Security and robustness are important aspects of modern software. Groovy makes it easy to run scripts submitted by your users (as well as your own scripts), but this can be a security hole which needs to be shielded not only against unauthorized access but also against accidental programming errors. No one wants a set of long running scripts to cause a denial of service. These scripting annotations automatically add safety hooks into scripts so that they timeout, respect a thread interrupt, or otherwise behave correctly. They are designed to be automatically added to scripts executing in GroovyShell or another evaluator, but you can also use them yourself on your own scripts and classes.

@GROOVY.TRANSFORM.TIMEDINTERRUPT

Annotating a class with @TimedInterrupt sets a maximum time the script or instances of the class are allowed to exist. If the maximum time is exceeded then a TimeoutException is thrown. This annotation is designed to guard against runaway processes, infinite loops, or a maliciously long running user script.

When annotated, the object instance marks the instantiation time in the constructor. If this instance later detects that the maximum run time is exceeded then it throws an exception. Checks are made at the beginning of every method call, the first line of every closure, and within every iteration of a for or while loop. If the object sits idle and is never invoked, then no exception is thrown regardless of how much time passes.

There are a variety annotation parameters you can use to fine tune the behavior, which are described in Appendix E.

@groovy.transform.ThreadInterrupt

Long running user scripts *should* periodically check the Thread.currentThread().isInterrupted() status and throw InterruptedExceptions when an interrupt is detected. However, in practice scripts are almost never written this way. An easy way to properly respect the interrupted flag is to use the @ThreadInterrupt annotation. When this annotation is present, your script or class will automatically check the isInterrupted() flag and throw an InterruptedException if the thread is interrupted. These checks occur at the start of every method call, at the start of every closure, and within every iteration of a loop.

Similar to @TimedInterrupt, there are some parameters you can use to tweak the behavior of @ThreadInterrupt, which are detailed in Appendix E.

@groovy.transform.ConditionalInterrupt

The last annotation in the Interrupt family is @ConditionalInterrupt. This annotation allows you to specify your own custom interrupt logic to be weaved into a class. Like the others, the interrupt check occurs at the start of every method, the start of every closure, and each iteration of a loop.

The way you specify the conditional interrupt is within a closure annotation parameter. You can reference any variable that is in scope within this closure. For scripts, general script variables are in scope, and for classes instance field are in scope. Listing 9.22 shows a script that executes some work 1000 times or until 10 exceptions have been thrown, whichever comes sooner.

Listing 9.22: Using @ConditionalInterrupt to set an automatic error threshold.

```
@ConditionalInterrupt({ errorCount >= 10})
import groovy.transform.ConditionalInterrupt

errorCount = 0

1000.times {
    try {
         // do some work
    } catch (Throwable t) {
        errorCount++
    }
}
```

As with all the Interrupt annotation, there are a variety of parameters you can use to tweak the functionality, as described in Appendix E.

9.2.7 And More Transformations

There are other transformations as well, but they are covered elsewhere in the book. @PackageScope has already been discussed in Chapter X, and @Category and @Mixin are covered in Chapter X. See the Swing chapter for a discussion of @Bindable, @Vetoable, and @ListenerList. @Newify is covered later in the DSL chapter, and @Field is covered in Chapter X.

That's the end of our tour of the AST transformations that come with Groovy. You can go and use these annotations today without knowing much more. But you don't have to be satisfied with just what Groovy gives you. You're free to write your own annotation as well. The rest of this chapter delves into the task of implementing your own annotations using AST transformations. We're going to discuss local and global transformations, writing your own AST, testing your work, and the known limitations.

So why exactly would you want to write your own AST transformation? There are some good reasons to use compile-time metaprogramming instead of runtime. If you want Java to see the dynamic changes you make to a Groovy class, then use an AST transformation to write your changes directly into the produced class file. The code generation transformations are good examples of doing this. If you need to avoid evaluating a method parameter before a method is invoked, then use an AST Transform to avoid or wrap the call, as the @Log transformation does. And as we'll see later, you may also find compile-time metaprogramming a good fit for advanced or fine grained control over domain specific languages (DSLs). Lastly, if you want to do something wildly different, like change the semantics of the language, then your best approach is an AST Transform. Let's get into some deeper explanations and in-depth examples.

9.3 AST by Example: Local Transformations

All of the examples presented so far, such as @ToString and @Canonical, are known as "local transformations". A local transformation relies on annotations to rewrite Groovy code. There are other forms of transformations as well, however a local transformation has the advantage of being the easiest to write: Groovy takes care of instantiating and invoking your transformation correctly, as well as making sure to *avoid* calling it when it is not needed. Features written as local transformations modify the class generated by Groovy and are activated by annotating either a method or a class.

Let's start our exploration with a simple initial example of a “Local Transformation”. To demonstrate a local transformation, we are going to create a method annotation that marks a method as being a Java main method. The transformation will automatically give the JVM class file a main method that can be a public entry point to run the class, and that main method will contain a copy of the code defined in the greet() method. Listing 9.23 contains the contents of Greeter.groovy, which shows how to use the annotation to annotate a method.

Listing 9.23: Annotating a public method as a main(String[]) method in Greeter.groovy.

```
package examples

class Greeter {

    @Main
    def greet() {
        println "Hello from the greet() method!"
    }
}
```

After we are finished, we will have written a transformation that creates a main method on the Greeter class. This main method will create an instance of the Greeter class and then invoke the greet() method on it. Our example has a limitation: the enclosing class must have a parameterless constructor that creates the object in a usable state. We have to have an instance in order to call an instance method, of course – and requiring a parameterless constructor just keeps things simple. Of course, another alternative would be to integrate with a Dependency Injection framework such as Guice… or to make the annotated method static instead. Listing 24 shows how we want the class to be invoked and run from Java, which relies on a public static void main(String[]) method existing in the class.

Listing 9.24: Expected output when using the @Main annotation

```
$ java -cp groovy-all-1.8.0.jar examples.Greeter
Hello from the greet() method!
```

From the sample usage we can glean some information about the objects involved. We need to define an annotation called Main, and that must trigger the AST transformation to create the main method. There isn't much more to it than that. Creating and invoking the object is all done internally by Groovy. Figure 9.1 shows the classes involved with a local AST transformation.

[TODO: Use Standard Tools for this]

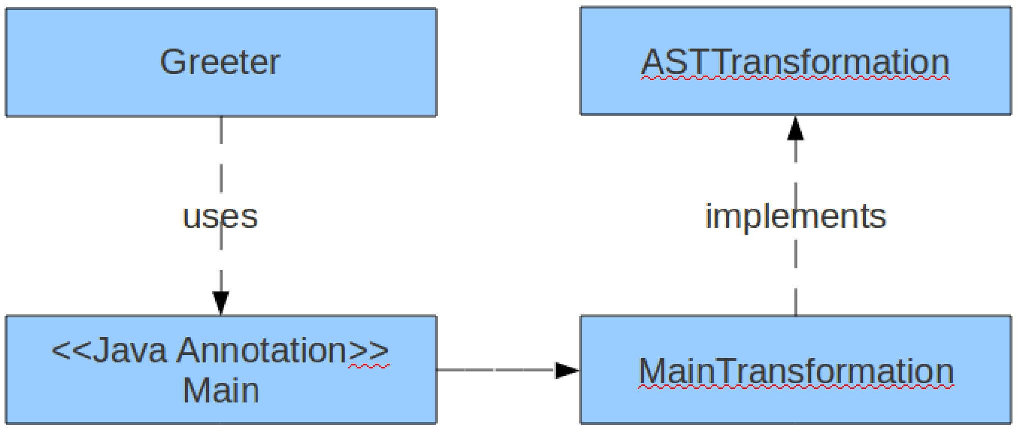


Figure 9.1: Classes involved with the @Main local AST transformations.

We're going to define the @Main annotation as a standard Java annotation; there is no Groovy magic involved. The retention policy should be SOURCE, meaning that Java does not carry the presence of this annotation through to the final class file. The target element type specifies what the annotation can be applied to – so in this case, we're going to use METHOD.

The only thing special about this annotation definition is the use of the GroovyASTTransformationClass annotation. This specifies the class which implements the logic of the AST transformation, and is how the compiler binds the pieces together. The argument for GroovyASTTransformationClass is a list of the the classes you want invoked when the annotation is found by the Groovy compiler. Listing 9.25 shows the definition of an @Main annotation defined in Main.groovy.

Listing 9.25: Defining an Annotation to be used in a local AST transformation in Main.groovy

```
package examples

import org.codehaus.groovy.transform.GroovyASTTransformationClass
import java.lang.annotation.*

@Retention (RetentionPolicy.SOURCE)
@Target ([ElementType.METHOD])
@GroovyASTTransformationClass (classes=[examples.MainTransformation])
public @interface Main { }
```

Now the only piece left to implement is the MainTransformation class. This will be instantiated and invoked by Groovy when your annotation is encountered. The ASTTransformation interface we have to implement has a single method, called visit. Typically, you only need to use the ASTNode[] parameter. Element 0 contains the annotation that triggered the transformation and element 1 contains the ASTNode that was annotated. The skeleton of the @Main algorithm is simple:

- find the method that was annotated with @Main (in this case greet())
- get a reference to the enclosing class (Greeter)
- create a synthetic public static void main method and instantiate the Greeter instance within it
- invoke the greet() method
- add the new method onto the Greeter class.

The implementation is shown in listing 9.26.

Listing 9.26: Implementing the ASTTransformation for the @Main annotation.

```
package examples

import org.codehaus.groovy.control.CompilePhase
import org.codehaus.groovy.transform.*
import org.codehaus.groovy.ast.*
import org.codehaus.groovy.control.SourceUnit
import org.codehaus.groovy.ast.builder.AstBuilder
import org.objectweb.asm.Opcodes

@GroovyASTTransformation(phase = CompilePhase.INSTRUCTION_SELECTION)
public class MainTransformation implements ASTTransformation {

    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
        // use guard clauses as a form of defensive programming.
        if (!astNodes) return
        if (!astNodes[0] || !astNodes[1]) return
        if (!(astNodes[0] instanceof AnnotationNode)) return
        if (astNodes[0].classNode?.name != Main.class.name) return
        if (!(astNodes[1] instanceof MethodNode)) return

        MethodNode annotatedMethod = astNodes[1]
        ClassNode declaringClass = annotatedMethod.declaringClass
        MethodNode mainMethod = makeMainMethod(annotatedMethod)
        declaringClass.addMethod(mainMethod)
    }

    MethodNode makeMainMethod(MethodNode source) {
            def className = source.declaringClass.name
            def methodName = source.name

            def phase = CompilePhase.INSTRUCTION_SELECTION
            def ast = new AstBuilder().buildFromString(phase, false, """
                package $source.declaringClass.packageName

                class $source.declaringClass.nameWithoutPackage {
                    public static void main(String[] args) {
                        new $className().$methodName()
                    }
                }
            """)
            ast[1].methods.find { it.name == 'main' }
        }
    }
```

There are several important things to note about this example. First, the class is annotated with @GroovyASTTransformation, which tells the Groovy compiler the phase in which we want our transformation to be invoked. This is a required annotation for a transformation and must be Semantic Analysis or later. It can't be any earlier than that because your original annotation would not be compiled at that point. There is a bit of a chicken and egg problem with trying to go earlier.

The makeMainMethod method is especially interesting, because it shows how to create ASTNode objects. You can call constructors directly, and with an IDE this is often the easiest way to create the AST you need. However, you're also free to use the AstBuilder object, which has an API to simplify the creation of AST nodes. We'll cover it in more depth later, but in this example we're creating a new method that is public, static, and void, and has the appropriate String[] parameter. The body of the method instantiates an instance of our annotated class and invokes the annotated method on it.

This code sample does *some* error checking on the input because in production code it is always best to state your assumptions with a few assertions or guard clauses. It may not be obvious, but there are quite a few assumptions made in this small example. For instance, what happens if a class has two methods annotated with @Main? What happens if the class already has a main method defined? You should probably report these as compile errors using SourceUnit.addError(SyntaxException) or SourceUnit.addException(Exception). What happens if there is no parameterless constructor? As you can see, the edge cases of writing transformations can sometimes be challenging. Writing an AST transformation forces you to sit and think about just what *could* happen in the language, and you'll come away from the experience with a much better understanding of Groovy.

When you write an AST transformation, you will need to decide which compiler phase to target. The choice depends on what you're trying to do to the AST. A full description of the different compiler phases, as well as some hints for choosing which phase to target, appears in Appendix F.

Local transformations require an annotation, which is not particularly limiting when you consider that Groovy annotations are more flexible than Java annotations. They can appear in more places within a source file than in Java, including import statements. When in doubt choose a local transformation because it is the easiest to write. You can always refactor to a global transformation or hard-code a transformation into a classloader later.

9.4 AST by Example: Global Transformations

Global transformations are similar to local transformations except that no annotation is required to wire-in a visitor. Instead of having the end user specify when your transformation is applied, global transformations are simply applied to every single source unit in the compilation. Global transformations can also be applied to any phase in the compilation, even those before semantic analysis. With this flexibility comes a performance penalty. All compilations will take longer, even if your transformation is not used. For this reason you should use global transformations with reticence and consider implementing global transformations in Java for the performance benefits.

Global transformations are specified in jar file metadata. To deploy a global transformation it must be packaged into a jar file, and the META-INF metadata must specify the fully-qualified path of your transformation class. Let's see this in action with an example. Imagine a transformation that adds a static method to every class that returns the date and time of the compilation as a String. You could use it from a script as in listing 9.27.

Listing 9.27: Example showing usage of getCompiledTime() on a class.

```
println 'script compiled at: ' + compiledTime
class MyClass { }
println 'script class compiled at: ' + MyClass.compiledTime
```

Remember, free standing scripts without classes still get generated into a Script subclass during complication, so adding a getCompiledTime() method to every Class in the SourceUnit should be enough to accomplish this feature. For this transformation we're going to add a public static method called "getCompiledTime" to every class in the SourceUnit, and it will simply return the date of compilation as a String. In local transformations we manipulated the supplied ASTNode[] to find the context in which we were invoked. For global transformations this array holds little of interest. Instead we need to query the SourceUnit to find our source AST. It contains all the classes that were defined in the file along with the script, which itself is a class of type Script.

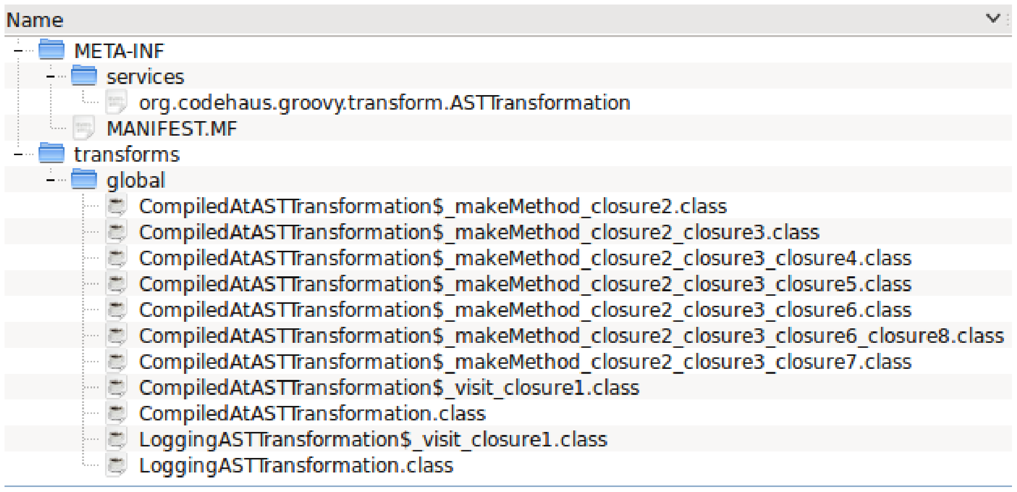
Listing 9.28 shows the implementation of a global transformation.

Listing 9.28: Adding a new method to a class using a global AST transformation.

```
package transforms.global

import org.codehaus.groovy.ast.*
import org.codehaus.groovy.transform.*
import org.codehaus.groovy.control.*
import org.codehaus.groovy.ast.expr.*
import org.codehaus.groovy.ast.stmt.*
import java.lang.annotation.*
import org.codehaus.groovy.ast.builder.AstBuilder
import static org.objectweb.asm.Opcodes.*

@GroovyASTTransformation(phase=CompilePhase.CONVERSION)
public class CompiledAtASTTransformation implements ASTTransformation {

    private final static compileTime = new Date().toString()

    public void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
        List classes = sourceUnit.ast?.classes
        classes.each { ClassNode clazz ->
           clazz.addMethod(makeMethod())
        }
    }

    MethodNode makeMethod() {

        def ast = new AstBuilder().buildFromSpec {
            method('getCompiledTime', ACC_PUBLIC | ACC_STATIC, String) {
                parameters {}
                exceptions {}
                block {
                    returnStatement {
                        constant(compileTime)
                    }
                }
                annotations {}
            }
        }
        ast[0]
    }
}
```

For simplicity, the error checking was left out of the example; in a real transformation you'd apply similar guard clauses to the ones we used in the @Main example. Now all we need to do is to tell the compiler about our transformation so it can be applied appropriately.

The first requirement is that the transformation must be deployed in a jar file. The jar must contain your AST transformation class and a special file named org.codehaus.groovy.transform.ASTTransformation in the META-INF/services directory. The services directory is part of the jar file specification, and it contains configuration files like the one we need to create. The configuration file is a simple text file, and each line is a fully-qualified class name of an AST transformation. The file content for our transformation needs to contain the line “transforms.global.CompiledAtASTTransformation”. The full contents of our jar file, including classes and services, can be seen in figure 9.x. The jar must contain all our classes plus our new text file named "org.codehaus.groovy.transform.ASTTransformation".



```
Name
META-INF
  services
    org.codehaus.groovy.transform.ASTTransformation
  MANIFEST.MF
transforms
  global
    CompiledAtASTTransformation$_makeMethod_closure2.class
    CompiledAtASTTransformation$_makeMethod_closure2_closure3.class
    CompiledAtASTTransformation$_makeMethod_closure2_closure3_closure4.class
    CompiledAtASTTransformation$_makeMethod_closure2_closure3_closure5.class
    CompiledAtASTTransformation$_makeMethod_closure2_closure3_closure6.class
    CompiledAtASTTransformation$_makeMethod_closure2_closure3_closure6_closure8.class
    CompiledAtASTTransformation$_makeMethod_closure2_closure3_closure7.class
    CompiledAtASTTransformation$_visit_closure1.class
    CompiledAtASTTransformation.class
    LoggingASTTransformation$_visit_closure1.class
    LoggingASTTransformation.class
```

Figure 9.x: Contents of LoggingTransform.jar

The name of the jar file does not matter. As long as it is on the class path during compilation, the Groovy compiler will read the configuration file and apply any transformations listed within it.

The fact that the transformation must be in a jar file has an effect on your project structure. If you define a global transformation and want to use it on your project, then the transformation jar must be built before the compilation of the rest of your project. For most build tools and IDEs this means creating a separate project for the transformation, possibly along with a separate build script.

ERROR REPORTING

Errors can be reported using the addError and addException methods on SourceUnit, however it is much better to use an ErrorHandler, which can be retrieved from the SourceUnit with the getErrorHandler() method. This object collects all of the error messages during the compile and has a broader API. There are methods to add an error and fail the build, add an error but continue, add warnings and more. And please, for the sake of your users, always add error messages that contain a good description of what happened, the conditions that caused the error, and the line number where the error occurred. If your users really wanted cryptic or bizarre compilation failures, they’d be using C++.

There is one more feature of global AST transformations which DSL writers find useful. You can specify a file extension to which the Groovy compiler will automatically apply your transformation. For example, Groovy++ applies a global transformation to all files ending in .gpp and .grunit. The mechanism to define a file extension is similar to defining the transformation. Simply write the file extensions (without any wildcards) into a file called "org.codehaus.groovy.source.Extensions" and include it in your jar next to the "org.codehaus.groovy.transform.ASTTransformation file. Each line of the file should list a file extension without any sort of wildcards attached. Your final jar file needs to contain both the ASTTransformation and the extensions configuration file, plus any required .class files.

By now you've seen both local and global transformations and how they are implemented. You've also heard the term *abstract syntax tree* several times, usually abbreviated as AST – but we haven't really looked at what it really means. It's time to take a deep dive into AST and the Groovy compiler.

9.5 Exploring AST

To understand how this neat stuff is implemented, we'll need at least a little knowledge of the Groovy compiler. In this section we'll define and explain abstract syntax trees (ASTs), show some of the AST visualization tools available for the platform, and cover the basics of the Groovy compiler.

An abstract syntax tree is a representation of your program in tree form. The tree has nodes which can have leaves and branches, and there's a single root node. Many compilers, not just Groovy, create an AST as a step towards a compiled program. In general and simplified terms, running a Groovy script is a five step process, as shown in figure 9.2.

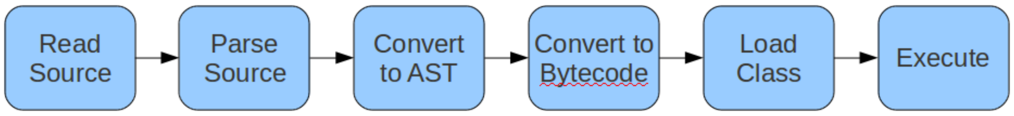


Figure 9.2: The general process of running a Groovy script

First the Groovy compiler reads the source file and checks it for basic forms of validity. Then the source code is converted into an AST, which is eventually converted to bytecode. Finally the JVM loads the class and executes it. The AST is where all of the interesting language stuff happens. For example, adding getters and setters for properties happens in AST and giving a script a main() method happens there. If you want to write a language feature, then you will quite possibly be working with the AST. Let's see some simple AST examples to help understand it better. Figure 9.3 shows the tree representing the expression "1 + 1", the simplest non-trivial example.

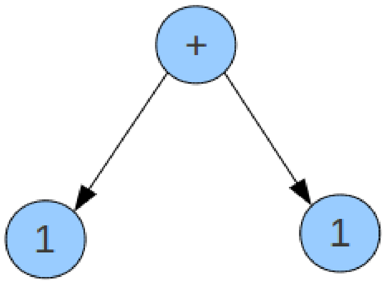


Figure 9.3: An abstract syntax tree for the expression '1 + 1'

The Plus operation is a *binary* one: it has two operands, a left and a right. When this program is executed, a plus operation is executed and (hopefully) the result is 2. Figure 9.4 shows a slightly more advanced example: the expression "1 + 2 + 3".

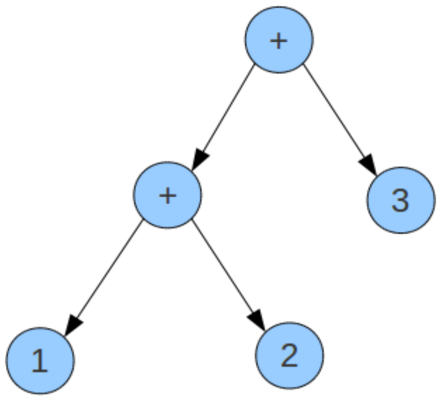


Figure 9.4: The abstract syntax tree for the expression '1 + 2 + 3'

It is no accident that the branch "1 + 2" forms the left most branch. Addition is associated left to right, and to compute the answer to "1 + 2 + 3" you must first compute "1 + 2". Only then will you be able to evaluate "3 + 3" and see the result as 6. The final figure (figure 9.5) shows a more realistic example, the groovy script "assert 1 + 1 == 2".

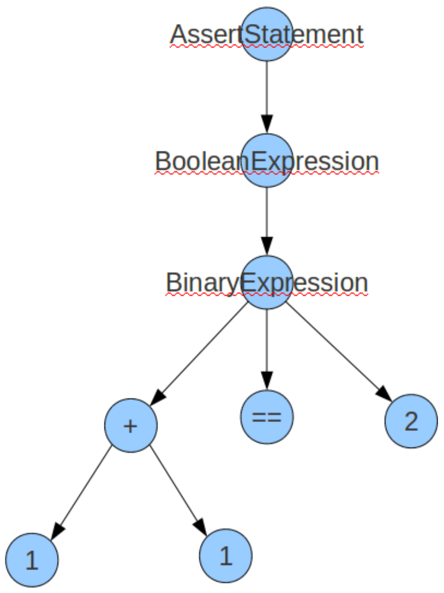


Figure 9.5: The abstract syntax tree for the expression 'assert 1 + 1 == 2'

In Groovy terms, this script creates an AssertStatement, which has a BooleanExpression, which in turn has a BinaryExpression. It is this BinaryExpression which holds the "==" equals operator. Entire programs are easily represented in tree form, and the tree can be analyzed, navigated, and transformed as part of the compilation.

Each language (or compiler, really) has its own tree structure, and its up to the AST implementors to determine the exact structure. In Groovy, each element of the tree is an instance of the class ASTNode, and there is a subclass for everything in the language: BooleanExpression, ForStatement, WhileStatement, ClosureExpression, to name but a few. There are over 75 subclasses of ASTNode, and having a good IDE to help you navigate the class hierarchy is highly recommended. Just point your IDE to the Groovy sources and you should be fine; some IDEs will automatically download them for you.

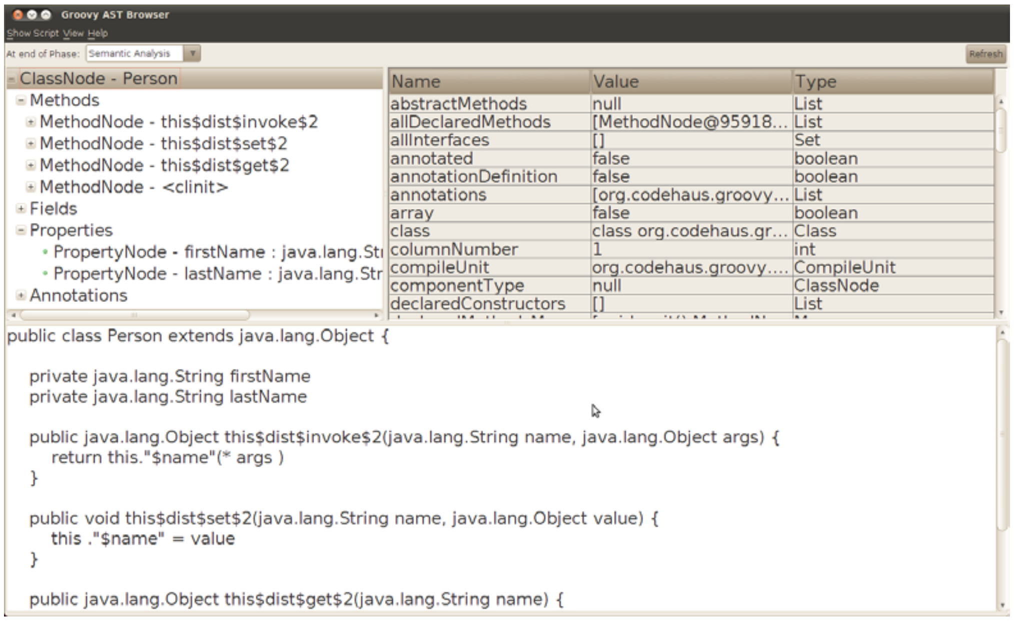
HOMOGENEOUS VS. HETEROGENEOUS AST

The term for having one subclass of ASTNode for each language element is "Heterogeneous AST". The tree is populated with many different types, and analyzing the tree means reading the type of the tree leaves, not just reading the leaf properties. The javac compiler from Oracle also uses a heterogeneous AST. The advantage is that it's easy to store and retrieve node specific data from the AST leaves. Other languages use a *homogeneous* AST, where every node in the tree is the same type. Imagine if all the objects in the entire tree were of the concrete type ASTNode. The main advantage in this approach is that tree visitors are trivial to write, but static analysis is more difficult. The book "Language Implementation Patterns" by Terence Parr offers excellent in-depth coverage of ASTs.

9.5.1 Tools of the Trade

At this point you no doubt have many questions about Groovy, ASTs, and class files. Groovy compiles to Java class files, so if you want to analyze bytecode of a .class file you can always use javap from the JDK if you really want to. However, that’s a very low level approach, which can be time-consuming and frustrating – particularly if you’re trying to examine code of any significant size. Luckily, there are many tools at your disposal if you're interested in digging a little deeper into how things work. If you plan on working with compile-time meta-programming then each of the tools listed here will be an invaluable asset in your toolbox.Groovy Console's AST Browser and Source Viewer

GroovyConsole, which is included in the Groovy installation, contains a tool that lets you view and analyze the AST of a Groovy script. Once you have GroovyConsole open, you can analyze any script using the menu item Script-> View AST or the Ctrl + T shortcut (Cmd + T on Macs). The window that opens is called the AST Browser. The AST Browser has three parts: the tree view, the property table, and the decompiled source view, as shown in figure 9.6.



```
ClassNode - Person
  Methods
    MethodNode - this$dist$invoke$2
    MethodNode - this$dist$set$2
    MethodNode - this$dist$get$2
    MethodNode - <clinit>
  Fields
  Properties
    PropertyNode - firstName : java.lang.Str
    PropertyNode - lastName : java.lang.Str
  Annotations
```

Name	Value	Type
abstractMethods	null	List
allDeclaredMethods	[MethodNode@95918...	List
allInterfaces	[]	Set
annotated	false	boolean
annotationDefinition	false	boolean
annotations	[org.codehaus.groovy...	List
array	false	boolean
class	class org.codehaus.gr...	Class
columnNumber	1	int
compileUnit	org.codehaus.groovy....	CompileUnit
componentType	null	ClassNode
declaredConstructors	[]	List

```
public class Person extends java.lang.Object {

    private java.lang.String firstName
    private java.lang.String lastName

    public java.lang.Object this$dist$invoke$2(java.lang.String name, java.lang.Object args) {
        return this."$name"(* args )
    }

    public void this$dist$set$2(java.lang.String name, java.lang.Object value) {
        this ."$name" = value
    }

    public java.lang.Object this$dist$get$2(java.lang.String name) {
```

Figure 9.6: Groovy's standard AST Browser

The tree view displays the abstract syntax tree of your script using a standard tree widget. You can expand and navigate nodes, and otherwise explore the AST. As you click a tree node, the property table on the right lists all of the properties of the node, and the main GroovyConsole window highlights the source code corresponding to that AST node. The decompiled source view, along the bottom of the window, displays the AST rendered as Groovy source code. The generated source is perhaps the easiest way to understand what the AST contains because it is much easier to read source code than a tree component. Consider again our Person class annotated with @ToString, repeated again in listing 9.35.

Listing 9.35: A simple person class annotated with @ToString

```
@groovy.transform.ToString
class Person {
   String firstName, lastName
}
```

Within the AST Browser, if you change the Phase drop down list to Canonicalization, then the decompiled source pane renders the Person class as along with the toString() implementation, shown in listing 9.36.

Listing 9.36: What the Groovy compiler sees for a @ToString annotated Person.

```
public class Person extends java.lang.Object {
   private java.lang.String firstName
   private java.lang.String lastName
   private boolean $print$names

   public java.lang.String toString() {
       java.lang.Object _result = new java.lang.StringBuffer()
       _result.append('Person')
       _result.append('(')
       if ( $print$names ) {
           _result.append('firstName')
           _result.append(':')
       }

        _result.append(org.codehaus.groovy.runtime.InvokerHelper.toString(
                                 firstName))
       _result.append(', ')
       if ( $print$names ) {
           _result.append('lastName')
           _result.append(':')
       }

    _result.append(org.codehaus.groovy.runtime.InvokerHelper.toString(
                                 lastName))
       _result.append(')')
       return _result.toString()
   }
}
```

As you can see, the @ToString annotation adds a field to our class called "$print$names" and the toString() method reads the fields and creates a toString implementation. The decompiled source viewer is one of the best features to learn and understand how Groovy works. Even if you're not attempting to write AST transformations, it can be a real learning experience to see how different pieces of Groovy source get transformed into the final output. Other Tools

The last important tools are a good decompiler and debugger. A decompiler will reverse engineer the source code out of a class file, and the results can be quite amazing. Any Java decompiler should work perfectly well with Groovy, and there are many open source and free ones to choose from. My favorite is the JD-GUI, which is free for non-commercial use but is not open source. Also a good IDE and debugger aide greatly when exploring the large ASTNode class hierarchy. When there are over 75 subclasses to navigate, it is important to be able to quickly find the source code you need and view the current state of instances. There are several open source and free options available for IDEs with great Groovy support.

We've seen a lot already: local transformations, global transformations, and some useful tools. Through the examples we've seen a few approaches to writing ASTs, such as the AstBuilder. Those really were *just* examples though – now we'll look at the topic in a lot more detail.

9.6 AST by Example: Creating ASTs

This section examines creating ASTs in more depth, first by building an AST manually, and then using the three different approaches offered by AstBuilder. It's hard to make a general comparison between the options... each approach has advantages and disadvantages and should be used in different scenarios. The next four examples all produce the same AST; a return statement that returns a new instance of java.util.Date. This is the same thing as the source code "return new Date()", except that it is the AST and not actual source. The examples start at the lowest level possible, working directly with ASTNode instances, and ascend towards a higher level, where you can write code that is automatically converted into an AST. Let's see it in action.

9.6.1 By Hand

The most basic approach is to directly manipulate and construct the concrete classes. The main disadvantages are verbosity, complexity, and a lack of abstraction. As you can see in listing 9.37, the code to produce just a return statement can become quite large.

Listing 9.37: Creating AST objects by hand.

```
import org.codehaus.groovy.ast.*
import org.codehaus.groovy.ast.stmt.*
import org.codehaus.groovy.ast.expr.*

new ReturnStatement(
       new ConstructorCallExpression(
           ClassHelper.make(Date),
           ArgumentListExpression.EMPTY_ARGUMENTS
       )
)
```

For simple problems that approach suffices. An IDE gives you code completion and some type checking, and should allow you to navigate to the source code of the ASTNode class hierarchy, which helps a lot with the learning process. Also, there are no limitations on the AST you can produce. Any tree whatsoever can be created by directly using the classes like this - something which isn't the case for some of the other approaches. Also, it is quite easy to merge in information from the calling context. For instance, if you wanted to write a string value you know at compile time (as in with our CompiledAtASTTransformation example), then it's just a matter of passing it to the correct constructor as an argument.

There are some serious disadvantages to this approach, and for larger, more real-world examples this technique becomes a burden. First, the code to create AST quickly becomes large and doesn't really resemble the source code it's trying to model. For big examples it is difficult to read the source code you write and mentally map it into the code it is meant to produce. Second, you need to manage things like VariableScope and tying the nodes together yourself. For example, many tasks involve two steps, such as creating a MethodNode and then adding it to the parent class. To be effective you'll need to learn a large part of the API. Lastly, this approach offers no abstraction layer over the raw AST.

The lack of abstraction can be seen in the example. For example, a ConstructorCallExpression accepts a ClassNode and an Expression as arguments (they are used as the constructor type and arguments). To write this AST by hand you need to know that an empty argument list is ArgumentListExpression.EMPTY_ARGUMENTS and not null. Also, you need to know that the best type to use for the constructor arguments is an ArgumentListExpression object. You *can* use other types but they probably aren't what you intend. Lastly and most importantly, a ClassNode should be made by calling the ClassHelper.make() method.

CREATE CLASSNODES WITH CLASSHELPER

The `ClassHelper` class contains logic for creating and caching `ClassNodes` correctly, which is not exactly a simple process. If you need a ClassNode object then always create it through ClassHelper passing either a Class reference or a String representing a fully qualified class name. The String parameter is useful when you do not want a compile time dependency on the target class.

These are all implementation details of the ASTNode classes. A good abstraction should allow you to create ASTNode types without knowing all of this low level information. Luckily, Groovy provides the AstBuilder.

9.6.2 AstBuilder.buildFromSpec

You've already seen the buildFromSpec option in a previous section. This approach provides a lighter DSL over the ASTNode class hierarchy, and you can see it is slightly cleaner than the By Hand example of producing "return new Date()" (listing 9.38).

Listing 9.38: Creating AST objects using buildFromSpec

```
import org.codehaus.groovy.ast.builder.AstBuilder

def ast = new AstBuilder().buildFromSpec {
   returnStatement {
       constructorCall(Date) {
           argumentList {}
       }
   }
}

assert ast[0] instanceof ReturnStatement
```

The AstBuilder is a convenient shortcut for writing shorter, more concise AST. A shorthand notation exists for every ASTNode type, and much of the API is simplified. For instance, you can work directly with Class objects instead of ClassNode objects, and scopes are largely handled for you. And similar to the “by hand” approach, there are no limitations on the AST you create and it is easy to merge in code and parameters from the surrounding context. The best documentation for buildFromSpec is the unit test, which shows the correct usage of every single node type.

AstBuilder.buildFromSpec helps eliminate verbosity and some complexity. It *almost* matches the flexibility of calling the constructors by hand, suffering only from the fact that passing and referencing Class literals means that class must be present at compile time (a limitation the manual approach does not share). The buildFromSpec API currently does not allow you to use ClassHelper in all cases, class literals are sometimes required. However, the main disadvantage is that the DSL offers little in terms of abstraction. To effectively write new AST you'll still need to know a lot about what AST you want to produce. You don't need to worry about scopes, but you will need to know that a ReturnStatement requires an Expression and a ConstructorCallExpression requires an ArgumentListExpression. The next two alternatives offer better abstraction but with some loss in flexibility.

9.6.3 AstBuilder.buildFromString

The AstBuilder object has a buildFromString method that converts Groovy source code into the corresponding AST. By default it compiles the code to the Class Generation phase and returns only the AST for the enclosed script, not any classes defined within the script. Of course, both of these behaviors can be changed by passing different arguments to the method. This approach allows you to create an AST without knowing anything about the underlying object hierarchy: at this point we have a genuine abstraction over the AST classes. Listing 9.39 shows the buildFromString approach in action.

Listing 9.39: Creating AST objects using buildFromString

```
def ast = new AstBuilder().buildFromString('new Date()')
assert ast[0] instanceof BlockStatement
assert ast[0].statements[0] instanceof ReturnStatement
```

The only knowledge required is that a script is a BlockStatement and that BlockStatement has a ReturnStatement in its list. As you can see, this is a terse mechanism for AST creation, and the intent of the produced code is clear. This is the preferred approach when accepting and compiling user input, because it can usually be converted into a String.

The main limitation is flexibility. How exactly do you merge in code or variables from the calling context? How would we implement the getCompiledTime method? We would need to resort to String concatenation, and for more advanced examples it might be too difficult to manage. Also, what happens when you need access to the ASTNode implementation such as VariableScope objects? This is an abstraction over ASTNodes that doesn't easily allow you to dive deeper into the code when the need arises. Finally, synthesizing some types of structural nodes is difficult. Listing 9.40 shows how hard it is to generate a free-standing method, such as the getCompiledTime() method from the previous section.

Can we add a frown face or something to this example?

Listing 9.40: Trying to mix dynamic code with buildFromString

```
import org.codehaus.groovy.ast.builder.AstBuilder
import org.codehaus.groovy.control.CompilePhase
import org.codehaus.groovy.ast.*

def ast = new AstBuilder().buildFromString(
    CompilePhase.CLASS_GENERATION,
    false,
    ' static String getCompiledTime() { "' + new Date().toString() + '" } '
)

assert ast[1] instanceof ClassNode
def method = ast[1].methods.find { it.name == 'getCompiledTime' }
assert method instanceof MethodNode
```

That's a little complicated! You can use the buildFromString method for this type of task, but it's fraught with difficulties. Pushing compile time data, such as the current date, into the AST requires String concatenation and escaping, and getting the MethodNode requires searching through all the methods defined on the class and pulling it out by name. The buildFromString method and the next approach are great for creating method bodies, expressions, or statements. But if you're dealing with structure like ClassNodes, MethodNodes, or FieldNodes, then it is easier to use buildFromSpec or create the nodes by hand.

9.6.4 AstBuilder.buildFromCode

The last approach is possibly the most interesting. Using the buildFromCode method you can specify your source code directly as source code, and the builder turns it into AST. This is similar to the buildFromString approach exception that the input is not a String, it is just code.

Listing 9.41: Creating AST objects using buildFromCode

```
def ast = new AstBuilder().buildFromCode {
   new Date()
}
assert ast[0].statements[0] instanceof ReturnStatement
```

This is quite simple, and it reads like code because it *is* code! The advantage is that the Groovy compiler and IDEs will syntax highlight correctly, do code completion, and generally validate your input. But its strength is also its weakness. The main disadvantage is that the Groovy compiler will validate your input. For instance, you cannot declare a new class within a closure body, so declaring a new class (or method) using buildFromCode is not allowed. Also, there is no way to bind in data from the enclosing context. The "new Date()" expression here is only executed at runtime. The scope at compile time is different from the scope at runtime, so any variables in scope at compile time will not be available when the code is executed. There is no way to write the "getCompiledTime()" method using this approach. When it's appropriate, this is the most elegant solution, and offers the best abstraction level – but there's a price to pay in flexibility.

There are many different scenarios where ASTs can be useful, and there are several APIs to help you build them. There is no one right way to create an AST. In general, my advice is the same for most things in life: start simple, stay simple. If you think the AstBuilder simplifies your implementation, then by all means use it. But if you find yourself fighting against it, or spending too much time figuring out how it works, then just go the simple route and write the AST by hand. There is a lot to learn with compile-time metaprogramming, and your time is probably better spent writing a few more tests than trimming down your AST generation by a few more lines of code. That brings us to our next topic: testing.

9.7 Testing AST Transformations

Test, test, test. It is hard to over-test an AST transformation because source code can come in a dizzying array of variations, each exposing unique edge cases. Also, during upgrades between Groovy versions you need a good regression test. Luckily, transformations are easy to test. Local transformations are the most testable. Typically, you create an inner class within your test case that contains your annotation and then test against that class. Consider the test for @WithReadLock in listing 9.48.

Listing 9.48: Possible unit test for the @WithReadLock transformation.

```
class ReadWriteLockTest extends GroovyTestCase

   private static class MyClass {
       @groovy.transform.WithReadLock
       public void readerMethod1() { }
   }

   public void testLockFieldDefaultsForReadLock() {
       def field = MyClass.getDeclaredField('$reentrantlock')
       assert Modifier.isPrivate(field.modifiers)
       assert !Modifier.isTransient(field.modifiers)
       assert Modifier.isFinal(field.modifiers)
       assert !Modifier.isStatic(field.modifiers)
       assert field.type == ReentrantReadWriteLock
   }
}
```

This test case asserts that WithReadLock on a method creates a private final instance field called "$reentrantlock" with type ReentrantReadWriteLock. This is a simple and readable approach, and it works especially well within an IDE where the class can be verified and easily seen in searches. But there are two disadvantages. One, with a lot of tests come a lot of inner classes, and this can clutter up your namespace. Your build will create many, many extra classes that don't have meaning outside of a specific test method. And two, there is no way to debug the AST transformation in an IDE because by the time the test runs the class is already compiled. To work around these issues you can use a GroovyClassLoader to compile the class. The same test is presented in listing 9.49 but this time using a GroovyClassLoader.

Listing 9.49: Improved unit test for the @WithReadLock transformation.

```
class ReadWriteLockTest extends GroovyTestCase {

   public void testLockFieldDefaultsForReadLock() {
        def tester = new GroovyClassLoader().parseClass('''
            class MyClass {
                @groovy.transform.WithReadLock
                public void readerMethod1() { }
            }
        ''')

        def field = tester.getDeclaredField('$reentrantlock')
        assert Modifier.isPrivate(field.modifiers)
        assert !Modifier.isTransient(field.modifiers)
        assert Modifier.isFinal(field.modifiers)
        assert !Modifier.isStatic(field.modifiers)
        assert field.type == ReentrantReadWriteLock
   }
}
```

With this approach debugger breakpoints should be hit when compiling MyClass, making development and troubleshooting much easier. Also, all class definitions are local to the test method, so the test is not polluted with dozens of private classes and there are never naming conflicts. However, the class definition within a String makes it more difficult to find usages of your annotation. If you need to create instances or run a script as setup, you may want to use GroovyShell instead of GroovyClassLoader, as shown in listing 9.50.

Listing 9.50: An AST transformation unit test based on GroovyShell.

```
class ReadWriteLockTest extends GroovyTestCase {

    public void testLockFieldDefaultsForReadLock() {
        def tester = new GroovyShell().evaluate('''
            class MyClass {
                @groovy.transform.WithReadLock
                public void readerMethod1() { }
            }
            new MyClass()
        ''')

        def field = tester.getClass().getDeclaredField('$reentrantlock')
        assert Modifier.isPrivate(field.modifiers)
        // and more assertions...
   }
}
```

Notice how this script returns a new instance of MyClass. GroovyClassLoader and GroovyShell are similar, and which you use is largely a matter of preference. One tip: try to leave your assertions out of the String script. The more you can leave out of the String the better, because tools will have a much easier time understanding and supporting your code.

Global transformations are a little harder to test because they must generally be packaged and on the classpath before your test is compiled. To make testing global transformations easier, Groovy contains a class created specifically for testing called TransformTestHelper. You configure the object with a transformation and a compiler phase in which the transform should run, and then ask it to compile a File or String into a class you can test against. Listing 9.51 shows an example of TransformTestHelper.

Listing 9.51: Using TransformTestHelper for testing transformations.

```
def transform = new CompiledAtASTTransformation()
def phase = CompilePhase.CONVERSION
def helper = new TransformTestHelper(transform, phase)
def clazz = helper.parse(' class MyClass {} ' )
assert clazz.getCompiledTime() != null
```

Between GroovyShell, GroovyClassLoader, and TransformTestHelper, there are quite a few options for testing. The hard part of testing is not overall test coverage, but covering the edge cases. For instance, do your tests cover code that is written as a script *and* code that is written as a class? How about a mixture of both? Does it cover inner classes and anonymous classes? Have you considered what happens with internal naming conflicts? How does it run when other transformations are present? Properly testing transformations is a fun challenge. There are many opportunities to learn from your own experience but also the experience of others. The Groovy source code contains many unit tests for transformations. Doing a little code archeology now is time well spent, especially if it avoids a future late-night support call.

9.8 Limitations

Congratulations, you have almost finished your training in compile-time metaprogramming. Consider yourself armed and dangerous, both to others and yourself. It may be tempting to write a new language feature, but be careful. There are limitations and drawbacks to mucking about with the Groovy compiler. This section contains the bare minimum set of limitations you should know before embarking on your journey.

It's Early Binding

Groovy's power comes from late binding. Methods can be added to classes at runtime. Method overloading is resolved at runtime. Missing method exceptions can be caught and handled at runtime. In contrast, all the AST transformation work occurs at compile time, making it less flexible than dynamic metaprogramming. It can be very useful, but in general runs against the spirit of Groovy. If you can find a runtime solution then use it. The best answer to the question "When should I use compile-time metaprogramming?" is "only when you have to".

It's Fragile

The syntax of Groovy and the GDK classes (anything in the groovy.* package) all form a public contract that is guaranteed to be backwards compatible between releases. You may have noticed from the import statements in the code examples that most the AST related code is in org.codehaus.groovy packages. The backwards compatibility promises are weaker here. As new features are added to the language, there may be instances where breaking changes are introduced to the AST node hierarchy. For instance, Groovy 1.7 contained a class called RegExExpression that is entirely missing in the 1.8 branch. Some open source projects (notably Spock) are complex enough to require a different build for each version of Groovy they support.

It Adds Complexity

When you use compile-time metaprogramming you are basically adding a feature to the language. If you add too many features, your users will drown in complexity. If you provide too many similar features, users be confused about the best way to use an object. Language designers talk about orthogonality and composition: features should be independent of one another and be able to be used together without conflicts. Complexity lies at the intersection of overlapping language features. Consider how Java generics, autoboxing, and primitive types intersect. There are many edge cases where unboxing a Boolean into a boolean throws a NullPointerException. Or a List can hold all objects except primitives. When several features come together edge cases occur, and sharp edges are dangerous. Be sparing in your cleverness.

Its Syntax is Fixed

AST transformations can change the *meaning*, or semantics, of code. For instance, Spock repurposes the logical OR operator (|) and the break/continue label to have a special meaning in test specifications. However, Spock does not introduce any new syntax. The syntax of Groovy is fixed by the parser. Invalid Groovy will not parse, and AST transformations will not be invoked for it. You can change the semantics of the language, but you cannot change the syntax… except that you actually can if you are determined enough. Under the covers, Groovy uses ANTLR as a parser, and it’s possible to write an ANTLR plugin for Groovy. That's a topic that deserves its own book, but information about how to do it can be found online.

It's Not Typed

Most interesting AST transformations rely on knowing the type of a variable. To Groovy, almost everything is an Object. It is surprisingly difficult to determine the type of an instance and impossible to determine at compile time exactly where a method call will dispatch. For instance, metaClass additions are rarely known at compile time yet effect method dispatch. You can try to keep track of this information yourself, but as soon as a closure is declared, or a second thread is run, then the variable may no longer be what you think it is. This is acceptable for some tools like IDE integration or static analysis that read and make suggestions based on AST. But if you're rewriting AST and generating new bytecode then guessing the type of an instance and getting it wrong can have disastrous effects on a program, especially if it fails to compile because of your mistake. Be careful with what you think you know about types. It's easy to make a guess, and it's easy to be wrong.

It's Unhygienic

It is possible, using an AST transformation, to introduce a field, class, method, or variable that conflicts with an existing one. For instance, if you add a method called getCompiledTime(), you need to consider the possibility that the target class already has that method. The term for a compile-time metaprogramming system that allows naming conflicts is "unhygienic". It's not really a term of derision, but it is obvious that the term was coined by users with a hygienic language. It's not an insurmountable problem, and you should carefully select names for synthetic variables and private fields and methods. The $ symbol is typically used in the identifier name because this symbol is rarely used in user-written code. For example, @WithReadLock generates a field called "$reentrantlock". You can still have a conflict, but it should be rare. Choose your names carefully.

And with that the compile-time metaprogramming training is complete. It's time to go out into the world and write some interesting code.

9.9 Next Steps

If you have an idea you want to implement, then the next steps are fairly obvious. Take a look at the templates in the Groovy source distribution, set up a project, and write some code. But before going too far I recommend talking about the idea on the Groovy mailing list. The community takes an active role helping people refine their ideas and make decisions about implementation. You may find you're talked into using runtime meta-programming instead.

If you don't have a specific idea but want to learn more, then writing a static analysis rule for CodeNarc is an excellent way to get started. It is a small and well-contained project, the community is friendly, and you should be able to make a contribution within an hour or two. There is a "create-rule" script that comes with the project that creates all the needed files and unit test templates for new rules. CodeNarc is based on AST visitors, and the various types of visitors are fully described in Appendix G.

For bookworms who prefer reading, there are other resources available. One of the authors keeps an active blog that includes several articles related to Groovy and compile-time metaprogramming. But other languages are worth investigating as well. Java, C#, Clojure, and JRuby also expose their ASTs to programmers or allow you to work directly with expression trees, and their documentation is easily found with a search engine. Of the languages listed, Clojure has the best support with a feature called Macros, which is arguably a more advanced and powerful form of the AST transformations described in this chapter. For an overview of language implementation in general, I highly recommend "Language Implementation Patterns" by Terence Parr. Finally, for an overview and classic reference for compile-time metaprogramming in Lisp, read Paul Graham's "On Lisp" which is freely available for download on the Internet.

9.10 Summary

Compile-time metaprogramming is a new, exciting, and growing area of the Groovy ecosystem. Many of the new Groovy features use compile-time metaprogramming to eliminate redundant or verbose code. Use annotations like @Canonical, @Lazy, and @InheritConstructors to remove this unnecessary code from your source files yet still have it visible from Java. Use @Delegate, @Immutable, and @Singleton for an easy path to correct object design. The annotations @Log, @Commons, @Log4j, and @Slf4j streamline declaring and using loggers. Declarative concurrency constructs like @Synchronized clean up multithreaded code considerably, @AutoClone and @AutoExternalize help make externalizing and cloning simple, and scripting has become much safer with @TimedInterrupt, @ThreadInterrupt, and @ConditionalInterrupt.

Writing your own transformation involves either a local AST transformation, which manipulates the AST when a certain annotation is discovered, or a global AST transformation, which is run for every compiled class. If your transformation needs to know information about the tree, then you'll probably need a code visitor to walk the entire abstract syntax tree, visiting nodes as they are found in the source code. If neither a local or global transformation is suitable for your scenario then you can always weave a visitor directly into a classloader without too much effort.

Groovy contains several alternatives for generating ASTs. Writing them by hand using the class constructors is always an option, but it's worth learning the AstBuilder API as well. The buildFromSpec method offers a useful DSL over the constructors; buildFromString offers a useful abstraction over all the classes, and buildFromCode provides an intuitive and elegant way to convert source into an AST.

There are several standard tools for working with ASTs. The javap application displays the raw .class file output. Groovy Console's AST Browser is much more advanced, and shows the AST in tree form and also generated source form. A good Java decompiler is always a useful view of transformation results too, but perhaps the best tool is a large suite of unit tests. GroovyClassLoader, GroovyShell, and TransformTestHelper can all be used to test drive (or regression test) AST transformations.

Compile-time metaprogramming is not suitable for every scenario. In general, prefer late binding and flexibility. But there are concrete advantages: Java integration, easy embedded language development, delayed evaluation, and the ability to change the semantics of the language. AST transformations are opening a whole new world of what's possible with Groovy. Hopefully, after reading this chapter you are better prepared to go into that world and see what new and useful tools you can create.

17
Concurrent Groovy with GPars

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

Edsger Dijkstra, How do we tell truths that might hurt?

This chapter covers

- How Groovy makes concurrency more approachable
- Parallel collections, fork/join, map/filter/reduce, dataflow, actors, and safe agents
- Putting these concepts to work with the GPars library

We're going to start our exploration with some general considerations about concurrency followed by moving from the simple to the more advanced usages of concurrency. Along the way, we'll visit waypoints that show various means of coordinating concurrent tasks: from predefined coordination to implicit and explicit control. We'll move on to investigate how to safeguard objects in a concurrent environment and wrap the topic up with a final showcase. But let's start by considering why we might want to enter this challenging landscape in the first place.

Public wisdom has it that we will no longer see the major speed-ups in processor cycle times that we all got so used to. In the past, the safest way to improve software performance was to wait for 18 months, get a new computer, and enjoy the doubled speed.

These days, it's more likely that you'll see a slight decrease in processor speed but with the benefit of having twice as many processing units (cores). Our programs must now be prepared to take advantage of the new direction of hardware evolution.

This *could* mean putting the burden of managing concurrency on the application programmer. But considering the huge amount of difficulties that come with classical approaches to concurrency this doesn't seem like a wise choice.

An alternative approach is to put the burden on framework designers so that we can run our code in a managed environment that handles concurrency for us. The Java Servlet framework may serve as an example: the Servlet programmer – and this includes all Servlet-based technologies like JSP, GSP, JSF, Wicket, and else - doesn't care much about concurrency but the webserver executes the application for many requests in parallel. The programmer only has to obey some restrictions like not spawning threads on his own and only sharing mutable state in dedicated scopes. Admittedly, some projects break these restrictions since they are not technically enforced, but by and large this has been a very successful model.

The concurrency concepts we will look at in this chapter follow the successful Servlet approach, in that they introduce an elevated level of abstraction. This allows the application programmer to focus on the task at hand and leave the low-level concurrency details to the framework.

17.1 Concurrency for the rest of us

Your job as an application programmer is to get the sequential parts of your code right, including their test cases. When concurrency is required, you can choose one of the tools explained in this chapter, passing it you sequential code for execution. Understanding the concepts is a prerequisite for choosing the most appropriate one for the situation. You don't have to understand the inner workings of each implementation but you need to understand its approach and constraints.

17.1.1 Concurrent != parallel

A full exposition of concurrency is beyond the scope of this chapter, given that there are whole books devoted to the topic. Also, it is not our job to explain the concurrency support provided by the Java language and the `java.util.concurrent` package in the Java standard library. We'll approach the topic from a Groovy point of view and assume that you are at least somewhat familiar with the Java basics. The Groovy view starts with the observation that concurrency is more than parallelism.

NOTE

Concurrency allows better utilization of resources, higher throughput, and faster response times – but the real value is in the coherence of the programming model. Each concurrent task fulfils one single coherent purpose. Multiple such tasks may run sequentially, in intermixed time slices or in parallel.

Let's start with resource utilization. The obvious resource that you want to use efficiently is your processing capacity: spreading calculations over many processing cores to get the results faster. Note that this only makes sense if those cores would be otherwise idle! With a dual-core machine you are often better off leaving the second core to the operating system to run its other processes. Prominent examples of "other processes" are your database and web server.

Spreading computation over many cores, processors, or even remote machines is what we call parallelism. Concurrency goes beyond parallelism, though. It allows asynchronous access to the database, file system, external devices, the network, and foreign processes in general, no matter whether they are managed by the operating system or other applications. If you are into service-oriented architectures (SOA), you can think of all of these resources as services that are typically slow. If we were to work in a synchronous fashion – waiting for each service to complete before progressing to the next step – we would not exploit other resources to their maximum, especially not our processing capacity.

There is one special service that is particularly slow but has very low tolerance for latency: the user. His input is notoriously slow but as soon as he submits it, he expects a response immediately. A responsive user interface may be the best example of concurrency. Even on a single-core machine, the user legitimately expects that he can move the mouse, enter text, click a button, and so on while the application is fetching web pages or sending them to a printer. This may well make the overall task marginally *slower* as the processor spends time switching context between background threads and the user interface – but the experience is a much more pleasant one for the user.

All this may sound as if asynchronous resource consumption is the only goal of concurrency. It is the most obvious one but certainly not the only one, and possibly not even the most important one. At its heart, concurrency is a great enabler for a coherent programming model.

Imagine writing a graphical application from scratch. You wouldn't want to intermix your application code with checking every tenth of a second whether the mouse has moved and the cursor on the UI needs repainting. Nor would you want to repeatedly check for garbage collection from within your application. Luckily, Java comes with a concurrent solution that takes care of updating the UI and running the garbage collector. The main point here is that this allows each piece of the system – your application code, the UI painter, and the garbage collector – to focus on its own responsibility while remaining blissfully unaware of the others.

CONCURRENCY FOR SIMPLER CODE

Concurrency enables us to write simple, small, coherent actions that implement exactly one task. Simple actions like these are easier to test, easier to maintain, and easier to implement in the first place.

These benefits do not come for free. There is controlling effort for starting and stopping each task, mutually exclusive assignment of resources (scheduling), safeguarding shared resources, and coordination of control when, for example, one task consumes what a second task has produced.

Far too many developers are obsessed with performance improvements, overlooking the other benefits that a well-designed concurrent programming model yields.

JAVA'S BUILT-IN CAPABILITIES

Java has supported concurrency at the language and library level right from the first version. Starting a new `Thread` and waiting for its completion is simple. Groovy sprinkles a little sugar on top with the GDK so that you can start a new `Thread` more easily by using the `start` method with a closure argument.

```
def thread = Thread.start { println "I'm in a new thread" }
thread.join()
```

The introduction of the `java.util.concurrent` package brought many improvements, including thread pools, the executor framework, and many datatypes with support for concurrent access. If you haven't looked at this package yet, now is the time to do so. You will find excellent tutorials on the web as well as very good books like *Java Concurrency in Practice* (B. Götz et al., 2006) and *Concurrent Programming in Java* (D. Lea, 1999).

Reading these books can also be a scary experience, though. The authors walk through various examples of seemingly simple code and explain how it fails when called concurrently. I guess this is the reason why many developers shy away from concurrency. They don't want to appear incompetent and leave those fields to the experts that can manage this black art. Well, we have to overcome this fear somehow, and the concepts introduced in this chapter are targeted at giving you an enjoyable pathway into concurrency.

The first notable difference is that we are rarely going to use the concept of a thread. Instead, we'll think in terms of *tasks*. A task is a piece of sequential code that may run concurrently with other tasks. This may involve thread management and pooling under the covers but you don't have to care.

We will free you from dealing with Java language features such as `volatile` and `synchronized`. They require some advanced knowledge of the Java memory and threading model and are all too easy to get wrong. Likewise, there is no more need for wait/notify constructions for thread coordination, which is an infamous source of errors. Since we don't expose threads, we can offer less error-prone task coordination mechanics.

17.1.2 Introducing new concepts

In order to make concurrent programming easier, we will introduce concepts that are new in the sense that they are not yet widely known, even though most of them were developed a long time ago and have implementations in other languages as well. They cover three main areas:

- Starting and stopping concurrent tasks
- Coordinating concurrent tasks
- Controlling access to shared mutual state

Parallel collections with *fork/join* and *map/filter/reduce* are concepts that hide the work of starting and stopping concurrent tasks from the programmer and coordinate these tasks in a predefined manner.

Actors create a frame in that tasks can run without interference but they start, stop, and coordinate explicitly.

Dataflow variables, operators, and streams coordinate concurrent tasks implicitly such that downstream data consumers automatically wait for data providers.

If your tasks need to access shared mutual state, you can delegate the coordination of concurrent state changes to an *agent*.

As you will see, we will use a lot of Groovy features to make the above possible, particularly closures, meta-programming, and AST transformations. The real heavy lifting is done by the implementation in the GPars library.

USING GPARS

At the time of writing, GPars is still an external library. By the time you read this, it may have become part of the Groovy standard library. The package structure is already prepared for that move so that you can run the code presented here without modification.

The simplest way to use the GPars library is to just prefix your code with

```
@Grab('org.codehaus.gpars:gpars:0.10')
<some import statement here>
```

This statement will transparently download and cache the specified version of the library (0.10 as of now) and its dependencies. If you would rather like to add GPars as a dependency to your Gradle or Maven build or download its jars manually, please refer to http://gpars.codehaus.org, which is also the place to find additional information including many demos and the comprehensive documentation.

Now we've set the stage, let's visit a very common application of concurrency: processing all the items in a collection concurrently.

17.2 Concurrent collection processing

Processing collections is particularly auspicious when each item in the collection can be processed independently. This situation also lends itself naturally into processing the items concurrently.

Groovy's object iteration methods (each, collect, find, and else) all take a closure argument that is responsible for processing a single item. Let's call such closures *tasks*. Naturally, GPars builds on this concept with the capability to process these tasks concurrently in a fork/join manner.

FOR CLARIFICATION

In this chapter, the term "fork/join" always indicates that several items are each processed in their own "forked" task and all tasks are immediately "joined" after execution. Please be aware that there the same term may have different meanings in other contexts.

Listing 17.1 uses the fork/join approach to concurrently calculate the squares of a given list of numbers by using the `collectParallel` method that the `withPool` method adds through meta-programming to a list of numbers. This method works exactly like Groovy's collect method, beside that we collect concurrently now.

Listing 17.1 Calculating a list of squares concurrently

```
@Grab('org.codehaus.gpars:gpars:0.10')
import static groovyx.gpars.GParsPool.withPool

def numbers = [1, 2, 3,  4,  5,  6]
def squares = [1, 4, 9, 16, 25, 36]

withPool {
   assert squares == numbers.collectParallel  { it * it }
}
```

The concurrency is almost invisible: there is no thread creation, no thread control, and no synchronization on the resulting list visible in the code. This is all safely handled under the covers.

DISCLAIMER

Calculating squares concurrently is only an introductory example for educational purposes. In practice, the overhead of concurrency only makes sense if the tasks can be split up in reasonably sized, time consuming chunks.

You may wonder how many threads listing 17.1 uses for calculating the squares. The short answer is: you shouldn't care. The longer one is: GPars uses a default that is calculated from the number of available cores plus one. That makes three for a dual-core machine. Alternatively, you can explicitly supply the number of threads to use as the first argument to the `withPool` method:

```
withPool(10) {
   // do something with a thread pool of size 10
}
```

`GParsPool` does not create threads. Instead, it takes them from a fork/join thread pool of the jsr166y library, which is a candidate for inclusion in future Java versions. GPars uses this library extensively, especially its support for parallel arrays that are the basis for all parallel collection processing in GPars.

17.2.1 Transparently concurrent collections

Having the `*Parallel` counterparts of the Groovy object iteration methods is nice and convenient. However, the method names are a bit lengthy and do not feel very groovy. Couldn't we just use the standard method names and give them a concurrent meaning?

Listing 17.2 makes the list of numbers transparently subject to concurrent treatment with a method name that `withPool` adds to collections and that is aptly named makeTransparent.

Listing 17.2 Calculating a list of squares with transparent concurrency

```
import static groovyx.gpars.GParsPool.withPool

def numbers = [1, 2, 3,  4,  5,  6]
def squares = [1, 4, 9, 16, 25, 36]

withPool{
    assertSquares(numbers.makeTransparent(), squares)
}
def assertSquares(numbers, squares) {
    assert squares == numbers.collect { it * it }
}
```

Note to self: makeTransparent() has been renamed.

Groovy meta-programming is again in action here. When called from within the `withPool` closure, the standard `collect` method is modified to delegate to the `collectParallel` method for collections that have been made transparent.

Note that the `assertSquares` method knows nothing about concurrency! In fact, when this method is called from outside the `withPool` closure, it will calculate the squares sequentially. When called from inside the `withPool` closure, the calculation runs concurrently.

IN OTHER WORDS

Transparently concurrent collections enable you to pass collections into methods that have been written for sequential execution and make them work concurrently for a specific caller. The caller can even decide about the "amount" of concurrency by passing the pool size argument to the `withPool` method.

Think how much easier this makes unit testing of methods like `assertSquares`. Of course, this approach has its limits. If we do something really silly, let's say side-effecting from inside our task, then our code may run fine sequentially but not when passed a transparently concurrent collection.

For example, the following code *does not* construct an ordered String of squares:

```
def assertSquares(numbers, squares) {
    String result = ''
    numbers.each { result += it * it }  // This is wrong, don't do it!!!
    assert squares.join('') == result
}
```

When called with `numbers.makeTransparent()` the above may work accidentally but at times, a higher number will be processed before a smaller number and the assertion will fail. Even worse, modifying a variable in this way is not a thread-safe operation! There are three separate operations involved: reading the current value from the variable, computing the new value, and then writing the new value to the variable. If these operations are interrupted by another task the results may be inconsistent, with one task overwriting the result of another. This is a special case of a race condition: a missing update.

Therefore, when you run the above code multiple times, you will see that the result string is often missing squares.

For the record, the correct and concurrency-friendly solution would be

```
def assertSquares(numbers, squares) {
    assert squares.join('') == numbers.collect{ it * it }.join('')
}
```

The good news is that you can easily avoid errors like the one above by simply sticking to the rule of avoiding state changes from inside the iteration methods.

Transparency has some interesting characteristics. First, it is *idempotent*. Calling `makeTransparent` on a collection that already is transparent returns the collection unmodified. Second, it is *transitive*. When you call a method like `collect` on a transparently concurrent collection, the returned list is again transparently concurrent such that you can chain calls. Listing 17.3 chains calls to `collect` and `grep` with the effect that `grep` is also called concurrently. The code first collects all squares and then filters the small ones.

Listing 17.3 Using transitive transparent concurrency to find squares < 10

```
import static groovyx.gpars.GParsPool.withPool

withPool {
    def numbers = [1, 2, 3,  4,  5,  6].makeTransparent()
    def squares = [1, 4, 9]
    assert squares == numbers.collect{ it * it }.grep{ it < 10 }
}
```

The `collect` and `grep` methods use the same fork/join thread pool. In fact, every concurrent collection method called from the same `withPool` closure will do so, regardless of whether they appear as transparent or *Parallel invocations.

The fork/join approach is probably the simplest step into concurrent programming but for the small squares problem, we could do better. Listing 17.3 first collects all squares, stores them in a list, and then processes the temporary list to filter the small squares. It would be more efficient to spare the temporary list and do the squaring and filtering in one task. We will revisit this approach in section 17.2.

17.2.2 Available fork/join methods

The full list of available concurrent methods is in class `groovyx.gpars.GParsPoolUtil`. The transparent methods are in `groovyx.gpars.TransparentParallel`. Table 17.1 puts the two versions next to each other.

Table 17.1 Concurrency-aware methods in "withPool"

Transparent	Transitive?	Parallel
`any { ... }`		`anyParallel { ... }`
`collect { ... }`	yes	`collectParallel { ... }`
`count(filter)`		`countParallel(filter)`
`each { ... }`		`eachParallel { ... }`
`eachWithIndex{ ... }`		`eachWithIndexParallel { ... }`
`every { ... }`		`everyParallel { ... }`
`find { ... }`		`findParallel { ... }`
`findAll { ... }`	yes	`findAllParallel { ... }`
`findAny { ... }`		`findAnyParallel { ... }`
`fold { ... }`		`foldParallel { ... }`
`fold(seed) { ... }`		`foldParallel(seed){ ... }`

`grep(filter)`	yes	`grepParallel(filter)`
`groupBy { ... }`		`groupByParallel { ... }`
`max { ... }`		`maxParallel { ... }`
`max()`		`maxParallel()`
`min { ... }`		`minParallel { ... }`
`min()`		`minParallel()`
`split { ... }`	yes	`splitParallel { ... }`
`sum()`		`sumParallel()`

Contrasting table 17.1 with the Groovy object iteration methods shows a few notable differences that are due to the concurrent processing.

- In addition to `find`, there also is a `findAny`. While `find` always returns the first matching item in the order of its collection, `findAny` may return whatever matching item it finds first.
- The GDK `inject` method is replaced by `fold`. While `inject` runs through the collection in strict order, there is no such order in concurrent processing and thus the contract differs. The `fold` method acts like `inject` but you have to be aware that its task closure may be invoked with any combination of items and/or temporary results.
- Transparent concurrent methods are only transitive when they return a collection as their return type. Note that using the transparent `find` method on a list of lists also returns a collection but this will not be made transparent automatically.
- Not all Groovy object iteration methods have a concurrent counterpart. Some are simply missing at the time of writing, while others don't make sense in a concurrent context.

Finally, it is worth noting that this approach to concurrent processing is not restricted to collections but can be used with any Java or Groovy object - the Groovy object iteration logic applies.

We will now elaborate on this approach a bit further by investigating the map/filter/reduce concept.

17.3 Becoming more efficient with map/filter/reduce

We have seen concurrent tasks of calculating squares and filtering in listing 17.3 with the fork/join approach. First we had to collect all the squares; only then could we proceed with the filtering part. This isn't ideal: we don't really need the intermediate results as a collection.

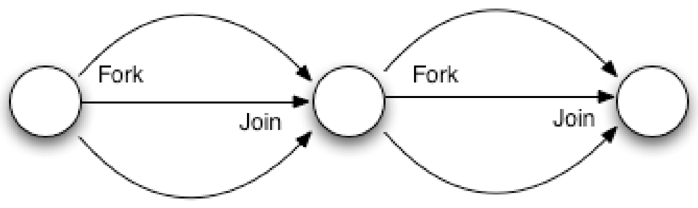
Fortunately, there's an alternative. The *map/filter/reduce* approach allows us to chain tasks in a way that doesn't restrict us to finish all the squaring before filtering. To make the difference even more obvious, listing 17.4 shows a map/filter/reduce performing a variant of the squaring problem. We've made two changes: incrementing the value before squaring it and adding the squares instead of filtering. What has been `collect` and `fold` in fork/join, becomes `map` and `reduce` for map/filter/reduce. The methods are used in a similar fashion but as we will see they work quite differently.

Listing 17.4 Using map/filter/reduce to increment each number in a list, square it, and add up the squares - all concurrently

```
import static groovyx.gpars.GParsPool.withPool

withPool {
    assert 55 == [0, 1, 2, 3, 4].parallel
        .map    { it + 1  }
        .map    { it ** 2 }
        .reduce { a, b -> a + b }
}
```

The `map` and `reduce` methods are available on parallel collections. We get such an instance by holding onto the `parallel` property of our list. This property is available inside the `withPool` closure.

Figure 17.1 depicts the difference in the workflow. Assume that time flows from left to right, bubbles denote states of execution, and arrows show scheduled tasks. If you imagine a sweeping vertical line, you can see which tasks can be executing at any point in time. While fork/join always has the same order, the map/filter/reduce example is only one of many possible execution orders. Its inner bubbles can freely flow horizontally like pearls on a string.



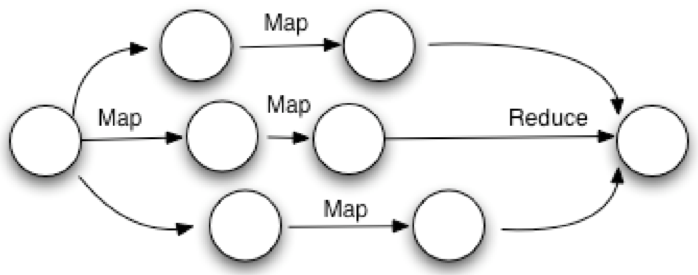


Figure 17.1 Contrasting task concurrency for fork/join vs. map/filter/reduce where map/filter/reduce can achieve a higher degree of concurrency.

In the map/filter/reduce example there are many valid execution orders. On one run all the increments may be calculated before all squares, effectively giving you fork/join workflow, but this would be a very unlikely coincidence.

Equally, on another run we could end up with one increment and its square being calculated, then a second one, and then both being passed into the reduce task even before the third increment starts!

Either way, GPars makes sure that all the increments, squares, and their sum are calculated correctly in the end. But the many different possible workflows open more possibilities for different tasks running concurrently. The task coordination is still predefined even though the coordination scheme spans over more tasks and allows for more variability in scheduling.

With fork/join, a collect task could only run concurrently with other invocations of that collect task. With map/filter/reduce, any task can run concurrently with any other one, thus providing a higher degree of concurrency.

FOR THE GEEKS: THE MERITS OF "MORE CONCURRENCY"

If the scheduler has more options for assigning a task to a thread, there is a lower probability that a few slow task invocations thwart the overall execution. With more options in the workflow, map/filter/reduce offers more concurrency over fork/join.

We have seen that map/filter/reduce works on a parallel abstraction that comes with the concurrency-aware methods listed in table 17.2. Note that only map and filter return a parallel datatype that allows further map/filter/reduce processing.

Table 17.2 Concurrency-aware methods for map/filter/reduce

Method	Chainable	Analogous to
`filter { ... }`	True	`findAll`

`getCollection()`		
`map { ... }`	True	`collect`
`max { ... }`		
`max()`		
`min { ... }`		
`min()`		
`reduce { ... }`		`inject, fold`
`reduce(seed)  { ... }`		`inject, fold`
`size()`		
`sum()`		

This gives us enough knowledge to finally present the small squares problem with map/filter/reduce in Listing 17.5.

We make use of the `filter` method that only passes temporary results down the execution stream if they satisfy the given closure. This is analogous to the `findAll` method for sequential code. The `filter` method is such an important part of the concept that we have included it in the name. This also distinguishes it from the more commonly known "map/reduce" label that is also used in very different[1] contexts.

For the assertion in listing 17.5, we need to refer to the `collection` property to unwrap our parallel datatype and make it comparable to the list of expected numbers.

Listing 17.5 Collecting the small squares with map/filter/reduce

```
import static groovyx.gpars.Parallelizer.GParsPool.withPool

withPool {
    def numbers = [1, 2, 3,  4,  5,  6]
    assert [1, 4, 9] == numbers.parallel
        .map    { it * it }
        .filter { it < 10 }
        .collection
}
```

So far, fork/join and map/filter/reduce have proven to be concurrency concepts that are fairly easy to use. This is mostly due to their baked-in, predefined task coordination that implements a well-known flow of data. When one task needs to wait for data from a preceding one, this is all known in advance and handled transparently. There's no room for errors to creep in.

In the next section, we will investigate how to coordinate tasks when we need more flexibility in the flow of data.

17.4 DataFlow for implicit task coordination

Both fork/join and map/filter/reduce work on collection data types that are transformed and processed. That makes their data flow predictable and allows for an efficient implementation.

In the more general case, we may need to derive a value from data that delivered by concurrent tasks. For this to work, we need to ensure that all the affected tasks are scheduled in a sequence that allows data to flow from assignment to usage. This may sound difficult, but with the DataFlow concept it's a snap.

Listing 17.6 demonstrates a simple sum where the input data isn't known at the time where we declare the logic of the task. Therefore, each reference is wrapped with a dataflow. Assignments to dataflow references happen in concurrent tasks.

[1] compare http://en.wikipedia.org/wiki/MapReduce

Listing 17.6 A basic DataFlow adds numbers that are assigned in concurrent tasks

```
import groovyx.gpars.dataflow.DataFlows
import static groovyx.gpars.dataflow.DataFlow.task

def flow = new DataFlows()
task { flow.result = flow.x + flow.y }     //#1
task { flow.x = 10 }                       //|#2
task { flow.y =  5 }                       //|#2
assert 15 == flow.result                   //#3
```

#1 Assign derived value
#2 Assign value
#3 Read value

We start with the calculation in #1 where a data flow variable `result` is derived from data flow variables `x` and `y`, even though `x` and `y` are not yet assigned. This calculation happens in a new task that is started by the `task` factory method. It has to wait until `x` and `y` have been assigned values.

Assignments to `x` and `y` in two other concurrent tasks #2 make these values available so that #1 can execute.

The main thread waits at #3 until `result` can be read. This means that #1 has to finish, which can only happen after both the tasks in #2 have finished. The data flow from #2 to #1 to #3 happens regardless of which task is started first. This is implicit thread coordination in action.

17.4.1 Reproducible deadlocks

Predefined coordination schemes like fork/join and map/filter/reduce are deadlock-free. It is guaranteed that the task coordination itself never produces a deadlock - the situation when concurrent tasks block each other in a way that prohibits any further progress.

Don't get me wrong: it is still possible to write code that uses fork/join mechanics and runs into a deadlock anyway. However, this wouldn’t be the result of the coordination scheme, but an error elsewhere in the code. If the forked code blocks on shared resources, you can still end up with a deadlock in the normal way.

On the other hand, with dataflow concurrency, we cannot guarantee the absence of deadlocks in the coordination itself. The following example demonstrates a dataflow deadlock due to circular assignments.

```
def flow = new DataFlows()
task { flow.x = flow.y }
task { flow.y = flow.x }   //#1
```

#1 Deadlock!

However, for all practical cases, dataflow-based deadlocks are *reproducible*. The above example will *always* deadlock.

This has a huge benefit: it makes the coordination scheme unit-testing friendly! Aside from pathological cases, you can be sure that your code does not deadlock if your test cases do not deadlock.

FOR THE GEEKS: A PATHOLOGICAL CASE

Testability fails as soon as assignments to dataflow variables happen at random, like this: `flow.x = Math.random() > 0.5 ? 1 : flow.y`

Beside the testability, dataflow variables have another feature that makes them convenient to use in the concurrent context: their references are immutable. They never change the instance they refer to after the initial assignment. This makes them not only safe to use but also very efficient since no protection is needed for reading (non-blocking read). The benefit is greatest when the dataflow variable refers to an object that is also immutable, such as a number or a string.

But since dataflow variables can refer to any kind of object - which may happen to have mutable state - we may run into problems like below where a (mutable) list is assigned to a dataflow variable but possibly changes its state after assignment:

```
def flow = new DataFlows()
```

```
task { flow.list    = [0] }
task { flow.list[0] = 1   } //#1
println flow.list           //#2
```

#1 bad idea!
#2 prints [0] or [1] without guarantee

NOTE

Dataflow variables work best when used with immutable datatypes. Consider using the `asImmutable()` methods, use types that are handled by the `@Immutable` AST transformation, or safeguard your objects with agents (see section 17.4).

17.4.2 Dataflow on sequential datatypes

So far, we have only seen the merits of implicit task coordination with the dataflow concept for simple datatypes. This naturally leads to the question of whether we can use this concept just as well for processing more than simple data – and the answer is, "Yes, we can."

Think about it like this: implicit task coordination means that we automatically calculate a `result` *as soon as* dataflow variables `x` and `y` have assigned values. We can easily expand this concept to calculating a result *whenever* such an `x` and `y` are available!

In other words, we have an input channel that we can ask for `x` and a second one that gives us the next `y` to process. Whenever we have a pair of `x` and `y`, we calculate the result.

Listing 17.7 leads us into this concept by calculating statistical payout values that derive from the amount of a possible payout and the chance that this payout might happen. You can think of this as a gambling situation where you weigh the possible payout against your ante. Insurance companies follow a comparable approach when calculating risks.

The `operator()` method creates a `DataFlowOperator` and starts it immediately. The `chances` and `amounts` variables represent the input channels, and there is one output channel for the payouts. All the channels are of type `DataFlowStream` for implicitly coordinated reading and writing of input and output data. The closure that is passed to the `operator()` method defines the action to be taken on the input data. The next available, unprocessed item of each input channel is passed into it (`chance`, `amount`).

Listing 17.7 DataFlow streams and operators for implicit task coordination over sequential input data

```
import groovyx.gpars.dataflow.DataFlowStream
import static groovyx.gpars.dataflow.DataFlow.*

def chances = new DataFlowStream()
def amounts = new DataFlowStream()
def payouts = new DataFlowStream()

operator( inputs: [chances, amounts],
         outputs: [payouts],
        { chance, amount -> payouts << chance * amount }
)

task { [0.1, 0.2, 0.3].each { chances << it } }
task { [300, 200, 100].each { amounts << it } }

[30, 40, 30].each { assert it == payouts.val }
```

Note that the operator and the value assignments for the input channels all work concurrently but thanks to the implicit task coordination, we still have a predictable outcome.

The `DataFlowOperator` and `DataFlowStream` APIs are rather wide and full coverage is beyond the scope of this chapter. Please refer to the API documentation, the reference guide and the GPars demos for more details. There is one feature that shouldn't go unnoticed, though: dataflow operators are *composable*.

It is no coincidence that input and output channels are both of type `DataFlowStream`. The output channel of one operator can be wired as the input channel of a second operator. One can make a whole network of concurrent, implicitly coordinated operators.

17.4.3 Final thoughts on dataflow

Dataflow variables are very lightweight. You can easily have millions of them in a standard JVM.

They are also very efficient. A scheduler for dataflow tasks has additional information that allows picking tasks "sensibly" for execution.

Dataflow abstractions can help when writing unit tests for concurrent code. They can easily replace `Atomic*` variables, latches, and futures in many testing scenarios.

Most of all, dataflow is an abstraction that lends itself naturally for all those concurrent scenarios where the primary concern is the flow of data. Take the classical producer-consumer problem where a consumer processes data that producer delivers concurrently. It is all about the flow of data. Listing 17.7 is a more elaborate form of the same pattern, combining two producers, synchronizing on them effortlessly: we've solved the coordination and data flow aspects of the problem without even thinking about it!

It is a matter of modeling. We either model the flow of data indirectly through the concurrent operations that we perform on it or directly through dataflow abstractions.

Some experts go so far as to claim that without the need for data handling, concurrency is trivial and otherwise dataflow should be the first area to consider. This claim may be a little bit too bold, however. There are times when we need more control over task coordination than dataflow can provide. This is where actors enter the stage.

17.5 Actors for explicit task coordination

We have seen *predefined* task coordination with fork/join and map/filter/reduce and *implicit* task coordination with dataflow. The actor abstraction fills the hole of how to coordinate concurrent tasks *explicitly*.

Actors were introduced many decades ago and have undergone a rollercoaster ride of academic popularity, great hopes, challenges, disillusions, sleeping beauty, rediscovery, and recently resurgence in popularity. They've been at the heart of the Erlang concurrency and distribution model for a long time, proving the concept's value for parallel execution, remoting and building fault-tolerant systems.

Actors provide a controlled execution environment. Each actor is like a frame that holds a piece of code and calls that code under the following conditions:

- There is a message waiting in the actor's inbox.
- The actor is not concurrently processing any other message.

This description is the lowest common denominator between all the various actor concepts and implementations available. Beyond it, you will find all kinds of variations about whether or not an actor is allowed to have mutable state, whether messages have to be immutable, whether the actor and/or the messages have to be serializable, how their lifecycle is controlled, and so on. For the remainder of this chapter, we will avoid such controversy: whenever we use the word actor, we mean the GPars definition.

Listing 17.8 gets us started by creating three actors: a `decrypt` actor, an `audit` actor, and a `main` actor. The `main` actor sends an encrypted message to the `decrypt` actor, which replies with the decoded message. When the `main` actor receives that reply, it reacts to it by sending it to the `audit` actor, which in turn prints

```
top secret
```

Listing 17.8 Three actors for explicit coordination of decrypting and printing tasks

```
import static groovyx.gpars.actor.Actors.*

def decrypt = reactor { code -> code.reverse() } //#1
def audit   = reactor { println it }             //#2

def main    = actor   {                          //#3
    decrypt 'terces pot'                         //#4
    react { plainText ->                         //#5
        audit plainText                          //#6
    }
}
main.join()
audit.stop()
audit.join()
```

#1 reactor factory method
#2 reactor factory method
#3 actor factory method
#4 send message
#5 wait for reply
#6 send message

Hopefully by now you are comfortable with the static factory methods that GPars has consistently provided us. The `actor` #1 and the `reactor` #2 methods are two more examples of the same, living in the `Actors` class. They each return an `Actor` instance, which is started right away.

They both have a closure argument, telling them what the generated actor should do when its `act()` method is called, which happens as part of starting the actor. This is straightforward for the `actor{}` #3 factory method but a bit more involved in the case of `reactor{}`. Here, the given closure is wrapped so that it gets executed concurrently whenever a message is waiting in the inbox and the actor is not already busy. The message is passed to the closure and the closure result is replied to the sender. You can think of a reactor as having an `act()` method of

```
loop { react { message -> reply reactorClosure(message) } }
```

This construction is needed so often that the GPars team has put it into the `reactor` factory method for your convenience.

NOTE

You never call the `act()` method directly! This would undermine the actor's concurrency guarantees. Instead, you call the actor's `send(msg)` facility that puts the given message in its inbox for further processing. Sending is available in various shortcuts: the `send(msg)` method, `leftShift(msg)` to implement the << operator, and `call(msg)`, which enables the transparent method call (see xxx) that we use in listing 17.8 for sending messages (#4 #6).

Sending a message to an actor takes the form of an asynchronous request. The actor is free to process our message at any time. This also means that we do not wait for its response, unless we use the `sendAndWait()` method. When an actor replies to a message, it sends the reply to the originating actor. In listing 17.8, you see the `main` actor sending a message to the `decrypt` actor #4 and going into react mode #5, waiting for the reply message to arrive. The `decrypt` actor replies to the `main` actor, effectively sending the decrypted plain text as a message.

REACT MODE IS A STATE

Using an actor facility that makes the actor wait for a reply is an example of state. GPars supports such actor states but this is not common between various actor implementations. It's up to you to decide whether or not to use this kind of state.

Actors can be seen as asynchronous services. They wait idly until they have a message to process, do their job, and either stop or wait again. Running actors do not prevent the JVM from exiting; they are backed by a pool of daemon threads. This is why we need the last three lines in listing 17.8.

The `main.join()` waits until the main actor is finished. We can be sure that it has received the plain text and has sent it to the `audit` actor. But since the `audit` actor handles the request asynchronously, we cannot be sure that the printing has been done. We have to wait for the `audit` actor to finish as well by `audit.join()`. The `audit` actor is a reactor, though. It never finishes until we send it the `stop()` message. So

```
main.join()
audit.stop()
audit.join()
```

is the necessary coordination control that makes sure that the decrypted message appears on the console before our program exits. Try the program without these lines; it you run it several times you will see the output appearing at random.

There are so many conceivable applications of actors that we cannot possibly do them justice in this chapter. Table 17.3 lists some actor capabilities by method name.

Table 17.3 Actor capabilities (excerpt)

Method	**Capability**
`start()`	Starts the actor. Automatically called by the factory methods.
`stop()`	Accept no more messages, stop when finished.
`act()`	Contains the code to execute safely per message.
`send(msg)`	Passes a message to the actor for asynchronous sequential processing. Aliases for actor `x`: `x.leftShift(msg)`, `x << msg`, `x.call(msg)`, `x(msg)`.
`sendAndWait(msg)`	Passes a message to the actor for synchronous sequential processing. Waits for the reply. Comes with timeout variants.
`loop{}`	Do work until stopped.
`react{msg->}`	This is only available on subtypes of `SequentialProcessingActor`. It waits for a message to be available in the inbox, pops one message out of the inbox and passes it into given closure for execution. Comes with timeout variants.
`msg.reply(replyMsg)`	Sends the `replyMsg` back to the sender of the `msg`. Most useful inside a react closure where it is delegated to the processed `msg` so that it can be called without knowing the receiver.
`receive()`	Just like `react` but without a closure parameter to process. Returns the message. Comes with timeout variants.
`join()`	Waits for the actor to be finished before proceeding with current task.

Although this should give you an initial feeling for the Actor API, using it wisely is not quite as easy as it might seem. Out of all the concepts in this chapter, this is possibly the one at the lowest level of abstraction and with the highest potential for errors.

First, it is often suggested that actors should be free of side-effects, which is very restrictive since this doesn't allow printing to a console, storing a file, modifying a database, updating a user interface, writing to the network, and so on. A more practical requirement, however, is that only one actor should access one such device thus avoiding concurrent access. This is exactly what the `audit` actor in listing 17.8 does. The next time you see an actor presentation without such a safeguard, shout out loud!

Second, keep it simple. With many actors sending and replying to messages it is all too easy to run into deadlocks from circular references and other concurrency traps that we are here to avoid. They can also be difficult to debug and unit-test. If you cannot sketch your actor dependencies as easily as in figure 17.3, consider whether any of the other concurrency concepts may yield a simpler solution. They often do.

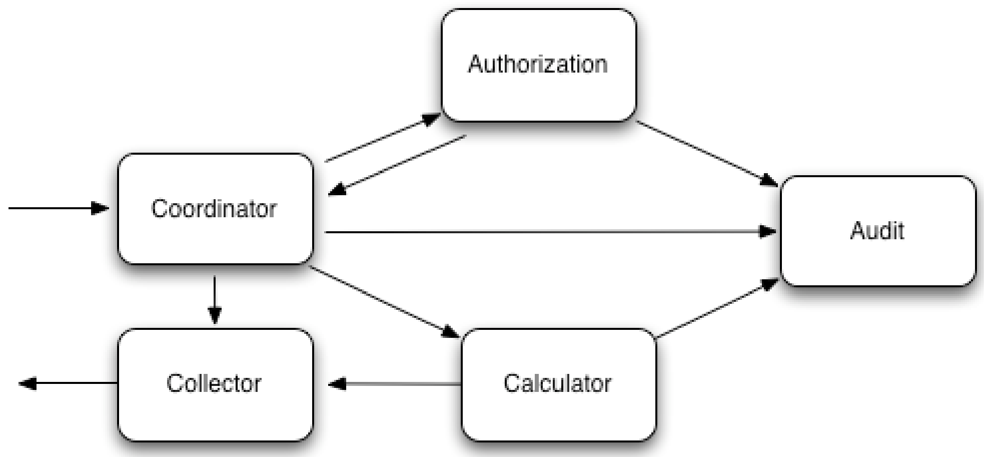


Figure 17.3 A simple example network of actors for processing a request. A coordination actor waits for the authorization reply and triggers a calculation. Many actors inform the audit actor. A collector returns the result.

Third, `sendAndWait()` is a troublesome feature. You may wait forever. So give it a timeout at least. But if that times out, what shall we do? Try again? The rule of thumb is that if you're using actors together with `sendAndWait()`, you've probably chosen the wrong concept.

When creating a network of actors you may get some inspiration for tailoring responsibilities along the lines of enterprise integration patterns as implemented in the Apache Camel project (see http://camel.apache.org/enterprise-integration-patterns.html and Camel in Action http://manning.com/ibsen/). If you start thinking in terms of Enricher, Router, Translator, Endpoint, Splitter, Aggregator, Filter, Resequencer, and Checker, you are on the right track.

17.5.1 Using the strengths of Groovy

We have seen that Groovy provides a very clean and concise API for creating and using actors. Listing 17.8 is pretty much the most compact piece of actor code that one can think of without sacrificing readability.

But there are two more Groovy features that make our language particularly interesting in this context: assigning event hooks through meta programming and using dynamic dispatch for reacting appropriately based on the message type.

Let's start with meta programming. Listing 17.9 uses a standard reactor that simply calls its own `stop()` method as soon as it receives a message. We would like to get notified when the actor stops and look into its inbox. What we will see is the remaining stop message:

```
[Message from null: stopMessage]
```

Listing 17.9 Hooking into the actor lifecycle through meta programming

```
import static groovyx.gpars.actor.Actors.*

def stopper = reactor { stop() }
stopper.metaClass.afterStop = { inbox -> println inbox }
stopper.send()
```

Actors can implement the optional `afterStop()` message for that purpose but the standard reactor that we use above has no such method. We don't need to write our own `Actor` implementation as we can add such a method through the metaclass.

Besides `afterStop()` There are other lifecycle hook methods like `afterStop()`: `onTimeout()`, `onException(throwable)`, and `onInterrupt(throwable)`. The final two in this list are particularly important since proper exception handling is easily overlooked in a concurrent context.

The third benefit of using Groovy for actors is its dynamic method dispatch. Whenever actors respond differently based on the message type they receive, there is some dispatch to be done - either manually or automatic.

Listing 17.10 compares the two approaches. The manual reactor switches on the message type, effectively taking a do-it-yourself approach to method dispatch. The auto message handler in the second part of the example defines `when` clauses for each message type and leaves the dispatch to Groovy.

Listing 17.10 Comparing manual and automatic method dispatch for message-type aware actors

```
import static groovyx.gpars.actor.Actors.*

def manual = reactor { message ->
    switch (message) {                        //#1
        case Number: reply 'number'; break
        case String: reply 'string'; break
    }
}

def auto = messageHandler {
    when { String message -> reply 'string' } //#2
    when { Number message -> reply 'number' }
}
```

#1 Self-made dispatch
#2 Groovy method dispatch

The difference may not look very significant in this small example but it makes a considerable difference when managing any reasonably sized actor of that kind. The `messageHandler` is again a factory method that returns an `Actor`, which happens to be a `DynamicDispatchActor`. You can use it in a number of different ways: through the factory method, by calling various constructors that allow registering of `when` closures, or by subclassing and implementing `onMethod(messageType)` hooks.

BY THE WAY

Static languages - the ones that have no dynamic method dispatch - have a hard time supporting actors with dispatch on the message type in a way that doesn't compromise their static language paradigm.

Actors can be difficult to handle but compared to other low-level constructs for explicit task coordination they have a pleasant structure and the send-reply-react scheme is easier to understand and handle than most of Java's built-in facilities.

Now that we have seen predefined, implicit and explicit task coordination we have the difficulty of choosing between them. Luckily, there is yet another candidate that we can delegate to.

17.6 Agents for delegated task coordination

Delegation is my favorite strategy. Whenever I don't know what to do, don't want to do it, or simply don't want to decide, I happily hand the work to a delegate. Delegates are abundant. They often appear as agents (think "real-estate") that are happy to work on your behalf. GPars can also create such helpful fellows and we use them for working on shared mutable state.

When it comes to shared mutable state, lots of concurrency experts shiver with disgust. But it is totally unavoidable as long as we integrate with Java, use its common datatypes, and call its methods - not only in the JDK but also in the vast space of open-source, commercial, and home-grown APIs that we rely upon.

So instead of denying reality, it’s more pragmatic to look for ways to safeguard our valuable assets. Listing 17.11 uses an agent to safeguard access to a String that we change in a concurrency-safe manner. We update the value by sending update instructions to our agent that does all the tiring work for his client.

IMMUTABILILITY IS NOT ENOUGH

Note that we do not need to safeguard the String as such since Strings are immutable. Anyway, we have to safeguard the reference that holds the String to make sure that the concatenation has been done on our original String - and not a concurrently changed one.

Listing 17.11 Safeguarding a String for concurrent modifications

```
import groovyx.gpars.agent.Agent

def guard = new Agent<String>()

guard { updateValue('GPars') }
guard { updateValue(it + ' is groovy!') }

assert "GPars is groovy!" == guard.val
```

Agents protect a secure place where the safeguarded object cannot be changed by anyone but the agent. Instructions on how to change the object are sent to the agent. Again, the usual methods are available; listing 17.11 demonstrates `send`, `leftShift`, <<, `call`, and a transparent method. The `updateValue()` message is used when the safeguarded object itself is to be replaced by a new one.

Agents can easily be used in combination with all the other concurrency concepts we've seen in this chapter. They are a very simple yet ubiquitously useful tool for the concurrent programmer.

17.7 Concurrency in Action

Let's round up our tour through Groovy concurrency with an example that fetches stock prices from the web in order to find the most valuable one. This task has recently gained some popularity for a number of reasons:

- Fetching web pages is slow compared to local calculations. Therefore, using concurrency is promising no matter how many processing cores we have.
- The effect can be achieved with many different approaches, which gives us freedom of choice.
- There are many solutions published for different languages that we can compare our solution against.

We start with the easy part: fetching the year-end closing price of a given stock ticker. Listing 17.12 connects to a Yahoo service that provides this information in CSV format. The result of fetching its URL looks like this[2]:

```
Date,      Open,   High,   Low,    Close, Volume,   Adj Close
2009-12-01,202.24,213.95,188.68,210.73,19068700,210.73
```

From that data, we need the fifth entry in the second line, which is what the `getYearEndClosingUnsafe` method returns. This method doesn't handle any problems with connecting to the service, so we've created an exception-safe variant `getYearEndClosing` for convenience.

Listing 17.12 Fetching the year-end closing price for a given stock ticker symbol

```
class YahooService {
    static getYearEndClosingUnsafe(String ticker, int year) {
        def url = "http://ichart.finance.yahoo.com/table.csv?" +
                "s=$ticker&a=11&b=01&c=$year&d=11&e=31&f=$year&g=m"
        def data = url.toURL().text
        return data.split("\n")[1].split(",")[4].toFloat()
    }
    static getYearEndClosing(String ticker, int year) {
        try {
            getYearEndClosingUnsafe(ticker, year)
        } catch (all) {
            println "Could not get $ticker, returning -1. $all"
```

[2] slightly formatted for better readability

```
                return -1
            }
        }
    }
```

Providing an exception-safe variant in addition to an unsafe method allows both convenience and caller-specific exception handling where each is required.

The API design of `YahooService` goes for static methods with immutable parameter types, which makes it very concurrency-friendly even though the code shows no trait of being concurrency aware. It almost entirely avoids access to foreign objects with the exception of `println`. Printing this way is considered a concurrency design flaw and only acceptable when printing a single line, knowing that the `PrintStream` synchronizes internally.

Stateless methods are often frowned upon as being against traditional OO style but for concurrency-friendly services, they make a lot of sense.

Now, let's assume we wish to check the prices for Apple, Google, IBM, Oracle, and Microsoft using the following stock ticker symbols:

```
def tickers = ['AAPL', 'GOOG', 'IBM', 'ORCL', 'MSFT']
```

Then we could sequentially find the most valuable one by collecting all prices together with its ticker symbol and selecting the one with the maximum price:

```
def top = tickers
    .collect { [ticker: it, price: getYearEndClosing(it, 2009)] }
    .max { it.price }
```

Nothing fancy here. This is all plain non-concurrent code that connects to the `YahooService` for one stock ticker after the other.

Listing 17.13 makes one small addition to turn this into a concurrent solution: by calling `makeTransparent()` on the tickers, which results in calling the collect logic concurrently. This fork/join approach requires us to put the code inside a `withPool` scope.

Listing 17.13 Fetching prices concurrently with fork/join

```
import static groovyx.gpars.GParsPool.withPool
import static YahooService.getYearEndClosing

def tickers = ['AAPL', 'GOOG', 'IBM', 'ORCL', 'MSFT']

withPool(tickers.size()) {
    def top = tickers.makeTransparent()
        .collect { [ticker: it, price: getYearEndClosing(it, 2009)]}
        .max { it.price }
    assert top == [ticker: 'GOOG', price: 619.98f]
}
```

Note that we use the `withPool` method with an argument to define the pool size. We want to have a concurrent task for processing each ticker such that we don't limit our network usage by our processing capacity. We go for highest concurrency even on a machine with a single core.

The solution in listing 17.13 is arguably the simplest one that we can get and it is so close to optimal that if you are a practitioner, you may want skip the rest of this section and go right to the summary. The concurrency-addicted developer may want to read on. There are some interesting variants coming!

Calculating the maximum once we have all prices available is a very quick operation and usually not worth optimizing but for the sake of exploring the concepts, we do it anyway. Listing 17.13 first collects all prices and starts calculating the maximum only after all the prices have been fetched. We could do a little bit better.

Suppose that AAPL and GOOG have been fetched but the remaining ones are still loading. We could use that network delay to eagerly calculate the maximum of the prices we already know. Listing 17.14 introduces what looks like a minimal change in the code to make this happen but is a rather fundamental change in scheduling.

Listing 17.14 Fetching prices concurrently with map/filter/reduce

```
import static groovyx.gpars.GParsPool.withPool
import static YahooService.getYearEndClosing

def ticker = ['AAPL', 'GOOG', 'IBM', 'ORCL', 'MSFT']

withPool(ticker.size()) {
    def top = ticker.parallel
        .map { [ticker: it, price: getYearEndClosing(it, 2009)] }
        .max { it.price }
    assert top == [ticker: 'GOOG', price: 619.98f]
}
```

We have gone from fork/join to a map/filter/reduce approach since finding a price is conceptually a mapping from a ticker symbol to its price and finding the maximum is a special logic of reducing the result set.

Note that neither `max` nor any other reduction method guarantees that we process prices as soon as two of them are available. In the worst case, we wait for the two candidates that finally turn out to be the slowest ones. But in average, we win.

Now, is listing 17.14 the best we can get? Well, there are so many options and we are entering the space of personal taste. Some interesting variants come with dataflow. Let's explore at least one in listing 17.15 that spawns a task for each ticker symbol, which is used as the dataflow index. When calculating the maximum, we refer to the price data flow entry thus implicitly waiting if the price has not yet been fetched.

Listing 17.15 Fetching prices concurrently with DataFlows

```
import groovyx.gpars.dataflow.DataFlows
import static YahooService.getYearEndClosing
import static groovyx.gpars.dataflow.DataFlow.task

def tickers = ['AAPL', 'GOOG', 'IBM', 'ORCL', 'MSFT']

def price = new DataFlows()
tickers.each { ticker ->
    task { price[ticker] = getYearEndClosing(ticker, 2009) } //#1
}
def top = tickers.max { price[it] }                          //#2
assert top == 'GOOG' && price[top] == 619.98f
```

#1 Set when available
#2 Read sequentially

We get the same concurrency characteristics as with map/filter/reduce in listing 17.14 but without the need for the extra ticker/price mapping.

This example is very well suited to investigate further concepts and you will find some more demos in the GPars codebase. Look for the `DemoStockPrices*` scripts. There are also actor-based solutions but I personally find them less attractive since the problem doesn't really call for explicit coordination. They also tend to be lengthier in terms of the code required.

Another interesting approach would be to use the dataflow `whenBound` feature where one can deposit a closure that is executed asynchronously after a value has been bound to a dataflow variable. However, this comes with some considerable effort in terms of coordinating the tasks to assert that all prices have been processed and also shielding the temporary maximum against concurrent access. This approach has the appeal of always calculating the currently best-known maximum as early as possible but it is anything but simple.

Weighing algorithmic appeal against simplicity is a design choice that we often encounter in concurrent scenarios. Don't think twice. Go for simplicity!

17.8 Summary

As a Groovy or Java programmer, you don't have to be afraid of the multi-core era. Java has provided us with a solid, battle-tested foundation for concurrent programming that Groovy uses to build more high-level abstractions upon.

Now is the time to make yourself comfortable with the various approaches to coordinate concurrent tasks. The predefined control flow on collections through fork/join and map/filter/reduce is possibly the easiest one to understand and start with. Implicit coordination with dataflow should be your choice whenever your focus is on the flow of data rather than the manipulation steps. Explicit control with actors should be your last consideration when no other concept applies. And regardless of how you coordinate your concurrent tasks, always consider using agents to protect shared mutable state.

It goes without saying that a mere chapter that tries to cover so many concepts cannot do justice to the full API of such a rich project as GPars and necessarily fails to present such a wide topic as concurrency in all its beauty.

Even more concepts are expected in the near future and may be available by the time you read this. Keep an eye on http://gpars.codehaus.org to get the latest updates.

Please allow me to point your attention to the grace and elegance that Groovy has shown once again in this chapter. The functional nature of closures blends naturally with the need to demarcate pieces of code for concurrent execution. Object iteration methods provide a perfect base for fork/join. Last not least, actors profit from dynamic method dispatch and meta programming. I'm so glad we have this language!

18 Domain-Specific Languages

Throughout this chapter, we'll cover:

- How Groovy allows you to write Domain-Specific Languages (nicknamed *DSLs*), i.e. languages tailored towards representing a particular domain of knowledge,
- How to concretely integrate DSLs in your applications,
- What the various techniques are to create readable and expressive languages,
- And how to test, secure and provide good error reporting.

Various papers and studies will give you statistics about project successes and failures. On the other end of the spectrum, proponents of development methodologies will tell you about various techniques and approaches you should adopt to ensure that your projects have better chances to succeed. You can read a lot of interesting literature on those topics and I would argue that oftentimes, a common denominator in those sources about what leads a project on its path to success or failure is the quality of communication between the different parties involved, how the various stakeholders exchange information and cooperate together towards a common goal of producing quality software that delivers on their promises at solving a particular domain problem.

Languages are at the root of any kind of communication and involve two interlocutors. A *Subject Matter Expert* (often reduced to the SME acronym) can write specifications in his mother tongue, say, English, with tons of very domain specific words and concept names that will be read by software developers. A developer can also somehow speak with a computer with different languages to tell it about the business rules of the application the SME is longing to play with. The former will use a natural language while the latter will use one or several general-purpose programming languages. There is obviously some translation process involved to codify requirements into executable code. And this process is usually not that straightforward as there may be ambiguities in the natural language used — a word can have several meanings depending on the context. Moreover, different approaches exist for implementing the same behavior — using different patterns, algorithms or idioms. More so than typing issues or lacking null pointer checks, what introduces bugs in our machinery is the imperfection of our understanding of each other's words and their meaning.

Let's step back a little and reflect on these considerations. In this Babel of languages, what if we had one language that everybody would unambiguously understand? What if this language contained words that all stakeholders would put the same meaning on, that would be both readable and expressive? And imagine that *Esperanto* even be understood by computers themselves? But does such a dreamed up ubiquitous language can even exist? Well, I wouldn't write that prose if I didn't think Groovy could play a central role here, and this chapter wouldn't be named "Domain-Specific Languages" if one couldn't create a special language in Groovy that would help reduce the gaps in understanding each other and help us make our project a success rather than a failure.

18.1 Groovy's flexible nature

The grammar of the Groovy language directly derives from Java 5, meaning that you can often copy and paste some Java code into your Groovy programs and have it run as is without any modification. However, as a Java developer, as you learn Groovy, you'll progressively start writing more idiomatic Groovy code. Over the course of the past chapters, you've discovered many language features and APIs which will make your code groovier and more concise! For instance, you'll get rid and even forget about those boring semi-colons — isn't the compiler supposed to be clever enough to figure out when a statement ends? You'll also quickly omit parentheses in various places, like for example in your `println` statements. You are rapidly benefiting from Groovy's flexible nature, at the syntax level, as well as the API level — after all, `println` is already a shortcut notation for `System.out.println`! Let's review what Groovy offers out of the box for making your code looks a little nicer on the eye.

18.1.1 Back on parentheses omission

In Groovy, the parentheses can be omitted under some circumstances. Basically, all top-level statements or expressions, called *command expressions*, can benefit from that rule.

Imagine you are a NASA engineer, and you've sent a rover on the ground of planet Mars. In the comfy seats of the Earth base commandment station, from miles and miles away — certainly with a delay due to the speed of light at which information travels — you are planning the journey of your little robot, on the rocky soil. You will tell your robot to move left and right. In Java, you would need to create a class (lots of surrounding boilerplate code), and describe the orders in a method, whereas in Groovy, you could just put all the orders in a simple script, but let's put all the accompanying code aside. What could the orders look like?

In Java, you would need some parentheses and a final semi-colon:

```
move(left);
```

Whereas in Groovy, this would look like an English-like imperative sentence:

```
move left
```

We don't gain yet that much, but the Groovy variant is free of punctuation noise.

In either case, you would simply need to have a method named `move()` defined in your script or class taking one parameter of type `Direction`, perhaps even an Enum value.

But here, we are only showing the command the operator sends to his remote robot! If we just have the `move left` order in our Groovy script and execute it, we will get some exception telling us the `left` variable is not found, and if it is found, we will get another exception afterwards indicating the `move()` method could not be found. So let us have a look at what the full script could look like:

Listing 18.1 Our naïve approach with a full script

```
import static Direction.*                  #2

enum Direction {                           #1
    left, right, forward, backward         #1
}                                          #1

class Robot {                              #3
    void move(Direction dir) {             #4
        println "robot moved $dir"         #4
    }                                      #4
}

def robot = new Robot()                    #5

robot.move left                            #6
```

Firstly, in #1, we define an enum for the directions, so that our robot can understand the left, right, forward and backward movements. In #2, we use a static import to import all the enum constants, so they are available in the body of our script. Then who receives the commands? Ah yes, the robot! So we, obviously, need a robot! In #3, we create a Robot class. This little robot has a move() method in #4, taking a Direction value. Then, in #5, we instantiate our Robot class to create a robot. And at last, in #6, we can tell our robot to actually move left.

That's quite a bit of code for telling our robot to move left, isn't it? A Domain-Specific Language isn't supposed to be concise and readable? For a single move, we need an enum definition, a static import, an instantiation, and the final command. All in all, fifteen lines of code for a mere command! Furthermore, our command does not really look like the one we mentioned earlier: there's a robot prefix! Have I lied to you? Can't we do better? Yes, we can!

Okay, first of all, we could start by speaking of integration. We should differentiate the infrastructural code, from the business code: the code of the robot, the directions, the instantiation of the robot are all about infrastructure, whereas the order we send to the robot, that's the actual business code, our Domain-Specific language code.

This is a good practice to cleanly separate both kinds of code, and that's also what will allow us to streamline the DSL bits, so they remain short, concise and readable. All the cleverness of the DSL will come from the infrastructural code, and how we integrate everything together. This dichotomy is also a separation of concerns: the infrastructure stays the same, or evolves at its own pace, and the same is true for the business code, as the orders sent to our rover vary depending on where the robot is, and where we want it to go.

GrooovyShell will be our weapon of choice for evaluating our business rules: this will be the class that will be integrated in our backend (it could be a pure Java or Groovy backend, or a mix of the two). The Direction enum and the Robot class will be part of our infrastructural code, already pre-compiled on our classpath, as part of the build process of our overall application. So compared to our full script from earlier, they will be taken out, in their specific files (in Listing 2 and Listing 3), and will be part of our domain model package (projectmars.model).

Listing 18.2 Our enum for directions: `projectmars.model.Direction`

```
package projectmars.model

enum Direction {
    left, right, forward, backward
}
```

Listing 18.3 Our core robot class: `projectmars.model.Robot`

```
package projectmars.model

import static projectmars.model.Direction.*

class Robot {
    void move(Direction dir) {
        println "robot moved $dir"
    }
}
```

Given those domain model classes, our business rules, in the form of a Groovy script, are already shorter, as shown in Listing 4.

Listing 18.4 Our updated business rules

```
import projectmars.model.Robot                  #1
import static projectmars.model.Direction.*     #2

def robot = new Robot()                         #3

robot.move left                                 #4
```

#1 We need to import the Robot class so we can give it orders
#2 A static import for directions for the movement directions
#3 The instantiation of the robot

#4 The actual move order

What is still missing in our big picture is the actual code integrating everything together, which is supposed to be using the GroovyShell class we mentioned earlier. For simplicity sake, we keep business rules in a simple multiline Groovy string, but they can come from some configuration file, from a database, or entered interactively in a kind of console application. Listing 5, shows our concrete integration. As our overall application is so far pretty simple, we keep our application main entry point inside a Groovy script, but obviously, this could also be a more involved full-blown class with more responsibilities.

Listing 18.5 `Main.groovy`, our integration and main entry point of our application

```
package projectmars.integration                        #1

def shell = new GroovyShell()                          #2
shell.evaluate '''                                     #3
    import projectmars.domain.Robot                    #3
    import static projectmars.domain.Direction.*       #3
                                                       #3
    def robot = new Robot()                            #3
                                                       #3
    robot.move left                                    #3
'''                                                    #3
```

Main.groovy lives in the projectmars.integration package as shown on #1, instantiates the GroovyShell class in #2, then we call its evaluate() method in #3, which takes a String — here a multiline String — in parameter. That method can also take a Reader or a File as parameter, if you are reading the scripts to evaluate from a different place, like a remote location, or from the file system.

EXTERNAL DSL FILE

In our example from , we evaluated a DSL which was in the form of a string. But the idea here, is that your DSL might come from an external file. You could as well store this DSL in a database, or elsewhere. Instead of the verbatim string, you could call shell.evaluate(new File(filename)), pointing at the file containing the code of your DSL.

Given we are done with our infrastructural code (our domain model classes and our application main entry point integrating the business rules), let us have another look back at our business logic from Listing 4 Our updated business rules. We have an import and a static import, a robot instantiation, and eventually the command we send to our robot. Is it perfect? Well, for one, we could consider the two imports as some useless boilerplate code (at least from the perspective of the person writing the business rules). Same thing for the instantiation of the robot: it is probably boilerplate as well, but above all, doesn't it give you a bitter taste in your mouth? We need a reference to the robot. That will definitely help us test our business code if we could inject a mock robot at some point, and if one day our robot is upgraded to a newer version, we could later inject a different instance of the robot. Last thing, the style used for sending our commands to the robot isn't looking like what we promised in introduction of our chapter, as we have this robot prefix.

To summarize, we want to:

- Get rid of the imports,
- Inject the robot more transparently,
- Improve the way we send orders to the robot.

We'll take care of those three points in the next sections.

18.2 Variable, constant and method injection

Tackling the injection first will permit us to get rid of the instantiation of the class and to remove the related import. How can we achieve this? By using the script's binding: every script has got a special kind of map in which dynamic variables can be saved and looked up.

Let’s have a look at a simple example first, evaluating math expressions: Listing 6 shows how the binding can be created in #1, and passed to the GroovyShell in its constructor in #2. We created a distance and time variables in the binding, and those two variables are actually available when we evaluate our math formula. No MissingPropertyException is thrown, the variables are present globally in the body of that script, without any prior definition or particular explicit lookup. You notice we assign the quotient of distance over time into a speed variable in #3. We have neither “def-ed” that variable, nor used a different approach than a plain assignment to pass the result of the calculation to the binding. In #4, the variables we had put into the binding are still there — and we haven’t updated their values, but we could have done that. And in #5, we discover that the binding now contains an additional variable: our speed variable is here, containing the result of the computation.

Leveraging the script’s binding is a great technique to exchange variables and values in and out, during the execution of a script.

Listing 18.6 Exchanging variables and values through the Script's binding

```
def binding = new Binding([distance: 11400, time: 5*60])  #1

def shell = new GroovyShell(binding)                       #2
shell.evaluate '''
    speed = distance / time                                #3
'''

assert binding.distance == 11400                           #4
assert binding.time == 5*60                                #4
assert binding.speed == 38                                 #5
```

With that knowledge in mind, you have guessed that the binding is going to be the approach we will be using for injecting the robot instance into our command script. And there is even a bonus: thanks to the duck-typing approach, we can just *use* that robot instance without having to specify in any way that that robot is an instance of Robot, and especially also without having to import the Robot class! As demonstrates Listing 7, we can remove one import and inject the robot into our DSL script, through a binding object.

Listing 18.7 Injecting a variable into the script's binding

```
def binding = new Binding([robot: new Robot()])            #1

def shell = new GroovyShell(binding)                       #2
shell.evaluate '''
    import static projectmars.domain.Direction.*

    robot.move left
'''
```

#1 We create a `Binding` object, that takes a map as parameter, containing our injected robot instance
#2 We pass that binding to the `GroovyShell` constructor

We have injected a robot into our DSL script. We don’t need to instantiate a robot object, and we managed to get rid of one import. But we are left with the static import for the directions, and the prefixed move method.

We managed to inject the robot variable, we can apply the same technique to inject the direction constants. This is a first approach we will take, but we will also have a look at a couple more approaches.

18.2.1 Injecting constants through the binding

In Listing 8, we enumerate the various directions manually, and we add them in the binding to be passed to the shell.

Listing 18.8 Injecting constants through the binding

```
def binding = new Binding([
    // injecting the robot
    robot:    new Robot(),
    // injecting the direction constants
    left:     Direction.left,             #1
    right:    Direction.right,            #1
    forward:  Direction.forward,          #1
    backward: Direction.backward          #1
])

def shell = new GroovyShell(binding)
shell.evaluate '''
    robot.move left                       #2
'''
```

This approach is certainly the simplest one: in #1, we add each and every direction manually into the binding, and in #2 we notice that the static import for the constants has totally disappeared from the script DSL. However, it is a bit fragile: what happens if we add a new direction value? We would have to update both the enum definition, as well as the integration where we inject the enum values into the binding. Fortunately, Groovy's magic will empower us to make things less brittle by using the `collectEntries()` method and spread map operator!

Listing 9 explains how we can inject all the `Direction` enum constants into the binding automatically, rather than manually. The `collectEntries()` method creates a map which is the association of the name of the enum values, and the respective values, and the spread map operator will merge those entries into the binding map.

Listing 18.9 Using the enum values with `collectEntries()` and the spread map operator

```
def binding = new Binding([
    robot: new Robot(),
    *:Direction.values().collectEntries { [(it.name()): it] }
])
```

This new binding definition still injects the robot, and spreads the content of a map into the binding map, to form one consistent map. The obvious benefit is that if ever you need more directions, maintenance is going to be easier since you just need to update the `Direction` enum, and not have to touch your integration code. No duplication!

With enums, we managed to solve the problem of maintenance and duplication elegantly, but sometimes, you don't have enums at your disposal — perhaps some legacy classes and interfaces you have no control over, or you don't want to import all of the enum values but just some. So you can go with manually adding each constant like we did in Listing 8 Injecting constants through the binding. But there are also other ways to add variables and constants into your DSL scripts. We will have a look at two techniques: adding imports and static imports automatically, and using a special base script class for your DSL scripts.

18.2.2 Adding imports and static imports automatically

Our integration so far used `GroovyShell` for evaluating our robot moves, and we saw we could pass paramaters like a `Binding` in order to pass variables and constants to be available during the execution of the script. But `GroovyShell`'s constructor also takes a `CompilerConfiguration` as parameter. And through the latter, we are able to define *compiler customizers* (there are three kinds of them, and one can create his own), as well as a *custom base script class*.

Let us start with looking at one special customizer: the import customizer. The name is pretty explicit: an import customizer helps you customize the imports of your scripts and classes. With it, you are able to add imports, static imports, as well star imports and star static imports. You can also do type aliasing for your imports and static imports.

Have a look at Listing 10, where we continue improving our integration script, by only injecting the robot variable in #1, by creating an import customizer in #2, then adding a static star import for the direction enum values in #3, specifying the customizer to be used by the compiler configuration, and eventually passing the configuration to the `GroovyShell` constructor in #5.

Listing 18.10 Using import customizers to add imports transparently

```
import org.codehaus.groovy.control.CompilerConfiguration
import org.codehaus.groovy.control.customizers.*

import projectmars.domain.Robot
import projectmars.domain.Direction

def binding = new Binding([robot: new Robot()])                  #1

def importCustomizer = new ImportCustomizer()                    #2
importCustomizer.addStaticStars Direction.class.name             #3

def config = new CompilerConfiguration()
config.addCompilationCustomizers importCustomizer                #4

def shell = new GroovyShell(binding, config)                     #5
shell.evaluate '''
    robot.move left
'''
```

#1 We now only inject variables into the binding
#2 We create an import customizer
#3 The `Direction.*` enum values are statically imported with a star import
#4 The import customizer is added to the compiler configuration object
#5 The compiler configuration is passed as argument of the `GroovyShell` instance

Given you have instantiated an `ImportCustomizer`, you can add:

- normal imports: `addImports(String… fqnClassNames)`
- an aliased import: `addImport(String alias, String fqnClassName)`
- a static import: `addStaticImport(String fqnClassName, String fieldName)`
- an aliased static import : `addStaticImport(String alias, String fqnClassName, String fieldname)`
- star imports: `addStarImports(String… packageNames)`
- static star imports: `addStaticStars(String… fqnClassNames)`

In our case, we only needed the latter variant that allowed us to add a static star import for all the enum constant values.

Compiler configuration customizers are interesting for Domain-Specific Languages purposes, and we will discover further down this chapter the other customizers for securing the scripts you are running, for applying transformations, or even creating your own customizers. But before that, let's come back on our original goals with our Mars rover. From our original script, we managed to get rid of the imports, and to inject the robot instance. However, the movement of the robot is still "prefixed", and is not as concise as it could be:

```
robot.move left
```

We want the movements to be sent to the robot, but we would like the code to look as if we were speaking directly to the robot, telling him explicitely "move left", because he knows we are talking to him — a pretty clever robot understanding human speech! But a move method would be a method on the current script that is running, not on the robot. So can we, somehow, redirect the methods to the robot? What we are looking for here, is a way to inject methods. There are some interesting approaches to do that.

18.2.3 Injecting methods into a script

A naïve approach could be to append methods at the end of our script code, before it gets evaluated, as illustrated by the approach in Listing 11. You would create your own evaluation method that would call `GroovyShell#evaluate`, but which would in turn do string concatenation to append each method definitions you would need to be carried over to our robot instance.

Listing 18.11 Appending method definitions to a script code

```
shell.evaluate '''
    move left
''' + '''
    def move(dir) {
        robot.move dir
    }
'''
```

This naïve approach is not ideal for a number of reasons. The implementation is fragile because you put code in a String that is not easily *refactorable*. For maintenance, this would also be problematic as you would have to update those appended method definitions when the Robot class is evolving. And last but not least, if the scripts end users are sending you are bogus and don't compile properly, you may get weird compilation error messages, as the parser would see some def token afterwards. So although this approach is easy, it should be avoided.

A proper way to add methods to the script class is to use a custom base script class. Scripts are all extending the groovy.lang.Script base abstract class, and CompilerConfiguration actually allows us to define a different base class for our scripts (as long as it is extending groovy.lang.Script).

First of all, we need to create our own script base class, as shown in Listing 12. Our class in #1 is declared abstract like its parent, and extends groovy.lang.Script. We then simply add a move() method with the same signature as the one of the robot in #2. In #3, we retrieve the robot from the script's Binding — the same binding that we pass to GroovyShell. Then in #4, we can delegate the call to the robot.

Listing 18.12 Defining a custom base script class

```
package projectmars.integration

import projectmars.domain.Direction

abstract class RobotBaseScriptClass extends Script {     #1
    void move(Direction dir) {                           #2
        Robot robot = this.binding.robot                 #3
        robot.move dir
    }
}
```

Now that we have our base script class ready, it is time to put it to good use, thanks to another option of CompilerConfiguration, as demonstrates Listing 13.

Listing 18.13 Configuring and using a custom base script class

```
import org.codehaus.groovy.control.CompilerConfiguration
import org.codehaus.groovy.control.customizers.*

import projectmars.domain.Robot
import projectmars.domain.Direction

import projectmars.integration.RobotBaseScriptClass              #1

def binding = new Binding([robot: new Robot()])

def importCustomizer = new ImportCustomizer()
importCustomizer.addStaticStars Direction.class.name

def config = new CompilerConfiguration()
config.addCompilationCustomizers importCustomizer
config.scriptBaseClass = RobotBaseScriptClass.class.name         #2

def shell = new GroovyShell(binding, config)
shell.evaluate '''
    move left
'''
```

#1 Import our new script base class
#2 Specify the base script class to use for script that will be compiled through `GroovyShell`. The class name must be passed as a fully qualified class name, in the form of a String.

NOTE:

When you add a getter in your base script class, for example `getMyConstant()`, you can then access it with `myConstant` in your script, as if it were passed through the binding. So that is another way of injecting a constant in the binding. However, if you defined a `setMyConstant()` setter method, this method would not be called upon assignment of `myConstant`, as the variable would go into the binding. If you wanted to call that setter, you would have to call it explicitely in the form of its method call.

Coming back to our abstract base script class, you certainly noticed we replicated the `move()` method from the `Robot` class. The script base class serves as a façade. The decoupling is certainly interesting, and here, we currently only have one method, but you would have to put a delegate method for each method of the `Robot` class otherwise.

If you recall from previous chapters talking about AST Transformations, you might find that using the `@Delegate` transformation a good fit for delegating methods. Listing 14 shows how to achieve this idea. The transformation will automatically add all the methods from `Robot` at compile-time in your script base class. You will also notice that we combined the `@Lazy` transformation as well, as the binding is populated after the class initialization and instance contruction.

Listing 18.14 Using `@Delegate` for method injection and delegation

```
abstract class RobotBaseScriptClass extends Script {
    @Delegate @Lazy Robot robot = this.binding.robot
}
```

The approach of delegation is good if you want to delegate all method calls, but if you are just interested in a specific set of methods, our previous solution worked well (by copying the signatures of our robot and delegating to the robot's methods). You can as well use the same approach that we used earlier for injecting variables, through the binding, by adding method closures into the binding, as we shall see further, when you want to manually pick some methods, or add some dynamic routines and don't want to have to use a custom base script class.

18.2.4 Adding closures to the binding

The script binding is a mere wrapper around a map. In that map of variables, we have the keys that represent the variable names, and the values that are the actual variable values. A value can be anything… including closures! And calling closure variables just looks like a plain method call.

You can quickly wrap method calls to a certain object instance with the method closures (sometimes called method pointers too), using the `.&` notation. If we revisit our previous examples, we could get a reference to the `move()` method of `Robot`, and add it to the binding of our script as Listing 15 shows.

Listing 18.15 Using a method closure to inject a method

```
def robot = new Robot()
def binding = new Binding([
    robot: robot,
    move: robot.&move              #1
])
```
#1 Obtain a method closure reference which wraps a call to the `robot`'s `move()` method

In this case, we are only injecting one method, and if we had to add all of them (robot's methods), we would have to manually add them all, or resort to using some reflection to find out all the available methods. But for one-off utility methods that need to be available to the script, using the binding in that way is very easy without necessitating the need for a custom base script class. Furthermore, when the functions needed can be totally dynamic, depending on the context, the ability to add any closure, referenced under any name, can be very useful.

Taking this idea further, what if our DSL was able to be case insensitive? Users of that DSL could make mistakes in their casing, and still have the robot obey their orders. In that situation, we can't add all possible methods and constants in all the possible combinations of uppercase and lowercase letters in the script base class or in the binding. So how could we proceed? We have two solutions at our disposal, again with custom base script classes or with custom binding classes: for arbitrary casing for method calls, we leverage our custom base script class, and for constants or variables, we'll use a custom binding class.

For arbitrary casing for our methods, Listing 16 Implementing `invokeMethod()` in the script base class shows how we added an `invokeMethod()` implementation in #1, to our base script class to intercept method calls. The calls are then delegated to the `robot` variable stored in the binding in #2. We use the GString method calls, putting the name into lowercase, and we *spread* the argument of the calls (using the spread operator `*`) back in the call of the method of our rover.

Listing 18.16 Implementing `invokeMethod()` in the script base class

```
abstract class RobotBaseScriptClass extends Script {
    @Delegate @Lazy Robot robot = this.binding.robot

    def invokeMethod(String name, args) {         #1
        robot."${name.toLowerCase()}"(*args)      #2
    }
}
```

For our direction constants, we will use a custom binding class, overriding the `getVariable(String name)` method so that we handle our own logic for retrieving variables from the binding, as displayed in Listing 17.

Listing 18.17 A custom binding overriding `getVariables(String)`

```
class CustomBinding extends Binding {
    private Map variables

    CustomBinding(Map vars) {
        this.variables = [
            *:vars,                                                #1
            *:Direction.values().collectEntries { [(it.name()): it] }   #1
        ]
    }

    def getVariable(String name) {
        variables[name.toLowerCase()]                              #2
    }
}
```

#1 We merge the variables passed to the constructor, as well as the Direction constants as mis-cased constants are not taken into account by the static import injection
#2 We search from the variables map an entry with a key in lowercase format

The techniques we have used so far were centered more around integration aspects. They allowed us to properly integrate our business rules using our Domain-Specific Language, and cleanly separate them from our domain model and from the infrastructural code, for a better design, easier maintenance, and overall better readability, without the usual boilerplates of imports and too much punctuation.

But there are many more techniques that we will discover in the next sections to add more flesh to our examples.

18.3 Adding properties to numbers

Now imagine you want to be a bit more precise in your robot movements and wish to specify some notion of distance. How would you go about telling your rover to move right by two meters? Well, we need to update our `move()` method to support a direction and a distance, and we need to implement that concept of distance.

For the latter, we can represent it as a combination of an amount (ex: 2, a number) and a unit (ex: meters, in the form of an enum), as proposes Listing 18.

Listing 18.18 Distance and unit concepts

```
package projectmars.model

import groovy.transform.TupleConstructor

enum Unit {
    centimeter ('cm', 0.01),
    meter      ( 'm',    1),
    kilometer  ('km', 1000)

    String abbreviation
    double multiplier

    Unit(String abbr, double mult) {
        this.abbreviation = abbr
        this.multiplier = mult
    }

    String toString() { abbreviation }
}

@TupleConstructor
class Distance {
    double amount
    Unit unit

    String toString() { "$amount $unit" }
}
```

We update our `Robot` class to support a movement with a direction and a distance by simply adding an overloaded `move()` method like this:

```
void move(Direction dir, Distance d) {
     println "robot moved $dir by $d"
}
```

In our DSL script, we can now call the new method with:

```
move right, new Distance(3, Unit.meters)
```

Is it satisfying? We are back with a feeling of “programming” — as if it were ugly — rather than giving plain English commands to our robot. What could we do to make things a bit better? We could inject the `Unit` constants with another start static import injection, to have something like the following:

```
move right, new Distance(3, meters)
```

We still have the `new` keyword, and the class name `Distance`, which are perhaps slightly redundant when a robot operator sees 3 and meters, he knows it is a distance. There is an increasing trend in the Java ecosystem towards “fluent APIs”, that is, APIs that “read” well, closer to a spoken language. With a factory method on `Distance`, we could turn our order into:

```
move right, Distance.of(3, meters)
```

This is already more pleasing to the eye, even if the overall command doesn’t sound yet like an English sentence. But what I suggest now is that we support the following syntax in our DSL:

```
move right, 3.meters
```

This notation is much more concise, reads well, and is closer to plain English. Later on, we'll also provide the shorcut notation of `3.m` in the international measures format. But how can we add properties to numbers in Groovy? There are a couple of approaches, actually. As we have discovered in Chapter 9 on the Groovy Meta-Object Protocol, we can modify the meta class or use a Category to add new methods and properties to any type, including to number types.

We could simply add the meters property to the meta class of Number with a statement like:

```
Number.metaClass.getMeters = { new Distance(delegate, Unit.meters) }
```

But modifying the meta class has a drawback: its reach is pretty much global to our JVM, and that means we would pollute the namespace of numbers even outside the reach of our DSL. We would also need a place to register that method: we can obviously do that in our integration script as usual. As an alternative, I would like to explore with you the idea of using a Category. Categories have the nice aspect of providing more fine-grained control over the scope of the "monkey patching" we are doing on numbers: changes to the affected classes are only available under the scope of the `use()` method, and in the current thread exclusively. As soon as you leave the block, the changes disappear.

Listing 19 shows the implementation of our Category.

Listing 18.19 Implementation of a distance category

```
class DistanceCategory {
    static Distance getCentimeters(Number num) {
        new Distance(num, Unit.centimeter)
    }

    static Distance getMeters(Number num) {
        new Distance(num, Unit.meter)
    }

    static Distance getKilometers(Number num) {
        new Distance(num, Unit.kilometer)
    }
}
```

Given our distance Category implementation, we need to apply that Category to the execution of our DSL script. To do that, we simply wrap the evaluation of the script with a `use()` block as follows:

```
use(DistanceCategory) {
    shell.evaluate '''
        move left
        move right, 3.meters
    '''
}
```

Now we are talking! Or actually, that is our DSL that is speaking for itself. Perhaps we could also let him express himself with some more formalism by allowing the following form:

```
move right, by: 3.meters
```

Notice that our method now takes a normal argument as well as a named argument. Mixing named and non-named argument is also an interesting technique for making our Domain-Specific Languages more fluent. In the next section, we will have a look at leveraging this approach.

18.4 Leveraging named-arguments

Groovy's support for named arguments in method calls helps clarify the meaning of the parameters that are given to a method, instead of relying purely on the position of those parameters.

When a method call is made using a mix of named and non-named arguments, Groovy follows a convention for the signature of the method to call. All named arguments are put in a map, which is passed as the first argument of the call, and all the other non-named arguments are passed afterwards, in the order with which they appear in the call. More concretely, given an hypothetical call like:

```
method argOne, keyOne: valueOne, argTwo, keyTwo: valueTwo, argThree
```

Groovy's runtime will interpret the call as:

```
method([keyOne: valueOne, keyTwo: valueTwo], argOne, argTwo, argThree)
```

And will in the end call the method with the signature:

```
def method(Map m, argOne, argTwo, argThree)
```

With that new knowledge in our bag of tricks, let's see how we can apply that to our `Robot` class by adding a new `move()` method variant with that approach:

```
void move(Map m, Direction dir) {
    println "robot moved $dir by $m.by"
}
```

Very straightforward. All the named arguments (here, only the `by` argument) are passed in the first Map parameter of the method, and all the non-named arguments (here, only the direction argument) are passed in the order with which they appear in the call, after the Map.

To go further with named arguments, we could imagine defining a speed of movement, to support a syntax like:

```
move right, by: 3.meters, at: 5.km/h
```

Our `move()` method can cope with an additional named argument, which will go into the Map parameter. We just need to adapt our dummy println statement for now:

```
void move(Map m, Direction dir) {
    println "robot moved $dir by $m.by at ${m.at ?: '1 km/h'}"
}
```

If no particular speed is provided, we assume the default speed is simply one kilometer per hour, by using the Elvis operator.

There are two things, which are not yet going to work: first of all, we haven't defined abbreviations in our distance category to support `m`, `km`, etc. Listing 20 shows an updated distance category. Secondly, we have actually a notion of speed here, which is a distance divided by a duration. But we have neither the notion of speed nor of duration.

Listing 18.20 New unit shortcuts for distances

```
static Distance getCm(Number num) { getCentimeters(num) }
static Distance getM (Number num) { getMeters(num)      }
static Distance getKm(Number num) { getKilometers(num)  }
```

Let's start with the notion of speed with the `Speed` class of Listing 21, where we assume speeds are always "per hour".

Listing 18.21 Speed class

```
@TupleConstructor
class Speed {
    double amount
    Unit unit

    String toString() { "$amount $unit/h" }
}
```

Now we need to figure out how to construct our speed objects from our DSL. You noticed that we used the divide operator `/`, and an `h` constant. That means we will need two things: operator overloading (seen in Chapter 3) to be able to call the `div()` method on distances, and a new constant in the binding of the script to provide the `h` hour unit.

This time, we'll start by the second point: we are going to introduce a duration concept, and inject the hour constant in the binding. For that we need a `Duration` enum as displayed in Listing 22.

Listing 18.22 Duration enum

```
enum Duration {
    hour
}
```

And we can inject the hour constant manually in the binding, since we really only care for that specific constant of time, with:

```
def binding = new Binding([
    robot: new Robot(),
    h: Duration.hour
])
```

When writing `5.km/h`, we have the shortcut equivalent of `5.getKm().div(h)`. The `getKm()` method comes from the distance Category. And we need to amend the `Distance` class to support the division as shown in Listing 23.

Listing 18.23 Updated distance with an overloaded division operator

```
@TupleConstructor
class Distance {
    double amount
    Unit unit

    Speed div(Duration dur) {
        new Speed(amount, unit)
    }

    String toString() { "$amount$unit" }
}
```

Supporting the familiar notation of speed like `5.km/h` involved three techniques at the same time: adding properties to number, operator overloading, and binding constant injection. Sometimes, for more concise elements in your Domain-Specific Languages, you will need to combine several techniques at the same time, to achieve your goals of readability and expressivity.

Our more complex command now looks like this:

```
move right, by: 3.m, at: 5.km/h
```

It is very readable, like plain English when we read that sentence aloud, but visually, there may be something that could be annoying: the quantity of punctuation that is needed to separate the various elements of that sentence. What if we could get rid of commas and colons, for instance? Thanks to Groovy's *command chains*, we can go as far as writing this kind of statements:

```
move right by 3.m at 5.km/h
```

Except the dots between the numbers and units, we got rid of the commas and colons. In the following section, we're going to have a look at these command chains, so we learn how to construct them, for the benefit of our Domain-Specific Languages.

18.5 Command chains

We started our journey about Domain-Specific Languages by speaking briefly about Groovy's flexible nature, and particularly the fact we can drop parentheses (and semi-colons) for top-level method calls. A standalone call to a method that takes some arguments can be written that way, making our `println`'s a bit nicer on the eye. These kinds of calls are called *command expressions* or *top-level statements*, as they look more like commands or orders than like usual Java programming. Command expressions can as well take named-arguments, or a mix of named and non-named arguments, as we saw in our examples. When the Groovy developers designed that specific aspect of the syntax, they always felt that at some point, we could probably go beyond just top-level statements, and find a way to expand those simple command expressions, into more complex constructs, where we would chain or nest method calls with that parentheses-free syntax. It took several years before a proposal emerged, providing an approach that sounded good enough, with a good dose of groove, and that would help developers write even more beautiful DSLs.

Let's step back a little and analyze a simple command expression. What is it? A method name (the method to call), some whitespace, and a comma-separated list of named and non-named arguments:

```
methodName argOne
methodName argOne, argTwo
methodName argOne, keyOne: valueOne, argTwo, keyTwo: valueTwo
```

We always have those two parts: method name, and arguments.

Now, what if we expanded that concept to chained method calls, what could the syntax look like?

```
methodOne argOne methodTwo argTwo
methodOne argOne methodTwo argTwo methodThree argThree
methodOne argOne, keyOne: valueOne methodTwo argTwo, keyTwo: valueTwo
```

And how would they be interpreted? As chained method calls with the usual syntax:

```
methodOne(argOne).methodTwo(argTwo)
methodOne(argOne).methodTwo(argTwo).methodThree(argThree)
methodOne(argOne, keyOne: valueOne).methodTwo(argTwo, keyTwo: valueTwo)
```

Notice the alternation and repetition of a method name and arguments (named and non-named). This is the essence of the command chains expressions introduced in Groovy 1.8.

So, a syntax like the one we want to achieve:

```
move right by 3.meters at 5.km/h
```

Is equivalent to:

```
move(right).by(3.meters).at(5.km/h)
```

Once we have mentally managed to parse that new syntax in our head with the right added parentheses and dots, the implementation becomes trivial. We change the implementation of the `Robot` class to have a `move()` method that takes a `Direction`, which returns some object (instead of void currently), on which we can call a method named `by()` that takes a `Distance` and returns yet another object (or the same!) that then has a method called `at()` which takes a `Speed` as argument.

This pattern is often used in Java fluent APIs, where we can chain method calls on the same object, as all the methods in the chain actually return `this`, the current object on which we operate. We could follow this approach of returning `this`, but instead, I will show you a nice trick with nested maps and closures, which allows us to avoid having to create a more complex fluent API approach à la Java. Here's what our new `move()` method can look like in Listing 24.

Listing 18.24 Chained method calls with nested maps and closures

```
def move(Direction dir) {
    [by: { Distance dist ->
        [at: { Speed speed ->
            println "robot moved $dir by $dist at $speed"
        }]
    }]
}
```

Let's decompose that implementation. First, our method is called: `move(right)`. This calls returns a map, whose sole key is `by`. From that map, you get the value associated with the `by` key, that's a closure, that you call with the `3.meters` distance as parameter, which in turn returns a new map with the `at` key, which corresponds to a last closure that we call passing it the `5.km/h` speed argument. This sequence of call is decomposed as shows Listing 25.

Listing 18.25 Decomposition of an extended command expression call sequence

```
def map1 = move(right)
def byClosure = map1['by']
def map2 = byClosure(3.meters)
def atClosure = map2['at']
atClosure(5.km/h)
```

Obviously, you will prefer the abbreviated version than the expanded one! At least I hope so. Command chain expressions will allow you to write more English-friendly sentences, with the minimum amount of clutter.

We discovered the pattern of the sequence of method name and arguments, with an even number of elements (ie. always a method and some arguments), but it's also possible to have an odd number, with a series of method name and arguments, and a final property access. Let's have a look at that with a concrete example: our robot moves, but it has also got arms to examine the rocky soil, so we can tell it to deploy his left or right arm:

```
deploy left arm
```

Now we have a "sentence" with three words. It doesn't fit our pattern of method name / arguments anymore. But when faced with an odd number of "elements", command chains have their own tricks! The last element is a property access. So the above is equivalent to:

```
deploy(left).arm
```

If you want to implement that chain of calls, you can apply our technique with nested maps and closures, but with a little twist, as the last element is not a method call:

```
def deploy(Direction dir) {
    [arm: {-> println "deploy $dir arm" }()]
}
```

Notice here particularly how the `arm` property from the map is associated with a closure call — see the parentheses after the closure definition. Otherwise, the `.arm` part of the expression would just return the closure, without executing it.

Similarly with odd number of words, we can use some "silent words" used as parameters of chained method calls, and use maps with default values in the place of method names as shows Listing 26.

Listing 18.26 An order DSL

```
def of = "silent word"                          #1
def buy(n) {
    [shares: { of ->                                   #2
        [:].withDefault { ticker ->                    #3
            println "buy $n shares of $ticker"
        }
    }]
}

buy 200 shares of GOOG
```

In #1, we defined a dummy variable called `of`, to which we assign a random value (it could as well be null). This variable is passed as parameter of the closure value of the `shares` key of the map in #2, but we don't use that parameter at all through the rest of the implementation. Then in #3, since we may have an infinite set of stock tickers (unlike in the case of the Mars rover with only a finite set of instruments like its arm), we use a map with default values, thanks to the Groovy development kit method `withDefault()`. Each time we try to access a key that wasn't in the original empty map on which that method is called, we execute the closure (which simply prints a message here).

With this approach of command chains, we are able to tackle DSL sentences that are more varied in style, and can correspond to proper English. But Domain-Specific Languages are not just about the ability to write English phrases, and we can go beyond some imperative commands and add various form of control flow to our mini-languages. Of course, it is possible to use all the control flow logic from Groovy in your DSLs (if/else, for loops, while loops, etc.), but in the next section, we will also discover how to create your own control structures.

18.6 Your own control structures

The ubiquituous `if` branching instruction is available in virtually all existing languages. It takes a boolean expression as parameter, and a block of code that is executed when the boolean expression is evaluated to true. For some reason, perhaps your business users are more comfortable with "when" than "if"? Could we have a control structure like our `if`, but with a `when` as keyword? What would we need to achieve that goal? A method taking a boolean expression and a closure to represent our code block to execute, as Listing 27 shows.

Listing 18.27 An alternative to if

```
def when(boolean condition, Closure block) {
    if (condition) block()
}

def a = 1
def b = 2

when(a < b, { println "a < b" })          #1
```

Our `when()` method does take a boolean expression as first parameter and a closure as second parameter on #1, but this doesn't look quite like our `if` statement yet. So what's missing? You guessed it, we remember about this syntactical rule which allows us to put the last closure argument "outside" of the parentheses — like `inject(seed){}`, etc. By following this rule, we can rewrite our control structure as:

```
when(a < b) { println "a < b" }
```

This is exactly what we wanted to achieve, we now have created a synonym of `if`. You can also imagine implementing an `unless` method that would be like `if`, but when negating the boolean expression: `unless (condition) {}`.

The astute reader might however notice one thing: since that's a closure we pass as last argument, the curly braces are always needed even when the closure only contains one instruction. Some will say it's a nice way of enforcing the good practice of always requiring curly braces, but others could think it's a lack of flexibility.

There are some cases like this one, where mere method calls with the nice Groovy syntax tweaks still differ in a way or another, but we can find workarounds, for example by using AST transformations, to alter the structure of our programs to reach our syntactical goals, but as a developer, you should also remember that it can come at a certain price. If you really want to have your `when` statement to support single block statements without curly braces, you would have to implement an AST transformation. That means that you would have more code to develop, test and maintain, and it will take you more time. As a developer, and as the guiding hand of the syntax of the DSL, you might have to make some compromises: do you want (or do your users want) as much flexibility of syntax as possible but at the expense of more time spent implementing the feature and more code to develop and maintain, or are they happy with a little sacrifice (like requiring curly braces) but have their DSL implemented more rapidly and easier to develop and maintain for the developers?

You will have to keep those considerations in mind when crafting your DSL. Sometimes, as developers, we tend to over-engineer code because we think in the end it'll please the end users, but we forget the agile mantra of "YAGNI" (You Ain't Gonna Need It), as those end users don't necessarily need that added flexibility of the language you're creating for them.

After this little warning stance, you may still want to know the solution on how to get rid of curly braces? I know you would want to know how to do that, damn engineer that you are! Before diving in, let me show you another example where curly braces might be good to remove, as the behavior of the code might be surprising otherwise, and where mandating the curly braces make the statement look weirder than it should.

Let me introduce you to the `till` construct! Just like `while`, `till` is a looping construct. The sole difference is that instead of looping while a condition is true, we will loop till the condition's evaluation becomes true. The problem here is that the condition should be evaluated each time we iterate and call the block of code, to see if we must still continue to iterate or stop. If we tried the naïve approach of Listing 28 we would get an infinite loop, as the condition is evaluated once as false, and the condition is not re-evaluated later on.

Listing 18.28 Erroneous implementation of the till construct

```
def till(boolean condition, Closure closure) {
    while(!condition) closure()           #2
}

def counter = 0

till(counter == 10) {                     #1
    counter++
}
```

On #1, when the call to the `till()` method is made, the boolean expression is evaluated only once, at that specific moment. It's not re-evaluated each time. The consequence is that our `while` loop in the implementation will loop forever, which is definitely not the outcome we wished for.

How can we have an expression that is re-evaluated each time? By using a closure, as Listing 29 explains.

Listing 18.29 Implementation of `till()` using a closure condition

```
def till(Closure condition, Closure block) {
    while(!condition()) block()                  #1
}

def counter = 0

till({counter == 10}) {                          #2
    counter++
}

assert counter == 10
```

In the implementation, on #1, we now evaluate the negation of the result of the boolean expression contained in the closure condition, but on #2, we noticed that the `till()` usage has become somewhat less appealing as we require curly braces for the condition, making the look different from our goal to create a construct like `while()`. Obviously, like in our `when` case, we wouldn't mind also being able to get rid of the curly braces for the block of code as well (the one increasing the counter).

We have already come up with three cases where it could be handy if we could treat a simple statement or expression as if it were a closure, by managing to find a solution where we could abandon the surrounding curly braces.

In the case of the `when` statement, we would have to transform:

```
when (condition);
statement;
```

Into:

```
when (condition) {
    statement
}
```

Whereas in the case of `till`, we would like to transform:

```
till ({ condition }) {
    statement
}
```

Into the following:

```
till (condition) {
    statement
}
```

Or if we wanted to get rid the curly braces of the single statement case, that would mean transforming:

```
till (condition);
statement;
```

Into this form:

```
till ({ condition }) {
    statement
}
```

Have you fastened your seat belt? Okay, let's have some fun transforming some nodes of the Abstract Syntax Tree that the Groovy parser creates!

In the following paragraphs, we'll focus on one particular transformation: allowing curly-braces-free when calls. We'll let you have fun with implementing all the cases — otherwise we would have to kill a couple more trees for producing the book with the increased number of pages.

In the AST Transformation chapter, you learned about the two kinds of transformations Groovy supports: local transformations and global transformations. Global transformations, in our case, are interesting because they are applied everywhere without the need of annotating elements of our business code with annotations, but that's also the drawback of the global application of the transformation, so everywhere we might have a `till` or `when`, the transformation would kick in. On the other hand, although local transformations exhibit annotations that may be foreign to business users' eyes, they have the advantage that the transformations are really just local.

For the purpose of this example, we will actually use local transformations, but as we don't wont to impose on our business users to have to use an explicit annotation, we will also learn how to *hide* the annotation, by injecting the local transformation transparently, thanks to compilation customizers. We will get the benefits of both kinds of transformations without their drawbacks: locality and transparency of application.

First of all, we will start by defining an annotation for our local transformation. Listing 30 shows our annotation definition.

Listing 18.30 `CustomControlStructure` annotation

```
import java.lang.annotation.*
import org.codehaus.groovy.transform.*

@Retention(RetentionPolicy.SOURCE)                              #1
@Target(ElementType.TYPE)                                       #2
@GroovyASTTransformationClass(classes = [WhenTillTransform])    #3
@interface CustomControlStructure {}
```

Our annotation doesn't have to be available at runtime through reflection, so the source retention policy, on #1, will be sufficient for our needs. The annotation will be put on types (ie. classes), on #2, and we will see how we are going to inject that annotation on the base script of our business rules, in a short moment. And on #3, we instruct the compiler that the transformation is implemented by the transform class called `WhenTillTransform` that we have still to implement.

Let us build an empty shell for our transformation, in Listing 31, where we fill in the gaps, as we will progress on our journey.

Listing 18.31 The `WhenTillTransform` class

```
import org.codehaus.groovy.ast.*
import org.codehaus.groovy.ast.expr.*
import org.codehaus.groovy.ast.stmt.*
import org.codehaus.groovy.control.*
import org.codehaus.groovy.transform.*
import org.codehaus.groovy.tools.ast.*

@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)
class WhenTillTransform implements ASTTransformation {
    void visit(ASTNode[] nodes, SourceUnit unit) {
        // we'll fill in the gaps!
    }
}
```

To take care of our new control structure, we use a technique we have already used, with a base script class implementing our special `when()` method, taking a boolean and a closure as parameters, as shown in Listing 32.

Listing 18.32 A base script class for our business logic's control structure

```
abstract class BusinessLogicScript extends Script {
    def when(boolean condition, Closure block) {
        if (condition) block()
    }
}
```

As we flesh out our overall solution, we need a little infrastructure to test our transformation with Listing 33.

Listing 18.33 Testing our transformation

```
def binding = new Binding([customer: [name: 'John Doe', age: 32]])      #1

def config = new CompilerConfiguration()                                #2
config.scriptBaseClass = BusinessLogicScript.class.name                 #3
config.addCompilationCustomizers(                                       #4
    new ASTTransformationCustomizer(CustomControlStructure))            #4
```

```
def shell = new GroovyShell(this.class.classLoader, binding, config)    #5
def result = shell.evaluate '''                                         #6
    when (customer.age >= 21) {                                         #6
        "Alcohol allowed for ${customer.name}"                          #6
    }                                                                   #6
'''
assert result == "Alcohol allowed for John Doe"                         #7
```

In #1, we are going to define a binding containing the data on which our business rule will work on. We inject a variable called customer, corresponding to a simple map (but of course, you can use plain classes, numbers, whatever you want). In #2, we instantiate a `CompilerConfiguration` object that we will use to define a base script class, in #3, for our business logic (that we just defined) and a compiler customizer that will inject our local AST transformation, in #4. `GroovyShell` will be our weapon of choice for evaluating our business rules using our new control structure. In #5, when instantiating the shell, we are passing the current classloader of the script as parameter, as I scaffold this example in the Groovy console, and I have actually put everything (annotation and transformation) in the same compilation unit (ie. in the same script), the shell needs to know all the classes that we're working on. We are also passing the binding for sharing the information that our business rules need, and the compiler configuration object. The business rule using our custom control structure in #6 is using the `evaluate()` method of the shell. We can then check the result returned by the evaluation of the business rule in #7.

So far so good, but what happens if we remove the curly braces, to implement our *no-curlies* requirement? We get the following exception:

```
groovy.lang.MissingMethodException: No signature of method: BusinessLogic.when() is
applicable for argument types: (java.lang.Boolean) values: [false]
Possible solutions: when(boolean, groovy.lang.Closure), wait(), run(), run(), grep(),
wait(long)
     at BusinessLogic.run(Script1.groovy:9)
```

What's happening here? Well, as we hinted before, Groovy thinks the `when` method call takes only a boolean argument, and treats the supposed body of the `when` as another statement, not part of the `when` call. It treats that code as if it were written as follows (semi-colons helps better visualize what Groovy understands here):

```
when (customer.age >= 21);
"Alcohol allowed for ${customer.name}";
```

So the goal of our transformation will be to analyze this AST to recognize the `when` calls, to wrap the following standalone statement within a closure expression, and to pass that expression as a second parameter of the `when` calls, while removing that free-standing statement from the code block it belongs to.

Let's put than plan into action, by filling the gaps of our transformation's `visit()` method, by creating our own implementation of `ClassCodeVisitorSupport`, in Listing 34.

Listing 18.34 Visiting code with `ClassCodeVisitorSupport`

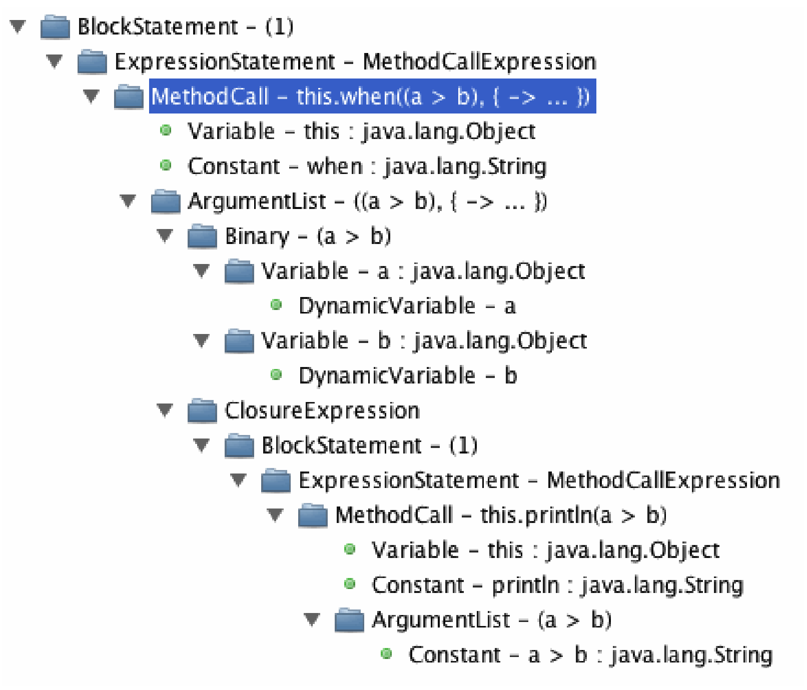
```
@GroovyASTTransformation(phase = CompilePhase.CONVERSION)
class WhenTillTransform implements ASTTransformation {
    void visit(ASTNode[] nodes, SourceUnit unit) {
        ClassNode annotatedClass = nodes[1]
        new ClassCodeVisitorSupport() {
            def currentMethod
            def currentBlock
            def currentStatement

            void visitMethod(MethodNode method) {
                currentMethod = method
                super.visitMethod(method)
            }

            void visitBlockStatement(BlockStatement block) {
                currentBlock = block
                super.visitBlockStatement(block)
            }

            void visitExpressionStatement(ExpressionStatement statement) {
                currentStatement = statement
                super.visitExpressionStatement(statement)
            }
            void visitMethodCallExpression(MethodCallExpression mCall) {
                super.visitMethodCallExpression(mCall)
            }

            protected SourceUnit getSourceUnit() { unit }
        }.visitClass(annotatedClass)
    }
}
```

ClassCodeVisitorSupport is going to be very handy for us, for "visiting" the data structure that is the Abstract Syntax Tree. It implements the famous visitor pattern, calling many visit* methods, as it encounters a particular node in the AST.

In our situation, we are interested in four particular methods: visitMethod(), visitBlockStatement(), visitExpressionStatement(), and visitMethodCallExpression().

To illustrate why those four methods are the ones we are going to pay attention to, let's have a quick look at the structure of the AST on for the following when instruction:

```
when (a > b) {
    println "a > b"
}
```



```
▼ BlockStatement - (1)
    ▼ ExpressionStatement - MethodCallExpression
        ▼ MethodCall - this.when((a > b), { -> ... })
            ● Variable - this : java.lang.Object
            ● Constant - when : java.lang.String
            ▼ ArgumentList - ((a > b), { -> ... })
                ▼ Binary - (a > b)
                    ▼ Variable - a : java.lang.Object
                        ● DynamicVariable - a
                    ▼ Variable - b : java.lang.Object
                        ● DynamicVariable - b
                ▼ ClosureExpression
                    ▼ BlockStatement - (1)
                        ▼ ExpressionStatement - MethodCallExpression
                            ▼ MethodCall - this.println(a > b)
                                ● Variable - this : java.lang.Object
                                ● Constant - println : java.lang.String
                                ▼ ArgumentList - (a > b)
                                    ● Constant - a > b : java.lang.String
```

Figure 18.1 Structure of a when() call

The "when" MethodCallExpression is wrapped in an ExpressionStatement, which is an element of the list of statements of the BlockStatement, that is in turn child of a MethodNode (that we don't see in this screenshot of the Groovy console AST browser). We'll track this structure by implementing the four adequate visitor methods.

Please note the fact we are always calling the super methods of the same name, as the parent class of our anonymous inner class knows all the traversing logic, and you don't have to take care of that logic yourself, that's that super call that will handle that for you.

Let's focus on the `visitMethodCall()` implementation now, as the other methods are really here only to track where we are in the AST, and have the right pointers for the places where the modifications of the AST will happen. Listing 35 shows how we can find the relevant `when()` calls we want to look at.

Listing 18.35 Spotting the when call

```
void visitMethodCallExpression(MethodCallExpression mCall) {
    if (
        mCall.objectExpression instanceof VariableExpression &&  #2
        mCall.objectExpression.variable == 'this' &&             #1
        mCall.method instanceof ConstantExpression &&            #3
        mCall.method.value == 'when' &&                          #4
        mCall.arguments.expressions.size() == 1                  #5
    ) {}
    super.visitMethodCallExpression(mCall)
}
```

Such `when()` calls are actually method calls on the `this` (#1) variable expression (#2), and the name of the method being called is a constant expression (#3) with value when (#4). We also differentiate calls when the normal boolean / closure pair is used as arguments, or when we trick the compiler into believing a `when(boolean)` call followed by a plain statement is actually another form of our control structure — here we don't want to modify the boolean / closure call at all, only the latter one.

We have found the calls we want to act upon, now what's next? Firstly, we should check that there actually is another statement after the when call, that will be our single-statement when body. Otherwise, if there were no following statement, that would mean the when instruction is not complete, and that will be spelled as a compilation error. But how do we know there is no following statement?

As Listing 36 demonstrates, we'll have a look at the list of statements contained in the block statement, find the index of the expression statement wrapping the method call (#1), to check if its index is the last one of the block #2). If the index is the last one, that means the associated when statement is missing, and we are instructing the compiler that a compilation error should occur.

Listing 18.36 Checking the index of the when call (snippet)

```
def idx = currentBlock.statements.findIndexOf {                  #1
    it == currentStatement                                       #1
}                                                                #1
if (idx + 1 >= currentBlock.statements.size()) {                 #2
    addError("The when instruction has no body.", mCall)         #3
} else { /* ... */ }
```

As usual, the Groovy development kit offers useful methods that we can take advantage of, for example, in #1, the `findIndexOf()` method allows us to find the index of the statement wrapping the when call, in the list of statements of the current block of code. As you realize, the other visitor methods have helped us tracking where we were (current code block and current statement). With the knowledge of the position of the statement in the block, we verify in #2 that the call is not the last element of the block, as otherwise, that means no following statement can be attached to our when call. If this verification fails in #3, we add a compilation error message that the compiler will throw at you! You can check that the error is thrown by commenting the last statement of the business rule. Finally, in the `else` part, we are going to continue our implementation, with the transformation per se.

So what's left to do? We want to find the statement associated with our when call, wrap it in a closure, and modify the when call to take that closure as second parameter.

We know the index of the when call, so the statement to be attached to the when call is the one following when:

```
def whenCode = currentBlock.statements[idx + 1]
```

We wrap that code within a `ClosureExpression`, whose constructor takes two arguments: an array of parameters that the closure can receive, and the statements forming the body of the closure:

```
def closureExp = new ClosureExpression(Parameter.EMPTY_ARRAY, whenCode)
```

With that closure expression we just created, we can push it as a second parameter of our when call with:

```
mCall.arguments.addExpression(closureExp)
```

The statement that we wrapped in a closure is still present in the list of statements of the block of code containing our when call, so we need, as a last step, to remove it from the list of statements with:

```
currentBlock.statements.remove(idx + 1)
```

Now, if you run the business rule again, you will notice that the new syntax is allowed: a when call and a curly-braces free single statement. Our final transformation class looks like Listing 37.

Listing 18.37 The complete AST transformation to allow brace-free 'when' statements

```
@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)
class WhenTillTransform implements ASTTransformation {
    void visit(ASTNode[] nodes, SourceUnit unit) {
        ClassNode annotatedClass = nodes[1]
        new ClassCodeVisitorSupport() {
            def currentMethod
            def currentBlock
            def currentStatement
            void visitMethod(MethodNode method) {
                currentMethod = method
                super.visitMethod(method)
            }
            void visitBlockStatement(BlockStatement block) {
                currentBlock = block
                super.visitBlockStatement(block)
            }
            void visitExpressionStatement(ExpressionStatement statement) {
                currentStatement = statement
                super.visitExpressionStatement(statement)
            }
            void visitMethodCallExpression(MethodCallExpression mCall) {
                if (
                    mCall.objectExpression instanceof VariableExpression &&
                    mCall.objectExpression.variable == 'this' &&
                    mCall.method instanceof ConstantExpression &&
                    mCall.method.value == 'when' &&
                    mCall.arguments.expressions.size() == 1
                ) {
                    def idx = currentBlock.statements.findIndexOf {
                        it == currentStatement
                    }
                    if (idx + 1 >= currentBlock.statements.size()) {
                        addError(
                            "The when instruction has no body.", mCall)
                    } else {
                        def whenCode = currentBlock.statements[idx + 1]
                        def closureExp = new ClosureExpression(
                            Parameter.EMPTY_ARRAY, whenCode)
                        closureExp.variableScope = new VariableScope()
                        closureExp.variableScope.parent =
                            currentBlock.variableScope
                        mCall.arguments.addExpression(closureExp)
                        currentBlock.statements.remove(idx + 1)
                    }
                }
                super.visitMethodCallExpression(mCall)
            }
            protected SourceUnit getSourceUnit() { unit }
        }.visitClass(annotatedClass)
    }
}
```

Such a transformation is not necessarily a big amount of code, but the explanations usually take longer than what is really takes in lines of code. The hardest part I would say, however, is to get to know better the internals of the Groovy compiler and its machinery, and how the AST is structured.

We solved one of the cases we listed, and I'll let as an exercise to the reader to continue fleshing out this transformation to cover the other cases.

Before closing this section on custom control structures, let's have a look at a last one, where we will be able to take advantage of command chains. A lot of testing frameworks these days are following the "Behavior Driven Development" approach, adopting the vocabulary of "user stories": given / when / then. For example, let's consider the scenario:

Given two numbers, a and b whose values are 1 and 2
When you add a and b together
Then the result of the addition is 3

We can interpret that scenario with the custom control structure shown below. It's a similar construct as `if` / `else`:

```
given {
    a = 1
    b = 2
} when {
    result = a + b
} then {
    result == 3
}
```

Such a structure is actually a chained method call structure equivalent to:

```
given({ … }}.when({ … }).then({ … })
```

We will apply the technique will learned with nested maps and closures:

```
def given(Closure g) {
    g()
    [when: { Closure w ->
        w()
        [then: { Closure t ->
             t()
        }]
    }]
}
```

The three closures are called and executed serially.

In the context of a script, when you assign values to variables that haven't been defined, then the script binding is used to store those values. So our example works fine inside a script. But if you'd run this inside a class, you'd get the following error message:

```
groovy.lang.MissingPropertyException: No such property: a for class: Test
```

To make it work for classes as well, you should use some kind of value holder object, for example a map, which should be used as delegate of all the closures. And to have the assignments be done on that value holder object, you should also set the resolve strategy of closures to use the delegate first, as otherwise, the containing class would be used, as shown in Listing 38:

Listing 18.38 given / when / then with closure delegation

```
def given(Closure g) {
    def valueHolder = [:]
    g.delegate = valueHolder
    g.resolveStrategy = Closure.DELEGATE_FIRST
    g()
    [when: { Closure w ->
        w.delegate = valueHolder
        w.resolveStrategy = Closure.DELEGATE_FIRST
        w()
        [then: { Closure t ->
            t.delegate = valueHolder
            t.resolveStrategy = Closure.DELEGATE_FIRST
            t()
        }]
    }]
}
```

Taking advantage of the way properties are resolved in the context of a closure is often used in Domain-Specific Languages, for builders for example, for switching the context of execution. In the next section, it's worth investigating a bit more what you can do with this technique, and particularly, how you can use the Groovy development kit's `with{}` method.

18.7 Context switching with closures

When you use POJOs (Plain Old Java Objects) or POGOs (Plain Old Groovy Objects) with lots of properties or when you assign values to many of those properties, your code can become quite verbose.

Considering an address bean like this:

```
class Address {
    String line1
    String line2
    String city
    String zipCode
    String country
}
```

If you instantiate such a class the "Java-way", you'd get the following code:

```
def addr = new Address()
addr.line1 = '1st, Main Street'
addr.line2 = 'Suite 345'
addr.city = 'Metropolis'
addr.zipCode = '12345'
```

The verbosity shows itself with the repetition of the `addr.` prefix. Constructors with named arguments improves the situation:

```
addr = new Address(
    line1: '1st, Main Street',
        line2: 'Suite 345',
        city: 'Metropolis',
        zipCode: '12345'
    )
```

But Groovy also adopts the context switching technique with closure delegation in the form of the `with{}` method:

```
    addr = new Address()
    addr.with {
        line1 = '1st, Main Street'
        line2 = 'Suite 345'
        city = 'Metropolis'
        zipCode = '12345'
    }
```

All the assignments are done on the properties of the object. They are not done on some fields or local variables.

If we come back to our examples with our Mars rover, a simple context switching might have been good enough for our users, as demonstrated by Listing 39. Furthermore, this example also proves this context switching works as well with method calls.

Listing 18.39 Robot and context switching with closures

```
    def robot = new Robot()

    robot.with {                 #1
        move left                #2
        move forward             #2
    }
```

In #1, we use again the `with{}` method with our robot instance, whereas in #2 we can give it its orders with the streamlined syntax.

One little downside though, is that perhaps the usage of "with" as a method name doesn't look ideal within the context of the rover, but we can very well alias this method, simply by adding (or injecting by metaprogramming) a method on `Robot` that would just delegate to Groovy's `with{}` method:

```
    void execute(Closure actions) {
        this.with actions
    }
```

Then you'd be able to send your commands that way:

```
    robot.execute {
        move left
        move forward
    }
```

Let's finish this section with a concrete example of how we can improve the usage of a library, using command chains, and using `with{}` again. In recent years, we have seen many projects using the "fluent API" approach, that we mentioned when we spoke about command chains. To illustrate this, I'll take inspiration from a class called `FetchOptions` from the Google App Engine project SDK, that is used to parameterize how data is fetched from the datastore used for storing non-relational information. This class can be easily replicated by the class presented in Listing 40.

Listing 18.40 A "fluent" API example

```
    final class FetchOptions {                                       #2
        private int limit, offset, chunkSize, prefetchSize
        private FetchOptions() {}                                   #3

        FetchOptions limit(int lim) {
            this.limit = lim
            return this                                             #1
        }
        FetchOptions offset(int offs) {
            this.offset = offs
            return this
        }
        FetchOptions chunkSize(int cs) {
            this.chunkSize = cs
            return this
        }
        FetchOptions prefetchSize(int ps) {
            this.prefetchSize = ps
            return this
        }

        static final class Builder {                                #5
            private Builder() {}                                    #6
            static FetchOptions withDefaults() {                    #4
                new FetchOptions()                                  #4
            }
            static FetchOptions withLimit(int lim) {                #7
                new FetchOptions().limit(lim)
            }
            static FetchOptions withOffset(int offs) {
                new FetchOptions().offset(offs)
            }
            static FetchOptions withChunkSize(int cs) {
                new FetchOptions().chunkSize(cs)
            }
            static FetchOptions withPrefetchSize(int ps) {
                new FetchOptions().prefetchSize(ps)
            }
        }
    }
```

The `FetchOptions` class an implementation of the classic "Gang of Four" Builder Pattern. You can find various variants with slightly different approaches, but the key aspect and common gene between implementations is usually the fact that you have several methods always returning `this`, the current instance, like in #1. That way, you can chain calls to methods of that instance that you are building, and create sentences that read well, although with a bit too much punctuation that blurs the reading of those sentences.

Looking at #2, you quickly realize that you cannot extend the class at will since the class is final, and you cannot easily instantiate it as the constructor is private, as shown in #3. The sole class allowed to instantiate `FetchOptions` is the internal `Builder` class, in #4, but unfortunately, this class is once again final (#5) and has got a private constructor (#6). It's not very friendly for hacking!

The `Builder` class then gives you various static methods (like in #7) so you can create `FetchOptions` instances, and then chain calls easily on `FetchOptions` once you've got your first instance created.

Let us now use this `FetchOptions` class and its `Builder`. Here is what we can do:

```
def options = FetchOptions.Builder.withLimit(10).offset(60).chunkSize(1000)
```

Having to prefix all options creations with `FetchOptions.Builder` is not necessarily very beautiful. But if you use a static import on the methods of the `Builder`, then the situation improves nicely with:

```
import static FetchOptions.Builder.*

def options = withLimit(10).offset(60).chunkSize(1000)
```

Now, combine that with command chains, and you can remove lots of punctuation noise:

```
def options = withLimit 10 offset 60 chunkSize 1000
```

What happens if ever we have such fluent APIs with literally tons of methods we need to call in a chain? You'll have to split the statement across several lines. Without command chain expressions, you can get away with:

```
def options = withLimit(10)
              .offset(60)
              .chunkSize(1000)
```

However, with command chains, you would have to use a backslash to split over several lines, as otherwise the Groovy compiler might think these are individual method calls (ie. not chained):

```
def options = withLimit 10      \
              offset 60         \
              chunkSize 1000
```

Using backslashes is probably not very intuitive for business users. Furthermore with a static import, we don't necessarily remember that we are creating fetch options, since we just see the various options themselves only.

Right, but what are we trying to achieve here? We want to have a concise and readable way of expressing the creation of fetch options. We'd like to be able define the various options on one line or many, transparently, but not at the expense of odd characters or abandoning command chain expressions, and still visually understand we do want to create fetch options. Fortunately, there's a solution for that, by combining static imports, command chains, and context switching with `with{}`.

Let's create our own fetch options builder utility class that will wrap the usage of `FetchOptions.Builder` in Listing 41.

Listing 18.41 The FetchOptionsBuilder class

```
class FetchOptionsBuilder {
    static FetchOptions fetchOptions(Closure c) {
        def opts = FetchOptions.Builder.withDefaults()
        opts.with c
        return opts
    }
}
```

Our very own `FetchOptionsBuilder` class contains a single static method called `fetchOptions`. We'll be able to static import it too. What's more interesting are the three lines of code from this single method. The first one hides the usage of the long form of the creation of the first fetch options instance. The second one then uses with to delegate all method calls and property accesses from within the body of the closure passed in parameter, so that the calls and access are routed to the `FetchOptions`. And the last one actually returns that `FetchOptions` instance.

Let's see this little cutie in action in Listing 42.

Listing 18.42 The FetchOptionsBuilder in action

```
import static FetchOptionsBuilder.fetchOptions

fetchOptions {
    limit 10 offset 60
    chunkSize 1000
}
```

You can create a FetchOptions instance by importing and calling our newly created utility class and its static method, by passing a closure to that method call, in which you can then define all the options you need, by chaining calls on a single line with command chains, or by stacking them up spanning several lines, or even a combination of both. Furthermore, you can assign the result of that call to variables, or use such calls verbatim as parameters of the datastore commands.

All that without the standard method call syntax of Java, or too much punctuation or weird line continuation characters. We managed to pimp our library with a more Groovy-friendly builder class, to instantiate a complex object.

Speaking of *builders*, we have talked about closure delegation and resolve strategy, a technique used by Groovy's with{} method and by many builders in the wild (like Grails'), and you have also learned a lot about them in the chapter about Groovy builders, on how to use existing ones provided by Groovy or how to create your own. Groovy builders are really great for creating hierarchical data structures. But in the next section, I'll show you another handy trick to create such trees of data.

18.8 Another technique for builders

Hierarchical data is everywhere. When you think of a car, it can be described with the decomposition of its various parts. Or if you look at your folders or your hard drive, once again, the hierarchical nature of the file system shows up. Groovy builders can come to the rescue for implementing a DSL for hierarchical representations (extending BuilderSupport, FactoryBuilderSupport or using ObjectGraphBuilder), but you can also take advantage of the @Newify transformation. But first, a few words about this transformation.

The standard way of instantiating an instance of the class is to use the new keyword that calls one of the constructors of the class. Several languages beyond Groovy use this notation, but others adopt different syntaxes. Ruby prefers a factory-like approach where new is actually a class method, so you can call MyObject.new(). Python, on the other hand, just doesn't use a method or a keyword, it simply appends parentheses and arguments to the name of the class with MyObject(). The @Newify transformation proposes to add those syntaxes to Groovy.

Listing 18.43 Ruby-style instantiation

```
import groovy.transform.*

@ToString
class Car {
    String make
    String model
}

@Newify
def car = Car.new(make: 'Porsche', model: '911')

assert car.toString() == 'Car(Porsche, 911)'
```

To use the Ruby-style approach, you annotate a class, a method, a field, or a local variable with the @Newify annotation, as shown in Listing 43. Then in the scope of application of that annotation, a new static method new() appears on all types so that you can instantiate objects by calling that factory method.

For Python-style notation, you need to use the @Newify annotation with a class or array of classes as parameter, as demonstrated in Listing 44.

Listing 18.44 Python-style instantiation

```
@Canonical
class Country {
    String name
}
```

```
@Canonical
class City {
    String name
    String zipCode
    Country country
}

@Newify(City, Country)
def paris = City('Paris', '75000', Country('France'))

assert paris.toString() == 'City(Paris, 75000, Country(France))'
```

Given the `Country` and `City` classes, and using the `@Canonical` transformation to have a nice `toString()` output and a Java-like tuple constructor, we can then apply the `@Newify` transformation with the classes for which we want to abandon the usage of the `new` keyword altogether.

You already see in that example how we can build something from its parts with a more concise instantiation notation. Let's push that further with another example where we can have some arbitrarily nested structure. As we're in the chapter about DSLs, and the L stands for Language, we're going to build a term expression language to represent some formula.

We need a base interface for representing terms, as well as several implementations of that term: one for a value, one for representing an addition, and another one for a multiplication (and you can add other ones if you so desire). Listing 45 gives you those interfaces and classes to get started building your term structures.

Listing 18.45 Term, Value, Add and Mult types

```
import groovy.transform.*

interface Term {}

@Canonical
class Value implements Term {
    def content
}

@Canonical
class Add implements Term {
    def left, right
}

@Canonical
class Mult implements Term {
    def left, right
}
```

With a Java-like instantiation, to represent the expression a * (b + c), you'd need to write it down like that:

```
def term =
    new Mult(new Value('a'), new Add(new Value('b'), new Value('c')))
```

That's quite a lot of `new` keywords! But if you apply the @Newify transformation, the expression becomes easier to read:

```
@Newify([Value, Mult, Add])
def term2 =
    Mult(Value('a'), Add(Value('b'), Value('c')))
```

Well, the expression itself is very nice, but you pay the tax of the annotation of the local transformation. But you are already familiar with `CompilerConfiguration` and its compilation customizer to transparently inject local transformations to make their use invisible from the users' perspective! I won't let you do this one as an exercise, as it's important to have a look at how we can inject a local transformation that takes parameters — so far, the ones we injected needn't any. Listing 46 shows you how do that.

Listing 18.46 Injecting a local transformation that takes parameters

```
def config = new CompilerConfiguration()
config.addCompilationCustomizers(
    new ASTTransformationCustomizer(                     #1
        value: [Value, Mult, Add], Newify)               #1
)
def shell = new GroovyShell(
    this.class.classLoader, new Binding(), config)       #2

def term3 = shell.evaluate '''                           #3
    Mult(                                                #3
        Value('a'),                                      #3
        Add(                                             #3
            Value('b'),                                  #3
            Value('c')                                   #3
        )                                                #3
    )                                                    #3
'''                                                      #3

assert term3.toString() ==
    'Mult(Value(a), Add(Value(b), Value(c)))'
```

In #1, we define an `ASTTransformationCustomizer` for the `@Newify` local transformation, and we pass the parameters needed by the transformation in the form of a map or named parameters. `@Newify` has a value as parameter, that needs a list of classes for values. When instantiating the `GroovyShell`, we need to pass the compiler configuration, but also the class loader of the script (as I ran these examples in one single compilation unit inside the Groovy console). Then in #3, we can evaluate our term expression without needing to use the `@Newify` annotation explicitely.

Until now, all our DSLs used various runtime and compile-time meta-programming techniques of Groovy. But all those mini-languages that we built were always designed within the boundaries of the syntax allowed by Groovy. In a way, it's nice because you've got the full power of a general-purpose dynamic language at your fingertips, but on the other hand, you are also limited by what the Groovy syntax allows. In the next section, we will discover a solution to allow some syntax adjustments to further customize your DSLs. Beware, it comes at a price, though!

18.9 Beyond the Groovy syntax

Groovy offers many ways to customize the language, both at compile-time and at runtime: you can add methods on the fly, add properties to number, do operator overloading to add operators to your own types, change the lookup logic in closures, and more. All this happens within the confine of the Groovy grammar.

But what if, for instance, you wanted to create your own custom operators? Groovy only allows you to overload a finite set of operators. You cannot create your own! It was a conscious design decision made by the creators of the language, to refrain developers from imagining ASCII art operators that would seriously alter the readability of your programs. But if for the purpose of a special DSL, you needed to invent a new operator? Although you should be careful to avoid creating a read-only language, you can actually create some derivative of the Groovy syntax yourself. Beware; this comes at a price, as your derived language won't be understood by IDEs, you would loose code-completion, and more. But we're grown-ups, aren't we? So let's see what we can do. But first, a little explanation on the compilation process.

The Groovy compiler uses the Antlr parser generator library to create a lexer and a parser for the syntax of the Groovy language that are used through the compilation process. From the textual form of your programs, Antlr generates a CST (Concrete Syntax Tree), from which the AST (Abstract Syntax Tree) is computed.

We already know there exists hooks, for instance, for acting on the AST for making various changes. This is what AST Transformations do! But you can also plug yourself earlier in the process, before the CST is generated. Here, we will be interested in looking at the text source before the CST is created. We can alter that text source, before feeding the lexer.

The idea I'm hinting at here is that we can take some text entry (your own programs and business rules), that may not comply with the Groovy grammar, make amendments to make it valid Groovy syntax, and then feed the compiler with that modified source.

Let's see that in action with a concrete example: we will work on the definition of a graph, with nodes linked together by arrows. First, we create our program and DSL in a classical fashion (within the bounds of the Groovy syntax), and then, we see how we can create a custom operator for it to make the code even more readable.

We'd like to be able to define graphs, like Figure 2 Graph example, with the following syntax:

```
graph {
    from a to b
    from a to c
    from a to e
    from c to d
    from c to e
}
```

We have a `graph{}` method taking a closure as argument, and within of the closure body, we define the arrows thanks to a command chain call (`from(node1).to(node2)`), and the nodes are actually defined as they are used.

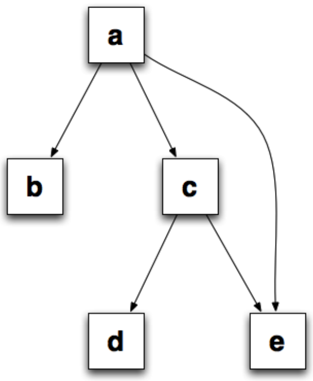


Figure 18.2 Graph example

To represent a graph, we need classes to represent nodes and arrows (the directed edge between our nodes). Listing 47 defines the classes `Node` and `Arrow`.

Listing 18.47 Node and `Arrow` classes

```
import groovy.transform.*

@TupleConstructor
class Node {
    String name
    String toString() { name }
}

@TupleConstructor
class Arrow {
    Node from
    Node to
    String toString() { "$from -> $to"
}
```

Our `Node` and `Arrow` classes are just plain POGOs, with a tuple constructor and a custom string representation. Nothing fancy so far. And we kept their implementations independent of the notion of graph. The meaty bits will come with our `Graph` class, in Listing 48.

Listing 18.48 The Graph class

```
@ToString                                                       #1
class Graph {
    List<Node>  nodes  = []                                     #2
    List<Arrow> arrows = []                                     #2

    static Graph graph(Closure c) {                             #3
        def graph = new Graph()                                 #4

        def definition = c.clone()                              #5
        definition.from = { Node fromNode ->                    #6
            [to: { Node toNode ->                               #6
                graph.arrows << new Arrow(fromNode, toNode)}    #6
            ]                                                   #6
        }                                                       #6
        definition.delegate = [:].withDefault { key ->          #7
            def n = new Node(key)                               #7
            graph.nodes << n                                    #7
            return n                                            #7
        }                                                       #7
        definition.resolveStrategy = Closure.DELEGATE_FIRST     #8
        definition()                                            #9

        return graph                                            #10
    }
}
```

We create our `Graph` class annotated with to `@ToString` transformation, in #1, to have a nice `toString()` representation that we will use later on for asserting our tree construction went fine. Our graph has got two list properties, in #2, to contain all the nodes of the graph, and the arrows linking the nodes between them. In #3, we create a static `graph{}` method (that will be statically imported in our graph definition script) to build the structure of the graph. What are we doing in this method? We instantiate a `Graph` object in #4. We clone the closure argument of the method in #5, as it's usually better to be safe in cases of the closure would be used in concurrent scenarios and have unwanted side-effects.

We add a `from` property closure in #6, which is used to support the `from(node1).to(node2)` notation with command chains. This closure will appear as if it were a `from()` method available in the body of the closure. This pseudo-method is responsible for creating the arrows, and adding them to the `arrows` list property of the graph.

Then, in #7, we define a delegate for the closure so that all unbound variables, like `a`, `b`, `c`, are actually transparently creating a `Node` instance, and add it to the `nodes` list property of the graph — note again the usage of Map's `withDefault{}`. And in #8, we set the resolve strategy of the closure to the delegate first, so that unbound variables are resolved against the closure's delegate.

Finally, we're able to call the closure in #9, and return the graph that we have produced through the definition closure, in #10.

With all that setup code ready, we are able to create a graph in Listing 49.

Listing 18.49 Creation of a graph

```
import static Graph.*                   #1

def g = graph {                         #2
    from a to b                         #3
    from a to c                         #3
    from a to e                         #3
    from c to d                         #3
    from c to e                         #3
}
```

```
assert g.toString() ==                    #4
    'Graph([a, b, c, e, d], [a -> b, a -> c, a -> e, c -> d, c -> e])'
```

We import statically the static `graph{}` method of the `Graph` class in #1. We use that method in #2 to define the structure of our graph, thanks to the declaration of all the direct edges between our nodes in #3. And we can assert in #4 that the structure of our graph is the one we wanted to represent, by comparing the `toString()` representation.

That's nice, our example works. But there's perhaps a downside… somewhat on purpose, obviously. The definition of the arrows linking the nodes together is perhaps a bit verbose, and the scientists using our DSL might want to have a more natural syntax, using, well, a real arrow character! Like the Unicode character U+2794: ➔, also called "HEAVY WIDE-HEADED RIGHTWARDS ARROW". So instead of writing `from a to b`, mathematicians could write a ➔□b

But what happens if we use such a notation in our graph definition? The Groovy compiler will complain with the following message:

```
Invalid variable name. Must start with a letter but was: ➔
```

A variable or a method name must be a valid identifier, and identifiers start with a letter. Our arrow is not considered to be a letter. So the code is not considered valid Groovy syntax, and the compiler rejects it.

BE CAREFUL

Our example text transformation is quite simple, but before jumping the gun and litter your code with cryptic characters, bear in mind that supporting Unicode can be problematic and lead to additional problems. First of all, how do your users actually enter that Unicode character with their keyboard? Are your DSL files properly encoded in UTF-8 or UTF-16, so that you don't see weird characters appear as the Unicode ones were not recognized? So take this example transformation with a grain of salt, and more as a proof of concept that you can do textual pre-processing before feeding Groovy with your DSLs.

We're in trouble here. We would like our DSL to support some syntax that is foreign, according to the Groovy grammar. But as we said in introduction, there's a solution for that: we can hook in the compilation process, and make a source code transformation to change it slightly so as to make it valid. So we would transform that arrow notation, with the notation we used in our implementation above.

To do that, we'll keep all our base classes exactly the same: `Node`, `Arrow`, and `Graph`. But we will create some kind of source pre-processor. As often for customizing the configuration, `CompilerConfiguration` will be our friend to set this up. We define a `ParserPluginFactory` class in Listing 50, that is used by the Groovy compiler to plugin the Antlr machinery.

Listing 18.50 A custom `ParserPluginFactory`

```
class SourcePreProcessor extends ParserPluginFactory {                #1
    ParserPlugin createParserPlugin() {                               #2
        new AntlrParserPlugin() {                                     #3
            Reduction parseCST(SourceUnit sourceUnit, Reader reader) {  #4
                def text = reader.text                                #5
                    .replaceAll(/(\w+)\d+➔\s+(\w+)/, 'from $1 to $2')         #6
                super.parseCST(sourceUnit, new StringReader(text))    #7
            }
        }
    }
}
```

In #1, we extend the base `ParserPluginFactory` abstract class, to create our own variant, the `SourcePreProcessessor` class. And in #2, we implement the `createParserPlugin()` abstract method.

This method must return a `ParserPlugin` implementation. Only two methods are needed for implementing that interface, but the sole existing implementation in the Groovy code base, `AntlrParserPlugin`, weighs in at 3000 lines of code and does a lot of work to create the Concrete Syntax Tree and then transform it into the initial Abstract Syntax Tree. So instead of reinventing two big weels, we follow the approach of simply extending Groovy's `AntlrParserPlugin`, and add our own bits where it makes sense, and call `super` to do the rest of the work. Good developers are supposed to be lazy, aren't they? So in #3, we create an anonymous inner class extending that big class. We are particularly interested in the method that is going to create the CST from the initial source text. This method, in #4, has got two parameters, the second of which is a `Reader` over the source text!

Two steps will do the work. First of all, we read the content of the reader in #5, and replace all the occurrences of the regular expression of an arrow surrounded by two node names, with a new string with the from / to syntax, in #6. And for the heavy lifting, we're going to call back to the `super` method, passing it a new reader over our transformed string, in #7. That way, the Groovy compiler will actually parse the modified source, which is now valid Groovy syntax, and not the initial one.

We're not totally done yet, we need to configure and wire everything now. That's what we're going to do next, in Listing 51.

Listing 18.51 Pluging and testing the pre-processor

```
def conf = new CompilerConfiguration()                                  #1
conf.pluginFactory = new SourcePreProcessor()                           #2
conf.addCompilationCustomizers(                                         #3
    new ImportCustomizer().addStaticImport(Graph.name, 'graph'))        #3

def shell = new GroovyShell(this.class.classLoader,                     #5
                        new Binding(), conf)                            #5

def g = shell.evaluate("""                                              #6
    graph {                                                             #4
        a ▨ b
        a ▨ c
        a ▨ e
        c ▨ d
        c ▨ e
    }
""")

assert g.toString() ==                                                  #7
    'Graph([a, b, c, e, d], [a -> b, a -> c, a -> e, c -> d, c -> e])'
```

Let's call our good old friend, `CompilerConfiguration`, to the rescue, in #1. We set the `pluginFactory` property of our configuration instance to point at our `SourcePreProcessor` in #2. After that, in #3, we add an import customizer for adding a static import for the `graph{}` method of the `Graph` class, so that the end-user doesn't have to do the static import himself in his script, in #4. We instantiate the shell in #5, we call the evaluation method with our source with the nice Unicode arrows in #6, and then we assert that the graph generated is correct in #7.

With this approach, we managed to take a graph definition which wasn't conformant with the Groovy grammar, turn it into proper Groovy syntax, and have the Groovy compiler parse that modified and compliant source. However, let me remind you this comes at a price: the source is indeed not valid Groovy syntax, and tools grokking Groovy code won't understand that syntax. Your IDE will complain with red squiggles all over the place in the editor, with compilation errors if you'd try to compile that class directly. Static analysis tools such as CodeNarc won't be able to do their job either. So be careful when you fire this gun! Perhaps the price is too high to pay for the benefits of pleasing the eye of mathematicians. Also, perhaps you might have found an existing operator that could have been overloaded? For example, the right shift operator >> would have looked enough like an arrow? Perhaps a method was actually okay? Be sure to think about the various options, and the cost / benefits, when designing the various aspects of the syntax of your DSL.

In the example we presented here, we had only a small modification to make, that a regular expression could do. But be careful not to do too many alterations of the grammar, otherwise, you'll really be implementing your own language. Then, regular expressions won’t be enough, and you will need to create your own lexer, parser, etc.

That said, to open up some perspectives, it can be interesting for cross-compilers that would compile other languages’ code, like JavaScript, to Groovy code — although the other way might be more interesting to make Groovy code run in the browser as JavaScript. Another example of usage can be for a new template engine that would spice up its syntax with Groovy fragments to have the full power of a general-purpose dynamic language.

In the case you need changes that a simple regular expression can’t fix, you can take the approach of generating the Groovy AST yourself, by providing a full-blown implementation of the `ParserPlugin` interface, but remember that this might be quite a lot of work to do.

All the techniques that we learned until now are about empowering developers to create nicely crafted DSLs, and end-users to code their business rules with a more concise and readable language, than with a plain programming language. But as the saying goes, with power comes great responsibilities! On one hand, as a developer of the DSL, you could trust your users to “do no harm”, but on the other hand, you could protect yourself from mistakes or intentional misbehavior, by securing your DSLs. That’s the purpose of the next section.

18.10 Securing your DSLs

The nice aspect of an embedded or internal DSL is that you have all the underlying language at your disposal for coding your business rules: using branching constructs, loop constructs, the wealth of the JDK APIs and third-party libraries, and so on. But sometimes, certain DSLs are really reduced in scope, and shouldn’t use anything beyond the area this DSL is supposed to be covering. Furthermore, there are situations where malicious usage of the DSL, the underlying language or the APIs could wreck havoc in your running application, or the overall system, and open up breaches of security.

Since we are on the Java platform, an obvious solution is to use a Java security manager. You can grant permissions or prevent access to certain methods (`System.exit(0)` anyone?), to system properties, to the file system, and many more things. There’s already ample enough documentation on this topic elsewhere, and that’s not the goal of this chapter to cover aspects of Java itself. But be sure to remember this facility when you try to secure your DSL. Also think of the cost of a security manager, as the security checks are happening at runtime, this may lead to longer execution times for your business rules. This might not be acceptable if your code needs to execute as fast as possible.

18.10.1 Introducing SecureASTCustomizer

In the previous sections, we had the opportunity to use compiler customizers for injecting imports or AST transformations. But there’s more! You can even actually create your own customizer by extending the `CompilationCustomizer` class, but here, we’ll be investigating another existing customizer: `SecureASTCustomizer`.

Just like the other customizers, this one should be set on the `CompilerConfiguration` object. It sports several setters to tell if the scripts and classes are allowed to:

- define a package name,
- define a method,
- define a closure.

And there is a white list / black list mechanism to say if the scripts and classes can use:

- imports: simple imports, static imports, star imports, static star imports,
- the various types of: statements, expressions, tokens, constant types, and receivers.

To get our feet wet, let’s dive in with a concrete use case. When you offer an extension point in your application, you expose an API (better, a DSL!) that users can use to interact with your software. If users should really only use those classes from that API, you can forbid them to access any other class, thanks to our secure customizer. In that case, the “whitelist” approach is interesting, as you can specify you only want to allow the usage of classes coming from a certain package. Progressively, you can open up other utility classes users may need. With the "blacklist" approach, you allow everything, except some classes. For example, your end-users shouldn't have access to the file system — note this is a case that is covered by security managers as well. Listing 52 shows how you can prevent access to classes from the `java.io` package.

Listing 18.52 Prevent access to `java.io` classes

```
import org.codehaus.groovy.control.*
import org.codehaus.groovy.control.customizers.*

def secure = new SecureASTCustomizer()
secure.starImportsBlacklist = ['java.io.*']                #1
secure.indirectImportCheckEnabled = true                   #2

def config = new CompilerConfiguration()
config.addCompilationCustomizers(secure)                   #3

def shell = new GroovyShell(config)

shell.evaluate '''                                         #4
    new File('.')                                          #4
'''                                                        #4
```

In #1, you specify the list of star imports that are forbidden. And in #2, we also enable the indirect import check as someone may be using fully qualified class names, instead of an import. We add the customizer to the compiler configuration object. Then, when you evaluate the script in #4, you get an error message like the following:

```
java.lang.SecurityException: Indirect import checks prevents usage of expression
     at secure1.run(secure1.groovy:13)
Caused by: java.lang.SecurityException: Importing [java.io.File] is not allowed
     ... 1 more
```

The error message tells us the import of the file class is not allowed. We didn't import it, but Groovy has an implicit import for `java.io` classes, and the indirect import checks helped us catch this case. It's usually a good practice to use the indirect check flag, especially in the cases where people use fully qualified names.

This first example was a bit trivial, but the samples offered by the Groovy project provide an interesting case study: the arithmetic shell.

18.10.2 The ArithmeticShell

You can use Groovy as an arithmetic expression evaluator. But if you do so, what would prevent users from doing things like `System.exit(0)` in your formulas? The arithmetic shell uses the secure customizer to only allow arithmetic expressions, and forbid anything else, be it using closures, creating classes, importing other classes than `java.lang.Math`, etc. Listing 53 is a reformatted excerpt of the `ArithmeticShell` class.

Listing 18.53 Configuration of the `ArithmeticShell` secure customizer

```
def secure = new SecureASTCustomizer()
secure.with {
    closuresAllowed = false                                     #1
    methodDefinitionAllowed = false                             #1

    importsWhitelist = []                                       #2
    staticImportsWhitelist = []                                 #2
    staticStarImportsWhitelist = ['java.lang.Math']             #2

    tokensWhitelist = [                                         #3
```

```
            PLUS, MINUS, MULTIPLY, DIVIDE, MOD, POWER,                #3
            PLUS_PLUS, MINUS_MINUS,                                   #3
            COMPARE_EQUAL, COMPARE_NOT_EQUAL,                         #3
            COMPARE_LESS_THAN, COMPARE_LESS_THAN_EQUAL,               #3
            COMPARE_GREATER_THAN, COMPARE_GREATER_THAN_EQUAL,         #3
    ]

    constantTypesClassesWhiteList = [                                 #4
            Integer, Float, Long, Double, BigDecimal,                 #4
            Integer.TYPE, Long.TYPE, Float.TYPE, Double.TYPE          #4
    ]

    receiversClassesWhiteList = [                                     #5
            Math, Integer, Float, Double, Long, BigDecimal            #5
    ]

    statementsWhitelist = [                                           #6
            BlockStatement, ExpressionStatement                       #6
    ]

    expressionsWhitelist = [                                          #7
            BinaryExpression,       ConstantExpression,
            MethodCallExpression,   StaticMethodCallExpression,
            ArgumentListExpression, PropertyExpression,
            UnaryMinusExpression,   UnaryPlusExpression,
            PrefixExpression,       PostfixExpression,
            TernaryExpression,      ElvisOperatorExpression,
            BooleanExpression,      ClassExpression
    ]
}
```

Allowing arithmetic expressions only is not such an easy task, when you have to somehow “dumb down” a full-blown programming language to just allow such expressions. To commence, things like using or defining closures, and defining methods (forbidden in #1) has nothing to do with arithmetic expressions. In #2, the white list mechanism is used to really only allow the static star import of all the `Math` static methods, which provides methods like sine, cosine and friends. In #3, the tokens recognized by the Groovy lexer are filtered to only allow the ones that could make up math expressions, like all the arithmetic operators, increment / decrement operators, and comparison operators. Numbers literals (the various `*.TYPE` elements) and the usage of the various `Number` classes are allowed in #4. The receiver classes, in #5, are classes that can be used and that can *receive* method calls. In #6, block statements and expression statements are allowed since an expression is wrapped in an expression statement, whose part of a block statement, which is the body of your script (your formula is the body of the `run()` method of `Script`). And to finish, a list of expressions that are white listed, but they are too numerous to detail them all.

That was quite a ride! When you really want to restrict very precisely what users can do with the language, crafting the right rules of exclusions and inclusions can be a long task.

As an exercise in hacking, you could have a go at trying to find a workaround to do things that shouldn't be allowed by this secured arithmetic shell. Often, remember that hackers are more malicious that you can be, and they can find back doors easily. They could put your system down by doing as simple as running an infinite loop doing nothing but consuming precious CPU cycles. How can you stop this?

18.10.3 Stopping the execution of your programs

Your application provides an extension point with a nice DSL that your users can use. For example, imagine a wiki engine that would allow authors to make their pages dynamic with some Groovy scripting inside the wiki markup of the pages. What if a malicious or not very careful user creates an infinite loop, what can you do to prevent this? Neither security managers not a secure customizer can help there much. But Groovy provides three interesting AST transformations that you can apply to the sources of your scripts so that you can stop their execution when a thread is called to be interrupted, after an elapsed period of time, or after some custom condition is met (for instance when too much of resource is used, etc.).

Chapter 9 on AST Transformations actually already covers the `@ThreadInterrupt`, `@TimedInterrupt` and `@ConditionalInterrupt` local transformations, so we won't go into much details about their usage. I'll just remind you how to make local transforms transparent to the users. As those transformations are local, users would need to put those annotations themselves in their scripts. To continue our idea of a scriptable wiki, they would need to litter their scriptlets without those annotations.

REMEMBER

Before going further, remember that such transformations can only be applicable on scripts and classes that are going to be compiled. You cannot apply them after the compilation has already happened — for example if you wanted to post-process classes from a JAR file, etc.

In previous sections, we have already learned this technique, but we can apply it again here, by looking at how we inject the `@TimedInterrupt` in an infinite looping script.

```
import groovy.transform.*
import org.codehaus.groovy.control.*
import org.codehaus.groovy.control.customizers.*

def config = new CompilerConfiguration()                                #2
config.addCompilationCustomizers(                                       #2
    new ASTTransformationCustomizer(value: 5, TimedInterrupt)           #2
)
def shell = new GroovyShell(config)
shell.evaluate '''
    for (i in 1..1000) {                                                #1
        sleep 1000                                                      #1
}
'''
```

A script like #1 would loop a thousand times, sleeping one second at each iteration, totaling more than 16 minutes of execution time. That can be a bit long for code that's really not doing anything useful! Thus in #2, we define and configure an `ASTTransformationCustomizer` for the `@TimedInterrupt` transformation, that will wait for 5 seconds before the script is interrupted. As usual, this customizer is specified on the `CompilerConfiguration` that we pass in the constructor of the `GroovyShell`.

Now what happens when you execute that program? You get a `TimeoutException` after 5 seconds:

```
java.util.concurrent.TimeoutException: Execution timed out after 5 units. Start time: Sun
Aug 21 01:09:44 CEST 2011
```

We saw how to filter the AST with the secure customizers, or how to stop the execution of long running or resource consuming business rules, but some malicious code could try to cheat with your software by using some meta-programming tricks. Let's see what you can do to prevent this to happen.

18.10.4 Preventing cheating with meta-programming

A customer I worked with didn't want to use security managers to forbid calls to `System.exit(0)` in their business rules, as security managers would almost double the runtime execution speed of those rules. They ended up hooking into the Groovy compiler to filter the AST to check for method call expressions that would happen on the `java.lang.System` class, with a method name of `exit`. They did that before the compilation customizers even existed. We'll replicate with customizers what they did. Interestingly, we will also learn how to create our own customizer beyond the three we have already learned about.

Listing 54 creates a custom customizer for `System.exit()`.

Listing 18.54 Failing compilation on `System.exit()`

```
import org.codehaus.groovy.ast.*
import org.codehaus.groovy.ast.expr.*
import org.codehaus.groovy.control.*
import org.codehaus.groovy.control.customizers.*
import org.codehaus.groovy.classgen.*
import org.codehaus.groovy.syntax.*
import static org.codehaus.groovy.control.CompilePhase.*

def config = new CompilerConfiguration()
def filter = new CompilationCustomizer(CANONICALIZATION) {             #1
   void call(SourceUnit src, GeneratorContext ctxt, ClassNode cn) {     #2
      new ClassCodeVisitorSupport() {                                   #3
         void visitMethodCallExpression(MethodCallExpression call) {    #4
            if (call.objectExpression.text == 'java.lang.System' &&     #5
               call.method.text == 'exit') {                            #5
               src.addError(new SyntaxException(                        #6
                   'System.exit() forbidden',                           #6
                   call.lineNumber, call.columnNumber))                 #6
            }
            super.visitMethodCallExpression(call)
         }
         SourceUnit getSourceUnit() { src }
      }.visitClass(cn)
   }
}
config.addCompilationCustomizers(filter)

def shell = new GroovyShell(config)
shell.parse '''
    System.exit(0)
'''
```

To create your own customizer, you just need to extend `CompilationCustomizer`, like we do in #1 by creating an anonymous inner class. You must then implement the `call()` abstract method, in #2. This method has the `ClassNode` of the script or class to be introspected as argument, for finding the offending method call. To do that, we're creating a `ClassCodeVisitorSupport` object in #3 that will visit all the method call expressions in #4. For each such call, we'll check if the receiver type is `java.lang.System` and if the method name is `exit`, in #5. If this is the case, in #6, we add a compilation error that will fail the compilation of the script or class — we'll come back in the next section on error reporting.

When compilation fails, you get a message like the following:

```
System.exit() forbidden at line: 2, column: 5
```

It's a bit more verbose than using a security manager, but it hasn't got any performance cost, which is good. So you might think “job done”! But are we really done with it here? We indeed fail the compilation when someone explicitly and literally writes `System.exit()` in the source code. Fine, and that's something along those lines that my customer did. But when I did the code review, I quickly noticed this wouldn't really cover all the cases — a case that the security manager would have caught though. As you start to know Groovy pretty well now, you might find out how to call `System.exit(0)` bypassing our new customizer.

If you try our customizer on the following code, the compilation will work, and running the code your program will exit:

```
shell.parse '''
    def clazz = 'java.lang.System' as Class
    def method = 'e' + 'x' + 'i' + 't'
```

```
        def params = [0]
        clazz."${method}"(*params)
    '''
```

The last line is an offending call to `System.exit()` that our customizer couldn't spot, since the values of the class, method name and parameters could not really be figured out until runtime, until the execution of the code itself.

So apart from using a security manager, what could we do? In this case, you could add some additional checks in your code checking the method call expressions to disallow those whose method expression is a `GStringExpression`:

```
if (call.method instanceof GStringExpression) {
    source.addError(new SyntaxException('GString method names forbidden',
        call.lineNumber, call.columnNumber))
}
```

Then the compilation will fail as expected even in that odd forged case. But it comes at the price of disallowing GString method calls, which may have been useful in your DSL in some context. As usual, there's no free lunch! Also, clever Groovy users can forge some innovative method calls by abusing other Groovy constructs. Securing your scripts and classes is not that trivial, but it also depends on how much secure it needs to be in the first place.

With a similar approach, you would want to prevent DSL users to do any meta-programming in the business rules they author. For example, to filter access to the metaclass to alter the behavior of certain classes, you might add the following checks in your customizer to prevent metaClass property access as well as GString property access (that could forge a metaClass property name):

```
void visitPropertyExpression(PropertyExpression expr) {
    if (expr.property.text == 'metaClass') {
        src.addError(new SyntaxException('Accessing metaClass forbidden',
             expr.lineNumber, expr.columnNumber))
    }
    if (expr.property instanceof GStringExpression) {
         src.addError(new SyntaxException('GString access forbidden',
             expr.lineNumber, expr.columnNumber))
    }
    super.visitPropertyExpression(expr)
}
```

For meta-programming alterations that would use categories, you could add checks in the method call expression visit method the following check:

```
if (call.objectExpression.text == 'this' && call.method.text == 'use') {
    src.addError(new SyntaxException('use(category){} forbidden',
        call.lineNumber, call.columnNumber))
}
```

Securing DSLs is an important aspect of their design, but another often overlooked aspect is the testing and error reporting phases, which are key to the quality and success of your endeavor. The next section proposes to look into this topic a bit more.

18.11 Testing and error reporting

As software developers, we often test the nominal cases first, to check that we properly implemented a feature. But we tend to think of the edge cases as an afterthought. Make sure to put the emphasis on testing various cases, especially including errors like typos a user could make!

There aren't particular techniques for testing Domain-Specific Language implementations per se, except perhaps the `TransformHelper` testing utility class that you used in Chapter 9, but we'll focus our discussion on making our DSLs more robust, in order to make the life of our users better, and see how we can make those edge cases give better and more meaningful error messages.

For our journey, we'll work on a SQL-like query language, with verbs and words like `select`, `from`, and `where`. We want to be able to issue queries like this:

```
query {
    select all from users
    where lastname == 'Guillaume'
}
```

In Listing 55 we scaffold our DSL.

Listing 18.55 Query language

```
class Query {
    static query(Closure c) {                                   #1
        def q = c.clone()                                       #2
        q.resolveStrategy = Closure.DELEGATE_FIRST              #2
        q.delegate = new Query()                                #2
        q()                                                     #2
    }

    def getProperty(String name) { name }                       #3

    Query select(column)    { this }                            #4
    Query from(table)       { this }                            #4
    Query where(condition) { this }                             #4
}
```

Our `Query` class features a query method that takes a closure as argument, in #1. We'll use a static import of that method later on (that we can inject with the approaches seen in earlier in this chapter). In #2, we specify the closure resolve strategy to have the `Query` instance to receive all the method calls and property lookups first. The parameters to our various verbs are looked up through the `getProperty()` method which just return strings for now in #3 — we're not building the full blown DSL! Finally, all our verbs are methods returning `this`, in #4, so as to be able to chain method calls with command chain expressions.

Now what happens when you execute a query where you make a typo in the verbs? For example:

```
query { selct all }
```

In the Groovy Swing console, you'd see an error like the following:

```
groovy.lang.MissingMethodException: No signature of method: Query.selct() is applicable
for argument types: (java.lang.String) values: [all]
    Possible solutions: select(java.lang.Object), split(groovy.lang.Closure),
getAt(java.lang.String), sleep(long), each(groovy.lang.Closure), wait()
     at errorreporting$_run_closure2.doCall(errorreporting.groovy:23)
     at errorreporting$_run_closure2.doCall(errorreporting.groovy)
     at Query.query(errorreporting.groovy:6)
     at Query$query.callStatic(Unknown Source)
     at errorreporting.run(errorreporting.groovy:23)
```

Highlighted in bold, you see that a `MissingMethodException` is thrown. The `selct()` method doesn't exist, and Groovy even suggests possible alternatives, like the right `select()` method. You also notice the line information where the problem occurred, although the class / method parts of the stacktrace is perhaps a bit obscure.

Should you want to provide your own exception and message, you could add a `methodMissing()` method to your `Query` class, so that query methods which are mistyped would go through that trap:

```
def methodMissing(String name, args) {
        throw new SyntaxException(
            "No query verb '$name', only select/from/where allowed"
        )
    }
```

This method uses your own custom syntax exception:

```
import groovy.transform.*

@InheritConstructors
class SyntaxException extends Exception {}
```

When you run your query, you'll get the following trace:

```
SyntaxException: No query verb 'selct', only select/from/where allowed
     at Query.methodMissing(errorreporting2.groovy:18)
     at Query.invokeMethod(errorreporting2.groovy)
     at errorreporting2$_run_closure2.doCall(errorreporting2.groovy:35)
     at errorreporting2$_run_closure2.doCall(errorreporting2.groovy)
     at Query.query(errorreporting2.groovy:12)
     at Query$query.callStatic(Unknown Source)
     at errorreporting2.run(errorreporting2.groovy:35)
```

With this approach, you get your own custom exception, with an even more explicit message, rather than the ones provided by Groovy itself. You'll notice however that the exception is coming directly from the `methodMissing()` method, and not from the place that issued the call, which happens only a couple stacktrace elements later.

The stacktraces shown here are actually already filtered in the Groovy Swing console to only show relevant elements of your own programs, and the full stacktrace outputted in your shell is much longer. You could still filter out some more by removing the stacktrace elements that hides away the right method call site by changing the `methodMissing()` implementation like this:

```
def methodMissing(String name, args) {
    def se = new SyntaxException(
        "No query verb '$name', only select/from/where allowed"
    )
    se.stackTrace = se.stackTrace.findAll { StackTraceElement ste ->
        ste.className != 'Query' &&
        !(ste.methodName in ['invokeMethod', 'methodMissing'])
    }
    throw se
}
```

We rewrite the stacktrace elements array of the exception, by removing the offending elements we don't want the user to see, so as to only see where the problematic DSL usage is situated. Then the filtered trace is more obvious and shows the relevant line first:

```
SyntaxException: No query verb 'selct', only select/from/where allowed
     at errorreporting3$_run_closure2.doCall(errorreporting3.groovy:41)
     at errorreporting3$_run_closure2.doCall(errorreporting3.groovy)
     at Query$query.callStatic(Unknown Source)
     at errorreporting3.run(errorreporting3.groovy:40)
```

Whether you let Groovy throw its method missing error and suggestion fixes, or if you choose to use your own explicit custom exception possibly with a filtered stacktrace, there's one common downside to both approaches: the exception happens at runtime, and not at compile-time!

Usually, developers coming from a statically typed language prefer catching errors as early as possible, and compilation would be the perfect moment. Similarly, business users using your DSL would appreciate that you offer them a tool (through mere compilation) letting them know they made a mistake, a typo, early on, rather than when the business rules are deployed in the production environment. For generating compile-time errors, nothing's better than an AST transformation or a compilation customizer! Using a customizer, you could get out with Listing 56.

Listing 18.56 Check query method names usage

```
new CompilationCustomizer(SEMANTIC_ANALYSIS) {
  void call(SourceUnit src, GeneratorContext ctxt, ClassNode cn) {
    new ClassCodeVisitorSupport() {
      boolean inQueryClosure = false

      void visitStaticMethodCallExpression(                           #1
          StaticMethodCallExpression call) {                          #1
        if (call.method == 'query' && call.ownerType.name == 'Query')    #1
          inQueryClosure = true
        super.visitStaticMethodCallExpression(call)
        if (inQueryClosure)
          inQueryClosure = false
      }

      void visitMethodCallExpression(MethodCallExpression call) {
        def methName = call.method.text
        if (
          inQueryClosure &&                                           #2
          call.objectExpression.text == 'this' &&                     #2
          !(methName in ['select', 'from', 'where'])) {               #2
            src.addError(new SyntaxException(                         #3
              "No query verb ${methName}, only select/from/where",    #3
              call.lineNumber, call.columnNumber))                    #3
        }
        super.visitMethodCallExpression(call)
      }

      SourceUnit getSourceUnit() { src }
    }.visitClass(cn)
  }
}
```

With the static import of the `Query.query{}` method, we check that we're in the context of such a call, by implementing the `visitStaticMethodCallExpression()` method, in #1, keeping a boolean flag up-to-date. And in #2, we check that all method call within that context have the correct spelling.

What's more important here is we'll have a closer look at #3, where we add the error message. We're calling the `addError()` method on the `SourceUnit`. When you use a customizer or an AST transformation, that's the method to use if you want to create a compilation error. This method takes either a `SyntaxException` as argument with whose constructor you can give an error message, but also the position of where the error is supposed to happen. There you can reuse the current AST node's line and column information, so as to have the compiler deliver nice error message with proper location.

The `SourceUnit` class also provides an `addException()` method which let you pass an exception as argument, but I would argue this method is less interesting as it's not generating the usual syntax errors IDEs would expect, nor does it give a change to properly specify position information. So I'd avoid using this version.

For more control over the error reporting from the `SourceUnit` when you're using a customizer or transformation, you can retrieve its error collector with the `getErrorCollector()` method, and then call more fine-grained methods than the two provided directly, as a shortcut, on `SourceUnit`. You'll be able, for example, to say if the error you're creating should fail the compilation right away, or continue to find other potential errors.

WHEN TO USE A TRANSFORM VERSUS A CUSTOMIZER

For compilation errors, you might wonder when you should use an AST transformation or if you'd rather prefer a compilation customizer, as transforms and customizer offer pretty much the same approach — introspecting the AST. The usual consultant's answer is… it depends! If depends on the techniques you used to implement your DSL, as well as on the integration strategy you've chosen to compile and execute your business rules. If you are already using a transform for implementing your DSL or parts of it, you should seize your chance to also add proper compilation errors there. If you are integrating your business rules by using Groovy's shell, classloader, scripting engine, etc, then you are able to define a compiler configuration object to configure the compilation, and thus add your customizers at that point. But if you are just using global or local transformations without a particular integration mechanism (ie. your code is pre-compiled, not compiled on the fly), your sole option will be to use an AST transformation.

We saw the case where the DSL users make a typo in a method name, but we can also have a quick look at what happens if he uses wrong arguments for the methods forming the verbs of the DSL. More concretely, what happens if a user passes a string instead of a date arguments?

To try that for real, let's come back to our initial `Query` class from the beginning. We'll need a new verb for our experiment: an `after()` action to check that some database result is after a certain date. We just need to add that method to our class:

```
Query after(Date d) { this }
```

This time, this method takes a `Date` instance. What happens if we pass a string instead of a date, for example a string representing a date, like `'2011/08/22'`? With neither a method missing trap, nor an AST transformation checking for mistyped verbs, you'd get an exception like the following one:

```
groovy.lang.MissingMethodException: No signature of method: Query.after() is applicable
for argument types: (java.lang.String) values: [2011/08/22]
Possible solutions: after(java.util.Date), ...
```

The message is as informative as before, so you could go away with it, even if it's not your custom exception showing up here.

If you had put in place a missing method trap, the error you'd get would be a bit more misleading though:

```
SyntaxException: No query verb 'after', only select/from/where/after allowed
```

Since the method with the proper signature wasn't found, it goes through our trap, and our error message is indeed misleading as the user typed `after` in his query, but the message seems to indicate `after` doesn't exist? So you could improve the error message to make it clearer than a verb is not just only the name of that verb, but also the type of arguments that it takes. You could also investigate using our earlier customizer to add proper checks for types, but sometimes, the Groovy AST doesn't always have enough type information at compile-time to figure out if there's an error or not. But for now, we're going to consider an alternative taking advantage of multimethods.

If you have overloaded methods taking different arguments, Groovy will always try to call at runtime the most appropriate method according to the runtime types of the arguments. In a nutshell, that's what we call multimethods. The idea is to play on that specific aspect of the multiple dispatch logic to lead your DSL to give better error messages.

In our case, we have an `after(Date)` method, but we can add an `after(Object)` method:

```
Query after(Object d)  {
    throw new SyntaxException(
        "The after method takes a Date as argument, " +
        "not ${d} of type ${d.class.name}")
}
```

Which yields a nicer error message:

```
SyntaxException: The after method takes a Date as argument, not 2011/08/22 of type
java.lang.String
     at Query.after(errorreporting5.groovy:25)
```

By overloading your DSL methods with ones taking a mere object type, if the user makes a mistake in terms of type, he will get a more precise and meaningful error message, as the multiple dispatch will route the call to that trap method, rather than letting the Groovy runtime not finding a matching method to call.

18.12 Summary

The main purpose of Domain-Specific Languages is to bridge the communication gap that leads to misunderstanding the needs of the end users, software bugs, delays in delivery, and inadequacy with the real requirements. To bridge this gap, we have learned in this chapter how to combine many features and techniques together to build your own DSLs.

We covered a lot of ground: Groovy's flexible syntax and command chains, static imports, constant and method injection, custom control structures, closure delegation strategy, AST transformations, compilation customizers, integration approaches, security concerns, and error reporting. Often, building a DSL in Groovy is a clever fashion of complying with the various call conventions to actually pimp an existing library so as to turn it into an easy to use DSL from Groovy.

By cleverly mixing these techniques, our business rules achieve a high level readability, more conciseness, without necessarily verging into ASCII art. The form of the DSL matters, and we need to keep in mind that a form or another might be more approachable for our end-users. That's why we need to work as a team, involving users early in the process, and work iteratively towards crafting the right language that everybody will understand.

As a parting thought, remember that your role as a knowledgeable Groovy developer and barman mixing ingredients for a nice DSL cocktail, that over-engineering your beverage might add too much complexity, increase the cost, and might sometimes go beyond what your end-users need.

19
The Groovy Ecosystem

"I can't imagine why anyone would need X" is a statement about your imagination, not X.

Dan Piponi, via Twitter

Groovy is a rich and flexible language, and every day Groovy programmers are finding new, novel, and exciting ways to bend Groovy to their needs. The Groovy Ecosystem refers to all of the projects built around Groovy; projects that solve a particular problem for a particular group of people, and projects that are essential to being a productive Groovy programmer.

This chapter starts by examining projects that make using Groovy as a scripting language and automation tool easier: Grapes for managing dependencies within scripts, Scriptom for working with Windows components, GroovyServ for making scripts run fast, and Gradle for project and task automation.

Groovy is a good choice to use as a system scripting language. Groovy scripts are easily executed from the command line and can automate repetitive tasks. Groovy is far less verbose than Java, can easily spawn new threads and processes, and has many convenience methods for interacting with the file-system. But the biggest advantage of scripting with Groovy is that you have access to every library you use in development. Need a script to access a SOAP based web service? Then you can use the same library from your development project within your script. Need to download and manipulate web pages? You can use `XmlParser` and the TagSoup Java library. Any Java library is available to Groovy, and that means it's available within a script.

After scripting, we'll look at two interesting projects meant to bring a higher level of quality for larger Groovy projects: CodeNarc for static analysis of Groovy code and GContracts for Design by Contract™ within Groovy. Then we'll look at three popular application development platforms tailor-made around Groovy: Grails for web application development, Griffon for desktop development, and Gaelyk for Google App Engine applications. Finally we'll explore the new frontier of Groovy++, a project bringing static, compile-time type checking and major speed improvements to Groovy.

Buckle up and hold on - the whirlwind tour is about to start.

19.1 Groovy Grapes for Self-Contained Scripts

A common usage of scripts is working with other teams, like the quality assurance or operations team. Perhaps there is a hard to reproduce defect and you need someone to execute a script on a remote machine. It is easy to email them a script, but if the script has a dependency on several jar files, then how do you easily package all that up into something executable? How do you get libraries onto the classpath correctly? This is the problem Grapes was invented to solve.

Grapes lets you add maven dependencies to your classpath from within a .groovy file. The script can then be executed without downloading the dependencies and constructing a lengthy command line. For example, consider a script that uses the TagSoup library to read data out of poorly formatted HTML files. Listing 19.1 gives an example of this, printing all the links in a person’s twitter stream.

Listing 19.1: Using Grapes in the parseTwitter.groovy Script.

```
@Grab(group='org.ccil.cowan.tagsoup', module='tagsoup', version='1.2')
import org.ccil.cowan.tagsoup.Parser

def parser = new XmlParser(new Parser())
def html = parser.parse("http://twitter.com/hamletdrc")

html.body.'**'.a.@href.grep(~/http.*status.*/)each {
    println it
}
```

This script declares a dependency on TagSoup version 1.2 using the `@Grab` annotation. To execute this script, simply invoke it with the command `groovy parseTwitter.groovy` and you will see a set of URLs printed to the console.

Clearly the script is importing and invoking objects from the TagSoup library, but where did this dependency come from and how was it resolved? The answer is that the Grapes module system is aware of the `@Grab` annotation. *Before* a script is executed, Groovy reads the `@Grab` annotations and resolves the parameters as Maven dependencies. Those dependencies are downloaded, resolved, and added to the classpath of the script. Only once all of the dependencies are resolved and in-scope does the script execution begin. There is no need to ever email another jar file to someone or construct a long classpath statement from the command line just so they can run your script. Grapes has you covered.

UNDER THE HOOD

Groovy uses Ivy to download the declared dependencies into your Grape cache, which is located in the `.groovy/grape` directory of your User Home directory. You can change this directory by passing a JVM parameter to groovy named grape.root. For example, passing "`-Dgrape.root=/home/.m2/repository`" configures Grapes to use your local Maven repository for the cache.

Many companies maintain their own internal Maven repository for their own propriety software or because they do not want developers downloading files from the Internet. You can tell Grapes about your own repositories using the `@GrabResolver` annotation. For example, if TagSoup was located in your own repository hosted at http://myrepo.my-company.com, then you would add the `@GrabResolver` annotation to your script, like so:

```
@GrabResolver(name='myrepo', root='http://myrepo.my-company.com/')
@Grab('org.ccil.cowan.tagsoup:tagsoup:1.2')
import org.ccil.cowan.tagsoup.Parser
```

These two annotations give you pretty much everything you need for working with Grapes. Also, notice how we specified the Maven dependency using the short form that separates the common parameters using a colon. There is a lot more customization available as well. Grapes is controlled by the ./groovy/grapeConfig.xml file in your user home directory. You can edit this file to permanently add Grape resolvers, change the local repository directory, and configure network proxies. Dependencies can also be manually installed, removed, and listed using the Grapes command line interface.

Grapes is becoming more popular and is a simple way to manage dependencies. It puts the abundance of Java libraries directly in the hands of the Groovy programmer. Tooling is becoming more frequent as well. IntelliJ IDEA has explicit support for Grapes and can automatically configure your project structure, and the website MvnRepository (http://mvnrepository.com), which allows you to search for dependencies, displays the correct @Grab usage for any library you find through their site. Grapes is an enabling technology that's fueling Groovy adoption, and it's an important fundamental to understand when getting ready to work with the Groovy ecosystem.

That's all you need to know about Grapes to get started, and bear in mind that it makes life easier when working with the libraries in the rest of this chapter. Next up we'll see how Scriptom makes COM and ActiveX scripting easier.

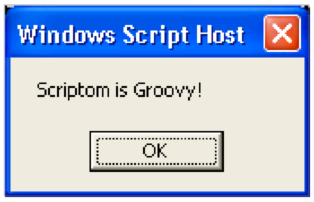
19.2 Scriptom for Windows Automation

Scriptom's name stems from a mix of the word scripting and the acronym COM, Microsoft's component object model. Scriptom allows you to manipulate COM and ActiveX objects as simply as if you were using Visual Basic or JavaScript. Combining Scriptom and Groovy means that you can take advantage of the Java world and its libraries, and at the same time control applications such as Microsoft Word or Excel from Groovy.

Scriptom is an add-on that you can install if you are running Windows. It ships with Groovy's Windows installer, or you can download and install it manually. Scriptom is composed of standard Java classes and native DLLs for both 32-bit and 64-bit architectures. The native code does the heavy lifting needed for COM integration and the Groovy and Java classes provide a dynamic DSL for the components. To test the installation, let's write our first ActiveX Groovy script:

```
import org.codehaus.groovy.scriptom.*
def wshell = new ActiveXObject('WScript.Shell')
wshell.popup('Scriptom is Groovy!')
```

If everything is installed correctly then running the code displays a short message in a native dialog box, as seen in figure 19.1. You run this code just like any other Groovy script: there's no need for classpath changes or anything else.



Scriptom is Groovy!

Figure 19.1: A Native Message.

The Scriptom module uses Jacob (Java COM Bridge), an open source Java/COM bridge that allows you to call COM automation components from Java. Jacob offers a generic API that can be used to access any native object. Scriptom builds on top of Jacob API to provide a more intuitive syntax, similar to the kind that VB programmers are used to. For instance, you can set and read properties and invoke methods using the standard Groovy syntax. For example, Listing 19.2 shows how to instantiate an instance of Internet Explorer, set some properties, and then invoke the Navigate method to display a page.

Listing 19.2: Working with ActiveX Objects.

```
def explorer = new ActiveXObject('InternetExplorer.Application')
explorer.Visible = true
explorer.AddressBar = true
explorer.Navigate('http://groovy.codehaus.org')
```

Most Microsoft applications can be automated with Scriptom using a COM interface. Access, Excel, FrontPage, Notepad, and the all the other members of the Microsoft Office suite can be manipulated with Scriptom. In addition to applications, several utilities available on the Windows platform let you interact with the operating system in a simple fashion. This is handy when your automation tasks include activities such as reading and writing keys in the registry, sending keystrokes to running applications, or popping up file dialogs, which is shown in Listing 19.3.

Listing 19.3: Working with ActiveX Objects.

```
def PARENT = 0
def OPTS = 0
def sh = new ActiveXObject('Shell.Application')
def folder = sh.BrowseForFolder(PARENT, 'Choose a folder', OPTS)
println "Chosen folder: ${folder.Items().Item().Path.value}"
```

With the `Shell.Application`, you can call the `BrowseForFolder` method, which shows a file-chooser widget to allow you to select a directory. The `PARENT` and `OPTS` values are the parent window (where 0 means there is no parent) and the option flags to use, respectively. On the last line, you can see that the method returns an object representing a file selection. On this object, you can call the `Items` method to retrieve the selected files and `Item` to select the chosen one. This item has a property called `Path` to retrieve the path of the chosen file. Finally, `value` is a Groovy property that lets you unmarshal the value of the `Path`.

You might be wondering how you can know which methods and properties are available on a given native object or application. Unfortunately, you will have to dive into the documentation of the application you are driving and see what is available through its exposed APIs. For instance, for Microsoft applications, the best source of information is the Microsoft Developer Network (MSDN) website: http://msdn.microsoft.com/library/.

The ability to script running applications is one side of the story, but there's also the other side: Scriptom can receive events when the person in front of the computer clicks buttons, types in text, or executes shortcuts. Also, Scriptom can receive and react to application events, such as reaching the end of a media stream in Windows Media Player. Registering for events is not reasonably straightforward, and the Scriptom website lists the full instructions on how to do so.

Groovy and Scriptom are a powerful combination to bridge two worlds: the Java world with its many free libraries and server-side applications, and Microsoft's platform and its end-user-rich native applications. Scriptom allows you to interact almost intuitively with the host environment to create complex automation tasks and control multiple applications and external Java libraries at the same time.

19.3 GroovyServ for Quick Startup

We've already seen how Grapes and Scriptom make Groovy an excellent choice for a scripting language. However, there is one challenge of the JVM we have not yet addressed: startup time. The JVM takes a relatively long time to startup. You can use the time command along with Groovy from the command line to write a small one-liner to display the current time, along with how long the command took to execute (Listing 19.4).

Listing 19.4: Timing Plain Old Groovy.

```
$ time groovy -e "println new Date()"
Thu Jun 02 13:37:15 CEST 2011

real 0m0.631s
user 0m0.700s
sys  0m0.130s
```

There are two things to notice about the output: 1) The one-liner does indeed print out the current date and time, and 2) the elapsed user-space time to execute this was 0.7 seconds. Compare this with how quickly the same Python script executes on the same machine (Listing 19.5).

Listing 19.5: Timing Python.

```
$ time python -c 'import datetime;print str(datetime.datetime.now()) '
2011-06-02 13:36:46.542847

real 0m0.024s
user 0m0.030s
sys  0m0.000s
```

The Python version takes 0.3 seconds, which is considerably faster. Critics of the JVM point to these startup times and claim that JVM languages are not fit for scripting because of these excesses. Luckily, there is a Groovy project called GrooyServ that fixes this situation. GroovyServ replaces the groovy client application with its own client called "groovyclient". It has the same API and command line parameters, and you can see from the output that its speed is comparable to Python's (Listing 19.6).

Listing 19.6: Timing GroovyServ.

```
time groovyclient -e "println new Date()"
Thu Jun 02 13:46:58 CEST 2011

real 0m0.036s
user 0m0.020s
sys  0m0.000s
```

Why so much faster? groovyclient is only half of the GroovyServ project, the other half is groovyserver. The groovyserver application starts up a JVM as a TCP/IP server and waits for groovyclient applications to connect. When a client connects then the existing JVM is reused to execute the script, which is much faster than starting up a new JVM process. In practice, you only need to know about groovyclient because it automatically starts the server the first time it is needed. So the very first time you use groovyclient the request takes longer, but all subsequent usages are very fast.

GroovyServ is partially native application, not a pure Java application, and it's available for Windows, Mac, and Ubuntu Linux. GroovyServ properly tracks the current working directory, executing your scripts out of the directory from which they were called, which is why a native application is required. GroovyServ should function just like the groovy command even though it is a TCP/IP server. The `System.in`, `out`, and `err` are all properly streamed to the client, and calls to `System.exit()` are sent to the client as well. Environment and classpath variables are also properly propagated from script instance to script instance.

Since GroovyServ behaves the same way as normal groovy, some people create an alias to GroovyServ that replaces their normal groovy command. For Mac and Linux users, add the following line to your profile:

```
alias groovy=groovyclient
```

Windows users can use the doskey command to create aliases:

```
doskey groovy=groovyclient $*
```

There are some limitations you should understand before going so far as to replace the groovy command. Every script does execute in its own `GroovyClassLoader`, but the JVM and `ContextClassLoader` are shared. This means that things like `System.getProperties()` are shared between scripts and metaprogramming changes to classes in one script may affect another. For example, adding a new method to `java.lang.String` makes that new method visible to all future scripts. You can only clear the classloader memory by restarting or killing groovyserver. To do this simply call “groovyserver -r” to restart or “groovyserver -k” to kill it.

Despite these limitations, GroovyServ goes a long way towards overcoming the startup time problems of the JVM. Future versions of Java might one day reduce startup times to a tolerable level, but until then GroovyServ is good enough and usable enough to be a simple solution to the problem. The next technology we'll look at is Gradle, which can help you with all sorts of automation and deployment concerns.

19.4 Gradle for Project Automation

So far we've seen several approaches for making Groovy a more effective scripting and task automation language. But clearly, Gradle is the *must have* application for project automation on the Groovy platform. Gradle is a project build system designed to allow simple project to have simple, convention based build, while still supporting the most complex builds for those that need it. Gradle's motto is, “make the simple things easy and the complex things possible.”

The build script for Gradle builds is a Groovy-based Domain Specific Language (DSL), which allows you to write builds in either a declarative or imperative manner, as well as just writing plain old Groovy code whenever you need it. Gradle integrates easily with Maven repositories for dependency management, supports multi-project and multi-artifact builds, has a rich plugin system, and is based around a real object-oriented domain model for projects. The easiest way to see the power of Gradle is with some examples. Listing 19.7 builds a full Groovy project, integrating with Maven repositories for finding dependencies.

Listing 19.7: Groovy Build Script (build.gradle).

```
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    groovy 'org.codehaus.groovy:groovy:1.8.0'
    testCompile 'junit:junit:4.8.2'
}
```

This is the entire contents of a typical build.gradle file. You can run it from the command line by simply typing `gradle build`. This script applies the Groovy plugin, declares Maven Central as a dependency repository, and then configures the versions for Groovy and JUnit. With this build you get all the standard build targets like `clean`, `build`, `check` (run the unit tests), and `assemble` (build the jar files), along with several more. The build conventions are the same as in Maven: put the production source code in the `/src/main/groovy` directory, test source in the `src/test/groovy` directory, and any resources in the `/src/main/resources` directory.

If you are going to build a jar file, then it's sensible to set a version number and include a manifest in the .jar file. Gradle allows you to specify this declaratively within your build file. You simply need to add the content of Listing 19.8.

Listing 19.8: Building a jar file

```
version = '1.0'
jar {
    baseName="mySample"
    manifest {
        attributes  'Implementation-Title': 'My Sample',
                    'Implementation-Version': version
    }
}
```

You can build the jar file from the command line by typing `gradle assemble`, and the build will produce a file named mySample-1.0.jar. Inside the jar is the correct MANIFEST.MF file. If you need to build a .war file then use the `war` plugin and the .war file will be generated for you.

Typical modern builds, especially in the enterprise, don't just test and assemble jar files, but they also upload them to a repository so that others can use the new code. Gradle includes a standard `uploadArchives` task for this, and you should configure the task to know where to copy the new files. In Listing 19.9 we publish to a local directory, but it's easy to publish to a remote location or several locations at once.

Listing 19.9: Publishing a jar file

```
uploadArchives {
    repositories {
       flatDir(dirs: file('my_repository'))
    }
}
```

After running `gradle uploadArchives` you'll see that mySample-1.0.jar was copied to the `my_repository` directory.

No whirlwind tour of Gradle is complete without mentioning the Gradle Wrapper. Gradle knows how to download and install itself on client machines so that there is no need for users to ever install Gradle once you have written your build file. This is perfect for Continuous Integration servers because there is nothing to install or configure on the remote machines. It's also convenient for open source projects where many users build the software infrequently and don't want long setup times. Perhaps more importantly than the convenience factor, using the Gradle wrapper ensures a consistent environment for everyone on your team. The wrapper guarantees that all developers are using the same version of Gradle without having to install multiple versions yourself. To enable the Gradle Wrapper for your build, you need to add the wrapper task to your script, run the task once, and then check the results into version control. The task is fairly short and only changes when you want to upgrade Gradle:

```
task wrapper(type: Wrapper) {
    gradleVersion = '0.9.2'
}
```

Run the wrapper once using the `gradle wrapper` command. This creates several files on disk that need to be checked in: gradle-wrapper.jar, gradle-wrapper.properties, and the gradlew.bat and gradle shell scripts.

Now any user can run the wrapper for any build task by typing `gradlew` instead of `gradle`. The wrapper will download and install Gradle, and then run any targets the user has invoked.

There are many Gradle features that can't be covered in this short space, and the topic deserves its own book. Fortunately the online and free documentation is excellent, and includes a lengthy user guide, several cookbooks and tutorial-style documents, and more. There are many features to explore, such as multi-project and multi-artifact builds, the Gradle Daemon (to increase performance), parallel unit test execution, multiple language integration, and dozens of plugins. If you need an automated Groovy build then Gradle is one of the best products to consider.

That's the end of the scripting and automation technologies. In the next section, we change gears and take a look at a static analysis tool that can help your Groovy code stay maintainable and of high quality.

19.5 CodeNarc for Static Code Analysis

The CodeNarc project analyzes Groovy code and warns you of possible defects, bad practices, dead code, or just poor Groovy style. It's a flexible system based around rules that find violations in your code, and it generates reports so you can fix problems either before checking code into version control or before the release. Consider the following innocent looking Groovy script, and then we'll see what CodeNarc thinks of it:

```
Map map = [a: 1, b: 2, "$c": 3, 'b': 4 ];
```

CodeNarc finds four violations in this one small example. Try to find them yourself before reading on. Ready for the answers?

- Duplicate Map Key – The map literal includes the key 'b' twice. The value set in a Map must be unique, so the resulting Map instance only contains three elements instead of the four specified.
- GString as Map Key – The element “$c” is a GString, which should never be used as Map keys. The `hashcode` of a GString is unstable, so you may not be able to find this element again!
- Unnecessary Semi-colon – The line ends in a semi-colon, which is unnecessary in Groovy.
- Unused Variable – The variable `map` is never used after being created.

This small example shows a good range of the issue types CodeNarc can catch. A duplicate map entry and an unused variable are examples of dead, or meaningless, pieces code. They're probably not bugs, but they could be masking a subtle problem where the code isn't exactly doing what you think it should. The GString as map key is almost always a bug and should never be used in code.

The unnecessary semi-colon is a style issue. Semi-colons appear frequently with new developers used to working with Java code. The rules about style issues, like this one, are used to help you convert to writing more Groovy code and relying less on Java idioms and practices.

Let's see a more advanced example that highlights the power and intelligence of CodeNarc. Listing 19.10 shows a closure that doubles all the values of a map.

Listing 19.10: Doubling Map Values.

```
def doubleMapValues = { map ->
    if (map == null) { return null }
    if (!map) { return [] }
    return map.values().collect { it * it }
}
assert [1, 4, 9] == doubleMapValues([a: 1, b: 2, c: 3])
```

CodeNarc produces two violations for the `doubleMapValues` closure. One is a simple style issue and the other is a more subtle error.

- Unnecessary Return Keyword – The last line of the method includes the return keyword, which is unnecessary in Groovy.
- Return null Instead of Empty Collection – If passed null the closure returns null. This means the user of the API has to perform null checks on the method result. It is a better practice to return an empty list when there is no result rather than a null.

What makes this an interesting example is that the `doubleMapValues` closure does not specify a return type, yet CodeNarc was smart enough to infer that the closure does return a collection and the closure could also return null. CodeNarc analyzes the return paths of dynamically typed methods and closures and attempts to infer their type.

There are over 200 rules in CodeNarc and the list is constantly growing. The rules are grouped into different rulesets, or categories, such as basic, design, concurrency, security, exceptions, and others. There are also framework-specific rules, such as rules targeted at Grails or the Spock Framework. One of the most interesting categories is the concurrency ruleset. Concurrency is easy to get wrong, and there are many bad practices that can be found automatically. While the Groovy language provides some nice shortcuts but it is always good to understand the fundamentals. Consider the concurrency related example in Listing 19.11.

Listing 19.11: Using @Synchronized.

```
class Person {
    List addresses

    @groovy.transform.Synchronized
    void setAddresses(List addresses) {
        this.addresses.clear()
        this.addresses.addAll(addresses)
    }
}
```

The violation generated for this code is “Inconsistent Property Synchronization”. The method `setAddresses` is synchronized, but the method `getAddresses` is not. Remember, Groovy generates a getter and a setter for each property, and this code has a synchronized setter but the hidden getter is not synchronized. The problem is subtle and clearly needs correcting. And while you’re creating the getter, remember to return a copy of the internal List so the code remains thread-safe.

CodeNarc can be run in a variety of ways. There is a command line runner that is simple to get working for small projects, and there are also Maven, Gradle, and Ant plugins so you can run CodeNarc as part of your regular build process. Also, Grails and Griffon users have a CodeNarc plugin that automatically runs against the codebase. But the simplest way to get CodeNarc up and running is to run it as a unit test from a GroovyTestCase. All of these methods are fully documented on the CodeNarc website at http://codenarc.sourceforge.net/. CodeNarc's output is either text, XML, or HTML. You can use the default HTML reports or define your own style sheets. Configuring CodeNarc, choosing which rules to run, and changing rule properties can all be done via Groovy markup, XML, or a plain text properties file, and again the CodeNarc website contains complete documentation. If you receive false positives or simply want to ignore some violations you're always free to apply the standard `java.lang.SuppressWarnings` annotation on classes or methods.

CodeNarc is a mature and positive addition to the Groovy ecosystem. It can be used by teams to ensure high code quality and consistency, and used by single developers to help migrate to the Groovy way of coding. CodeNarc is like a good pair programming partner, making recommendations when needed and being quiet when not. Since it's so easy to add to Groovy projects, why not give it a try?

The next project we'll review is also focused on code quality. The GContracts project allows you to follow an interesting design approach that encourages you to think about object interactions and contracts.

19.6 GContracts for Improved Design

The GContracts project brings the concepts of Design by Contract™ to the Groovy language. Design by Contract (DbC) is a software design approach in which specify how components interact with each other. The unique part of DbC is that the specifications, or contracts, are defined as source code within the program, rather than simply in documentation. Creating classes, fields, and a public API is one way to specify a contract within Groovy. DbC extends your design capabilities by allowing you to specify class invariants, method preconditions, and method postconditions. Listing 19.12 shows these contracts applied to a kettle object. For those unfamiliar, a kettle heats water, and you can either add water to a kettle or pour water out of a kettle.

Listing 19.12: Using GContracts' @Invariant.

```
@Grab('org.gcontracts:gcontracts-core:1.2.3')
import org.gcontracts.annotations.*

@Invariant({ waterVolume >= 0; waterVolume <= maxVolume })
class Kettle {
    int waterVolume = 0
    int maxVolume = 1000

    // ...
}
```

The example starts by grabbing the latest version of GContracts from Maven Central using the `@Grab` annotation, and declaring the Kettle class with two properties: `waterVolume` (the current amount of water in the kettle) and `maxVolume` (how much the kettle can hold). The interesting part of Kettle is the `@Invariant` annotation. This specifies logic that must always hold true for the object. It must be true after the constructor is called or after any method is invoked. Here the invariant states the water volume cannot be negative and the water volume cannot be more than the maximum volume of the kettle. `@Invariant` is an extension of the type system, and you can define how your type behaves using whatever Groovy code you like. If the invariant is ever violated then an exception is thrown from the object. There should be no way for a programmer to end up with an object whose invariant is violated. Beyond `@Invariant`, GContracts also provides `@Requires` and `@Ensures` annotations, which can be applied to methods (Listing 19.13).

Listing 19.13: Using GContracts' @Requires and @Ensures Annotations.

```
@Requires({ amount > 0 })
@Ensures({ waterVolume == maxVolume || waterVolume > old.waterVolume })
void addWater(int amount) {
    waterVolume = Math.min(maxVolume, amount + waterVolume)
}
```

The `addWater` method from the listing adds water to the kettle, making sure not to overflow the container. The `@Requires` code is a statement about what must be true *before* this method is called: the method parameter (`amount`) must be greater than zero. Violating the `@Requires` precondition produces an exception. The `@Ensures` code is a statement about what must be true *after* this method has been called: the water volume must be at the maximum level (`waterVolume == maxVolume`) or the volume must be greater than whatever the volume was at the beginning of the method call (`waterVolume > old.waterVolume`).

You may be wondering where `old` comes from in the expression `old.waterVolume`. The old variable is a snapshot, or copy, of the object's state before the method call. You also have access to the return value of the method using the `result` variable, as seen in listing 19.14.

Listing 19.14: Using GContracts' result Value.

```
@Requires({ desiredAmount > 0 })
@Ensures({  result >= 0;
            result == 0
                ? waterVolume == old.waterVolume
                : waterVolume < old.waterVolume
})
int pour(int desiredAmount) {
    int amountPoured = (desiredAmount <= waterVolume
            ? desiredAmount
            : waterVolume)
    waterVolume = waterVolume - amountPoured
    amountPoured
}
```

The pour method attempts to pour water from the kettle, returning the amount poured (`amountPoured`) to the user as an int. You can see in the `@Ensures` code that the result will always by zero or greater (`result >= 0`) and the final `waterVolume` of the kettle will be less than or equal to the original `waterVolume`. This `@Ensures` code is a combination of two Groovy statements separated by a semi-colon. You can put as much code as you'd like within the annotation parameters, either chaining all the expressions together with &&, ||, and parentheses, or by separating them with semi-colons. Any valid Groovy code is a valid contract expression.

Contracts can be inherited from parent types. If you specify a contract on an interface or parent class, then all implementations and subclasses inherit that contract. You can also finely control when to apply the contracts. The JVM has several assertion enabling and disabling mechanisms built in, and GContracts honors those settings. Passing -da to the JVM disables all assertions and -ea enables all assertions. Also, you can enable and disable assertions based on package name. Lastly, any contracts you write for objects appear in the generated Groovydoc for that object.

Design by Contract™ is a well-respected design approach that is often envied by the Java community. At one point, adding DbC was the highest voted issue in Sun's Java issue tracker. GContracts brings the core DbC features to Groovy, and Groovy's flexibility allows you to write contracts in a clean and code-centric way. GContracts is definitely a project to check out.

Next up is the jewel in the crown of Groovy: Grails. If you write web applications then you owe it to yourself to discover Grails.

19.7 Grails for Web Development

So far we've looked at a few libraries and applications that make working in Groovy a more productive experience. Now we're going to look at a few application development platforms, starting with Grails, that make writing and deploying full applications a breeze.

Grails is a platform for writing Groovy web applications, and at its core is a Model-View-Controller (MVC) design based on Spring, database persistence on top of Hibernate, view templating with SiteMesh, lots of project and deployment automation with Ant and Gant, and a heavy dose of meta programming. The underlying technology is mature and stable enough for the needs of any enterprise environment, and the use of Groovy as a language within all tiers of the MVC make it a pleasure to work with. Seeing Grails in action is the best way to appreciate it.

Getting a web application up and running requires running a few command line statements and editing one or two files. Grails handles almost all the hard work for you. After installing Grails, you create an application using the `create-app` command, as shown in Listing 19.15. For a quick prototype, we'll create a simple contacts manager application that lets you add, edit, and delete contacts from a list.

Listing 19.15: Creating a Grails application.

```
$ grails create-app contacts
Welcome to Grails 1.3.7 - http://grails.org/
Licensed under Apache Standard License 2.0
...
Created Grails Application at /home/hdarcy/contacts
cd contacts
contacts $
```

This script creates a new application template for you, and configures your environment with reasonable defaults, such as an in-memory database for development and an acceptable looking set of CSS style-sheets. At this point, you could run the application successfully, but there would not be much to see without defining any models, views, or controllers. Creating these is a simple step and again done from the command line, as shown in Listing 19.16.

Listing 19.16: Creating a domain object and a controller.

```
contacts $ grails create-domain-class contacts.Person
...
Created DomainClass for Person
Created Tests for Person
contacts $ grails create-controller contacts.Person
...
Created Controller for Person
Created Tests for Person
contacts $
```

As you can see from the output, not only were classes created, but also unit tests. All these files are on disk, and in order to see a meaningful contacts application we'll need to give the Person domain class a few properties and constraints, and also tell the controller to provide the standard create, read, update, and delete actions for the Person object. Listing 19.17 shows the updated Person class.

Listing 19.17:The /grails-app/domain/contacts.Person.groovy domain object.

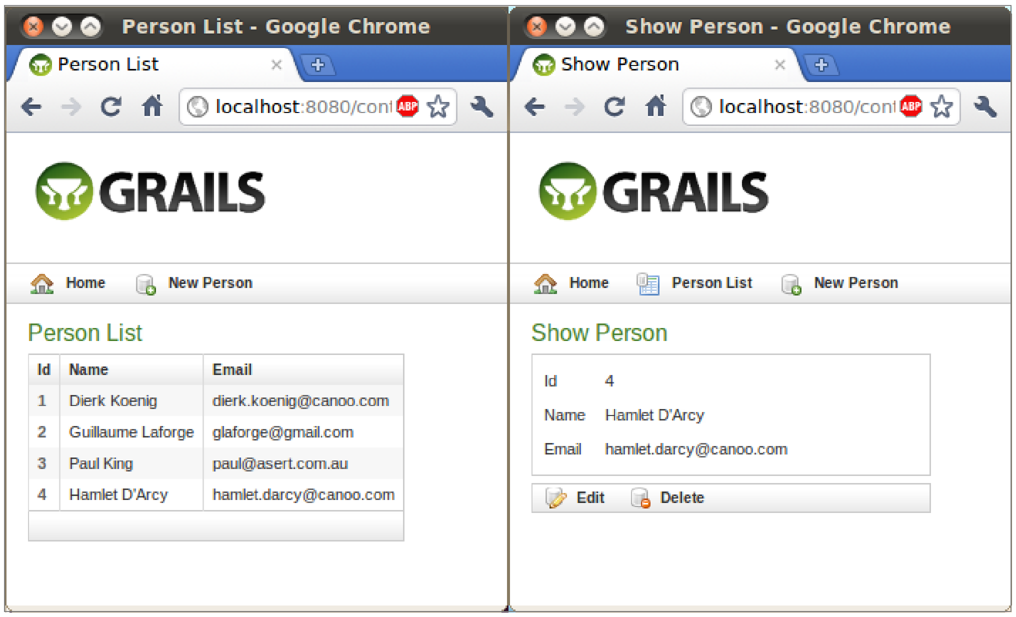
```
package contacts

class Person {
    String name
    String email

    static constraints = {
        name(blank: false)
        email(email: true)
    }
}
```

The application, when run, reads the properties from the object and displays a default user interface based on the class. The constraints DSL is a way to specify validations for your properties. In this case, a name is required, and the email field must be in a format for an email address. There are many more options for constraints and you can also customize them yourself. The last step before running the app is to tell the controller to provide the default scaffolding, as shown in Listing 19.18.

Listing 19.18: The /grails-app/controller/PersonController.groovy scaffolding.

```
package contacts

class PersonController {
    def scaffold = Person
}
```

With this in place, we can run the app (using the `grails run-app` command) and get a reasonable user interface for a Person. You can view all the people, add new people, edit an existing person, and delete a person. The web page displays an error if any of the domain constraints are violated, making sure your data is always consistent. Figure 19.2 shows the Person list and the detail view for a single person, all of which Grails generated for us.



Person List

Id	Name	Email
1	Dierk Koenig	dierk.koenig@canoo.com
2	Guillaume Laforge	glaforge@gmail.com
3	Paul King	paul@asert.com.au
4	Hamlet D'Arcy	hamlet.darcy@canoo.com

Show Person

Id	4
Name	Hamlet D'Arcy
Email	hamlet.darcy@canoo.com

Figure 19.2: Generated list and detail view.

There is much more you can do with Grails, such as define custom view pages, override the default controller behavior, or declare complex relationships between domain classes. But one of the most powerful features of Grails is its database interface. To interact with the database you use the Groovy Object Relational Mapping (GORM) interface. In short, Grails automatically provides methods on your domain objects for working with the database. For instance, each domain object has a `save()` method that persists the object to the database, and the domain classes have a dynamic query API built into them. You can use the Grails console (run with `grails console`) or the Grails shell (run with `grails shell`) to try queries interactively. Listing 19.19 shows GORM in action.

Listing 19.19: Accessing a database with GORM

```
import contacts
new Person(name: 'Dierk', email: 'dierk@canoo.com').save()   //|#1
new Person(name: 'Hamlet', email: 'hamlet@canoo.com').save() //|#1

def people = Person.findAllByEmailLike('%canoo%')    //#2
assert people.size() == 4
def person = Person.findByEmailLikeAndNameLike('%canoo%', 'Ham%')  //#3
assert person instanceof Person
```

#1 Create Persons
#2 Find Multiple Persons
#3 Find One Person

The listing starts in #1 by creating two Person objects and persisting to the database (or at least the Hibernate cache) using the `save()` method provided by Grails. But the real magic is in the dynamic finders on the Person class. The `findAllByEmailLike` at #2 and `findByEmailLikeAndNameLike` method at #3 are dynamically created at runtime. You can use any of the properties from your domain class to invoke a finder method like this, and many other comparators are supported, such as `between`, `lessThan` and `notEqual`. The full DSL is one of the most powerful features of Grails.

We've only covered the basics of working with Grails at the command line. There are many more commands available, and they can be listed with the `grails help` command. Some important commands are `grails test-app` (which runs all the tests), `grails war` (which creates a .war file suitable for deployment), `grails create-service` (which creates a service, allowing you to modularize and decompose your application), and most importantly `grails install-plugin`.

Internally, Grails is based on a plugin architecture. GORM itself is a plugin, for example, and it can be replaced with a non-relational database like Gemfire or Hadoop if you like. There are over 600 plugins available to be downloaded and installed, and the number is sure to grow even larger in the future.

Certain plugins are essential for a non-trivial application, such as the Spring Security Core plugin (`install-plugin spring-security-core`) which provides your application with role based security for controller actions and URLs. The Quartz plugin (`install-plugin quartz`) provides job scheduling so you can run regular tasks. The Searchable plugin (`install-plugin searchable`) adds easy search integration from the user interface for your domain classes, and the Mail plugin (`install-plugin mail`) lets your app send mail to users or administrators. There are many, many more to choose from.

Grails is the premier web application platform within the Groovy community, and is growing in usage as more developers see the productivity gains and make the switch. If you're writing web applications then Grails is a must-know platform. This section is only the smallest taste of the power of Grails, and many topics were skipped entirely. If you're interested to know more, then we strongly suggest you pick up one of the many books devoted solely to Grails, such as Grails in Action from Manning Publications.

Grails is a great framework, but not everybody develops web applications. If you like Grails but write desktop applications, then Griffon is the framework for you. We'll look at that next.

19.8 Griffon for Desktop Applications

Griffon is an application development platform for desktop applications, and the goal of Griffon is to bring all the benefits of Grails to desktop developers. Griffon started life as a fork of the Grails codebase, so many of the conventions and features are exactly the same between the two platforms. Today Griffon is definitely its own beast, is evolving in parallel with Grails, with its own distinct and active community.

The core concepts behind Griffon are MVC groups, services, events, and plugins. Applications are divided into several model-view-controller groups, and MVC groups can themselves be composed of other MVC groups. We'll make one ourselves to see how it works. Services are a way to move shared functionality into a component, similar to how they are used in Grails. The Griffon events system allows you to send and receive events between components, both synchronously and asynchronously. Events can be application lifecycle events, like “starting up” and “shutting down”, or you can define your own in-application events. Finally, plugins are a way to bundle and deploy reusable functionality.

To demonstrate the power of Griffon, we'll create an email client that allows you to send emails through a Gmail account. The main window lets you type in some typical email fields, and pressing Send sends the email through a Gmail account. Figure 19.3 shows the finished application.

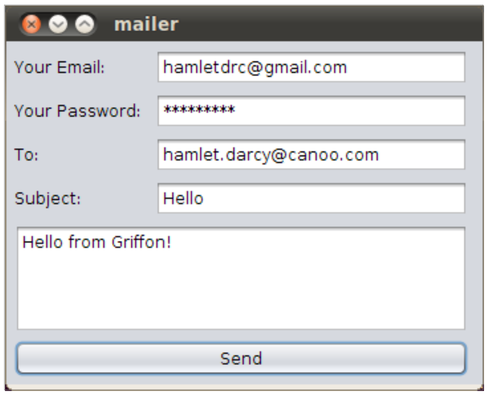


Figure 19.3: A simple Griffon application.

To get started, you create a Griffon app in the same was as a Grails app, using the create-app command. To get started we'll also install two plugins: MigLayout for easier form layout and Mail for SMTP mail integration. Listing 19.20 shows these commands.

Listing 19.20: Creating a Griffon App and Installing Plugins.

```
$ griffon create-app mailer
...
$ cd mailer
...
mailer $ griffon install-plugin mail
...
mailer $ griffon install-plugin miglayout
...
mailer $
```

At this point we have a basic “Hello World”-style desktop application with a single model, view and controller. We'll write the controller code first that sends emails. Controllers have public closures that are invoked from the user interface. The controller has an automatic reference to both the view and the model, and by default controller actions are executed on a separate thread, off of the UI thread. Listing 19.21 shows the code that sends an email using the Mail plugin.

Listing 19.21: Sending email in /griffon-app/controllers/mailer/MailerController.groovy.

```
package mailer

class MailerController {

    def model
```

```
        def action = {
            sendMail(transport: 'smtps', auth: true,
                mailhost: 'smtp.gmail.com',
                user:     model.yourEmail,
                from:     model.yourEmail,
                password: model.yourPassword,
                to:       model.to,
                subject:  model.subject,
                text:     model.text)
        }
    }
```

The `sendMail` method is provided automatically by the plugin, and the rest of the parameters are mostly moving data from our data model to the service call. The model does not yet have all of these properties, and we need to add them next. A Griffon model is not a domain model, but is an application model. An application model allows the view and controller to exchange data, whereas a domain model is a way to describe the concepts and entities in your system. For example, an application model might have a field called “enabled” or “busy”, while a domain model is more concerned with being a higher level description of the system. Listing 19.22 shows our application model.

Listing 19.22: The application model in /griffon-app/model/mailer/MailerModel.groovy.

```
package mailer

import groovy.beans.Bindable

class MailerModel {
   @Bindable String yourEmail
   @Bindable String yourPassword
   @Bindable String to
   @Bindable String subject
   @Bindable String text
}
```

The `@Bindable` annotation exists so that properties can be automatically bound to widgets. When a widget value (like a text box) changes then the bound domain object is automatically updated. There is no need to manually write any `PropertyChangeListener` code: Griffon handles it all for you.

The last piece of the puzzle is the view (Listing 19.23). The view layer is a DSL for Swing components. You can declaratively specify the layout of the form and supply constraints. In this case we're using MigLayout to achieve proper alignment of components.

Listing 19.23: The view in /griffon-app/views/mailer/MailerView.groovy.

```
package mailer

import net.miginfocom.swing.*

application(title: 'mailer', pack: true) {

    migLayout(layoutConstraints:'wrap 2', columnConstraints:'[left][fill]')

    label('Your Email:')
    textField(text:bind(target:model, 'yourEmail'))         //#1
    label('Your Password:')
    passwordField(text:bind(target:model, 'yourPassword'))    //#1
    label('To:')
    textField(text:bind(target:model, 'to'))      //#1
    label('Subject:')
    textField(text:bind(target:model, 'subject'))    //#1
    textArea(text:bind(target:model, 'text'), rows: 6, columns: 30, //#1
            constraints: 'span, grow, wrap')
    button(text: 'Send', actionPerformed: controller.action,
            constraints: 'span, right')                         //#2
 }
```

#1 Property Binding
#2 Button to Controller Wiring

The property binding for the widgets is within the bind method calls at the #1 annotations, and wiring a button to a controller action is as simple as adding the `actionPerformed: controller.action` parameter to the button at #2. The user interface for our mailer is displayed when you launch it with the `griffon run-app` command.

Griffon automates much of the application life-cycle, especially around deployments and packaging. Griffon has built-in support for generating Java Web Start (jnlp) applications, applets, and stand-alone apps. Additionally, the Installer plugin can be used to create native installers for a variety of platforms, such as Windows, Mac, and various flavors of Linux. Griffon also handles the dirty work of signing jar files; your application can be securely signed after placing your credentials in the correct configuration files. You can still use Griffon even if you want to commit to writing Java code. Many of the artifacts, such as the controllers and services can still be written in plain old Java.

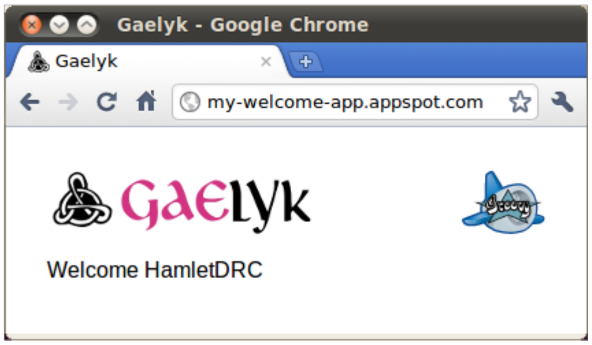
Desktop developers should take a long look at using Griffon for their next project. Griffon provides a strong design by basing applications on MVC groups, and plugins and services allow applications to naturally decompose into small, reusable pieces. The short-term benefits of using the many plugins and project automation scripts are obvious, but Griffon apps have a long-term advantage as well. The Griffon Way is a blueprint for well-factored and maintainable long-term desktop apps.

The next project we'll look at is Gaelyk, a framework for building lightweight web applications on top of Google App Engine. Gaelyk is a good choice for simpler web applications that benefit from a cloud datastore and free, easy deployments.

19.9 Gaelyk for Groovy in the Cloud

Gaelyk is a lightweight yet powerful framework designed for running Groovlets and Groovy Templates Pages in the cloud using Google App Engine. With Gaelyk you'll have access to all the GAE services like the data store, task queue, and Jabber API, and you'll also benefit from the power of using Groovy as a templating engine to generate your web site. Whether you need to generate HTML for a user interface or JSON for an ajax server, Gaelyk has you covered.

To demonstrate Gaelyk and GAE, we'll build a simple hello-world style HTML site that integrates with Google authentication, shown in Figure 19.4. Once you have the basics of security, routing, and Gaelyk's take on MVC, then you should have an easy time moving on to harder tasks like working with the datastore.



Welcome HamletDRC

Figure 19.4: A simple Gaelyk application.

There is a little installation work to do before getting up and running with GAE and Gaelyk. The first step is to download and install the Google App Engine for Java SDK; you should check the Gaelyk website to see which version is supported by Gaelyk. After that you'll need to register with App Engine and create an application. You register at and select an app name at http://appengine.google.com/. Our sample app is named “my-welcome-app”. The last part of the setup is to download and unzip the Gaelyk template project from the Gaelyk website.

Now that everything is installed, we can run the application locally to make sure that everything is working correctly. Gaelyk and GAE come with many helpful scripts that automate running and deploying apps. The following code snippet shows how to build and run the app locally:

```
$ groovy build.groovy
$ dev_appserver.sh
...
INFO: The server is running at http://localhost:8080/
```

At this point you can open your browser and see the standard welcome page of a Gaelyk app. With the installation verified, it's time to configure the application to use our ID and enable security. The app ID is defined in the file appengine-web.xml. Open this file with a text editor and write your app ID into the <application> tag, like so:

```
<application>my-welcome-app</application>
<version>1</version>
```

Also notice the version number. Over time, you'll want to increase this number each time you make a deployment or release. GAE lets you run several versions of your app at once, so managing the version number lets you test new versions in the cloud while your users continue to use the stable release.

Now let's update the web.xml to enable security. App Engine uses a web.xml file, which is the standard way to define servlets, filters, and security constraints within a Java application server. By default, a Gaelyk site is public and open to anyone. We'll want to enable security so that users are required to be logged in through Google. Open the web.xml file and copy in the security description from listing 19.24.

Listing 19.24: Enabling security in war/WEB-INF/web.xml.

```
<security-constraint>
    <web-resource-collection>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>*</role-name>
    </auth-constraint>
</security-constraint>
```

The GAE website has more information about the security constraints, but for now this is all we need. Anyone visiting our web page will need to be signed into Google first, and GAE handles the redirects and authentication for us.

At last we're ready to start coding some application logic. Gaelyk is based on a model-view-controller pattern that separates the user interface logic into .gtpl template files and business logic into Groovy based controllers. The key is in URL routing. As a URL as accessed, Gaelyk internally redirects that request to your controller, which performs logic and then renders a template view. In our simple app we're going to route all traffic to the "/" URL to the "welcome.groovy" controller. That controller will access the currently logged in user information and then forward the information to the view template. The view template just needs to print out a welcome message.

First step: edit the routes.groovy table to forward requests to our controller by adding the following line:

```
get "/", forward: "/welcome.groovy"
```

This configures the container so that any HTTP GET request is forwarded to welcome.groovy. The next step is, of course, to define the welcome controller. The controller is where you would normally access the data store, start tasks, or perform other complex logic. Our controller is quite simple; it just exposes the current user in the request and renders the default view, which is done with two lines of code:

```
request.currentUser = user
forward 'index.gtpl'
```

The last piece of the puzzle is editing the index.gtpl view. This is a Groovy Template page, and by default the text is markup. You tag syntax is similar to Java Server Page (JSP) syntax, and you can include scripts using the <% %> or the ${} notation:

```
<% include '/WEB-INF/includes/header.gtpl' %>
<p>Welcome ${request.currentUser.nickname}</p>
<% include '/WEB-INF/includes/footer.gtpl' %>
```

That's the basics of Gaelyk. We can build and test locally using the groovy build.groovy and `dev_appserver.sh` war commands. Google even provides mock account authentication for testing. When you're satisfied that everything is working locally then it's time to deploy to the cloud. Run the command appcfg.sh update war to push the deployment to GAE. You'll be prompted for your Google credentials, and then after a short wait you'll be able to access your application using the public App Engine URL.

This tutorial presents the bare minimum functionality of Gaelyk, but there is a lot more. Anything you can do with GAE is accessible with Gaelyk: the data store, task queue, Jabber, image and file services, and more. GAE is a great way to get apps up and running in the cloud, and Gaelyk is the best way to use Groovy to do so. The community is active and more Gaelyk apps are being deployed all the time. Now is a great time to give it a whirl.

The last library we'll look at is Groovy++, which bring static typing and speed improvements to the Groovy language. It's an ambitious project that you can use today to get both performance gains and better compile time checking of your Groovy code.

19.10 Groovy++ for Safety and Performance

The last stop on the Groovy ecosystem tour is the ambitious Groovy++ project. The aim of this project is to bring static, compile-time type checking to Groovy, which can improve performance and provide early error checking in the compiler, while leaving the flexible syntax and key features of Groovy the way they are.

One of the periodic criticisms of Groovy is that it's slow when compared with other languages using micro-benchmarks. It *is* slower when compared to statically typed JVM languages like Java and Scala. But it's also sometimes slower compared to other dynamic languages like JRuby and Python. There are several ways to address this complaint. First, the bottleneck of a slow system is often database access rather than application code. Even if Groovy is slow, it doesn't make sense to optimize Groovy out of your system if it isn't the real pain point of your overall system performance. Second, if application code really is your slowest point, then you can always rewrite those components in Java. But for the Groovy++ authors this was not a good enough response. They wanted to use the Groovy syntax, especially closures, but have the generated .class files be as fast as Java. They wanted to write Groovy code which would transformed into optimized bytecode by the time it was seen by the JVM. Currently, the project achieves some impressive results. Consider the code in Listing 19.25, which uses Groovy++ to sum a whole bunch of numbers.

Listing 19.25: Summing Numbers in Groovy++.

```
@Typed                    //#1
package example

@Grab('org.mbte.groovypp:groovypp-all:0.4.248_1.8.0')   //#2
@Grab('org.mbte.groovypp:groovypp:0.4.248_1.8.0')
import groovy.lang.Grab

long result = 0
for (long x = 0; x < 100000000; x++) {
    result = result + x
}

assert result == 4999999950000000L
```

#1 Static Typing Annotation
#2 Groovy++ Imports

This example runs in less than 0.5 seconds on my machine. It's not very Groovy code (it looks a lot like Java), and it took a while to come up with something so de-optimized, but it serves as a good example because there is very little dynamic about it. A compiler should be able to optimize this to be very fast. If you comment out the @Typed annotation at #1, then this becomes plain old Groovy code and is no longer Groovy++. Under Groovy this example takes almost 8 seconds to complete – about 16 times as long as the Groovy++ version. Impressive results, and close to the 0.2 seconds achieved with plain old Java. With Groovy++ you get many speed improvements but are still free to continue using closures, categories, and the Groovy GDK methods that you've come to rely on.

Performance is just one of the many aspects of Groovy++. The project also attempts to bring static type inference to the compiler. The code in Listing 19.26 shows code that compiles, runs, and executes with no problems in Groovy.

Listing 19.26: Groovy and Groovy++ Type Inference.

```
def str = "A string"
println str.toLowerCase()
str = 5     // reassigning a string variable to be an int!
println  str
```

This example prints "a string" and "5" due to Groovy's type coercion of Integers to Strings. In Groovy++ this code fails to compile, because the variable "y" is known to be a String but an integer is being assigned to it on line 3. Groovy++ is a statically typed language, and it does not have the same dynamic type coercion that Groovy has. This brings safety to your program, but you sacrifice some flexibility.

Static typing is not an all-or-nothing choice in Groovy++. If you annotate a package, class, or method with @Typed, then that unit is compiled by Groovy++. Also, if you give a file the .gpp file extension then the entire file is statically compiled. But Groovy and Groovy++ interact with each other nicely. You can use Groovy++ in the files you need and plain old Groovy when you want a more dynamic system. In fact, you can even mix and match the typing within the same file, as in Listing 19.27.

Listing 19.27: Groovy and Groovy++ Type Inference.

```
@Typed                     //#1
class MyClass {

    int getHtmlSize() {
        String html = buildDynamicHtml()
        html.length()
    }

    @Typed(TypePolicy.DYNAMIC)       //|#2
    def buildDynamicHtml() {         //|#2
        def writer = new StringWriter()
        new groovy.xml.MarkupBuilder(writer).root {  //#3
            child()
            child()
        }
        writer.toString()
    }
}

assert 38 == new MyClass().htmlSize
```

#1 Static Typing Annotation
#2 Dynamically Typed Method
#3 Dynamic Method Call

MyClass is annotated with @Typed at #1, so by default the entire thing is compiled as Groovy++. Type inference is active and the compile fails if you try to invoke dynamic features. However, the buildDynamicHtml method at #2 is marked as @Typed(TypePolicy.DYNAMIC), so within that method you are free to use any dynamic feature you want, such as an XML MarkupBuilder. Without the "mixed mode" annotation, this code fails to compile because there is no method on MarkupBuilder called root() at #3. Clearly dynamic and static code can easily live side by side.

There is a lot to Groovy++ beyond what you've seen here. It enables tail recursion, and also extension methods, which allow you to add dynamic methods to statically compiled classes. The project is often a bit of a playground for many new Groovy ideas because static typing allows a lot of features to be implemented that would be impossible in plain old Groovy. Some of the ideas in Groovy++ may one day make it back to core Groovy, but certainly not all of them will.

The main point of Groovy++ is speed, optimized bytecode, and compile-time type safety. At this point in time the project achieves these goals. If you want to continue using Groovy, but are having performance issues, then Groovy++ is certainly a library worth considering. However, remember that performance optimizations should only be put into place once you have measured and profiled your application and determined the real bottleneck. In real world Groovy usage, many developers find that Groovy is *not* the cause of a performance problem.

19.11 Summary

Thus concludes our whirlwind tour of the Groovy Ecosystem: those projects that are not exactly part of Groovy itself but are essential for being a productive Groovy programmer.

If you're using Groovy as a scripting and automation language then consider mastering Grapes, Scriptom, GroovyServ, and Gradle. Grapes is a great way to easily manage script dependencies and makes sharing easy. Scriptom provides a good way to automate Windows specific work. For frequent scripters, GroovyServ can speed up the startup time a noticeable amount. And Gradle is an import technology for not only building your Groovy and Java projects, but also for project automation in general.

On the surface, CodeNarc and GContracts are different technologies. CodeNarc is focused on finding and preventing bugs in your code and helping with the enforcement of coding standard. GContracts brings Design by Contract™ to Groovy, allowing you to design objects and interactions based on the expected contract of those objects. But the two technologies are similar in that both are focused on improving the overall quality of a system written in Groovy.

For web application developers, Grails is an important technology to master because it uses Groovy in several unique ways in order to make web apps fast to write, easy to maintain, and a joy to work on. And Griffon does the same for desktop applications. For cloud developers, Gaelyk is a good platform for running Groovy on Google App Engine. You can get up, running, and deployed quickly with minimum investment.

Lastly, Groovy++ is an exotic experiment with big ambitions and already big successes. This project shows just how malleable Groovy is: a new language extension was written on top of it. Groovy++ shows that when it comes to the limitations of Groovy, the true limit is not enforced by the technology but rather by your imagination. The tools presented in this chapter are useful, but there is still plenty of room for innovation in the Groovy ecosystem. So go ahead and try something new, whether it's one of these technologies or one of your own inventions.