SAMPLE CHAPTER

Groovy IN ACTION

Dierk König

with Andrew Glover
Paul King
Guillaume Laforge
Jon Skeet

Foreword by **James Gosling**

MANNING

Groovy in Action

by Dierk König

with Andrew Glover, Paul King

Guillaume Laforge, and Jon Skeet

Sample Chapter 2

brief contents

1 ▪ Your way to Groovy 1

PART 1 THE GROOVY LANGUAGE 27

2 ▪ Overture: The Groovy basics 29
3 ▪ The simple Groovy datatypes 55
4 ▪ The collective Groovy datatypes 93
5 ▪ Working with closures 122
6 ▪ Groovy control structures 153
7 ▪ Dynamic object orientation, Groovy style 174

PART 2 AROUND THE GROOVY LIBRARY 227

8 ▪ Working with builders 229
9 ▪ Working with the GDK 277
10 ▪ Database programming with Groovy 323
11 ▪ Integrating Groovy 360
12 ▪ Working with XML 401

PART 3 EVERYDAY GROOVY 451

13 ▪ Tips and tricks 453
14 ▪ Unit testing with Groovy 503
15 ▪ Groovy on Windows 546
16 ▪ Seeing the Grails light 572
appendix A ▪ Installation and documentation 606
appendix B ▪ Groovy language info 610
appendix C ▪ GDK API quick reference 613
appendix D ▪ Cheat sheets 631

2 Overture: The Groovy basics

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, later in the performance, were revisited, extended, varied, and combined. 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; however, 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.

Overtures allow 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/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 support for *Groovydoc* is in the works at the time of writing. It will be the Groovy equivalent to Javadoc and will use the same syntax.

Comments, however, are not the only Java-friendly part of the Groovy syntax.

2.1.2 Comparing Groovy and Java syntax

Some Groovy code—but not all—appears exactly like 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 the classic Java *for(init;test;inc)* loop. As you will see in listing 2.1, even language semantics can be slightly different (for example, with the `==` operator).

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 and method definitions (except for nested classes)
- Control structures (except the classic *for(init;test;inc)* loop)
- Operators, expressions, and assignments
- Exception handling
- Declaration of literals (with some twists)
- Object instantiation, referencing and dereferencing objects, and calling methods

The added value of Groovy's syntax is to

- Ease access to the Java objects through new expressions and operators
- Allow more ways of declaring objects literally
- Provide new control structures to allow advanced flow control
- Introduce new datatypes together with their operators and expressions
- Treat *everything* as an object

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, less verbose, 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'
```

Not only is the Groovy code shorter, but it expresses our objective in the simplest possible way. 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.

In chapter 7, we will also talk about optional `return` statements.

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.

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 parameters are valid. In this book, however, 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.[1]

```
assert 1 == 1
```

This demonstrates that `assert` can take full expressions, not just literals or simple variables. Unsurprisingly, `1` equals `1`. Exactly like Ruby and 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.

Assertions serve multiple purposes:

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

REAL LIFE A real-life experience of the value of assertions was writing this book. 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 raw version of this book in MS-Word format that contains no code, but only placeholders

[1] Groovy's meaning of *truth* encompasses more than a simple boolean value, as you will see in section 6.7.

that refer to files containing the code. With the help of a little Groovy script, all placeholders are scanned and loaded with the corresponding file, which is evaluated and replaces the placeholder. For instance, the assertions in listing 2.1 were evaluated and found to be correct during the substitution process. The process stops with an error message, however, if an assertion fails.

Because you are reading a production copy of this book, that means the production process was not stopped and all assertions succeeded. This should give you confidence in the correctness of all the Groovy examples we provide. Not only does this prove the value of assertions, but it uses 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 do something with 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.

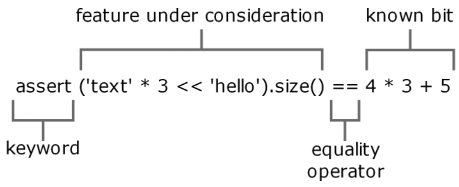
feature under consideration
known bit
assert ('text' * 3 << 'hello').size() == 4 * 3 + 5
keyword
equality operator

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. Section 6.2.4 covers assertions in 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 move you far toward the goal of writing 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 enable you to start experiencing Groovy *on your own*. We encourage you to experiment—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 blueprint from which objects are drawn.

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.[2]

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'
```

[2] 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*.

```
String getTitleBackwards(book) {
    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 other 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 how to organize larger script-based applications. In Groovy, the preferred way is not meshing together numerous script files, but instead grouping reusable components in 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.

Programming the `Book` class and the script that uses it was simple. It's hard to believe that it can be any simpler, but it *can*, as you will 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 entity on its own. It's a concept made up from a naming convention. If a class exposes methods with the naming scheme `get`*`Name`*`()` and `set`*`Name`*`(`*`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.)

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

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.

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.

Listing 2.4 Defining the `Book` class as a GroovyBean

```
class Book {
    String title        <─┐ Property
}                         │ declaration

def groovyBook = new Book()                                   │ Property use
                                                              │ with explicit
groovyBook.setTitle('Groovy conquers the world')             <│ method calls
assert groovyBook.getTitle() == 'Groovy conquers the world'   │

groovyBook.title = 'Groovy in Action'              │ Property use with
assert groovyBook.title == 'Groovy in Action'      │ Groovy shortcuts
```

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.

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

2.3.4 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 = 'Gina'
def book = 'Groovy in Action'
assert "$nick is $book" == 'Gina is Groovy in Action'
```

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 available methods and operators 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 provides a means for easy declaration of regular expression patterns as well as 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.

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/`.

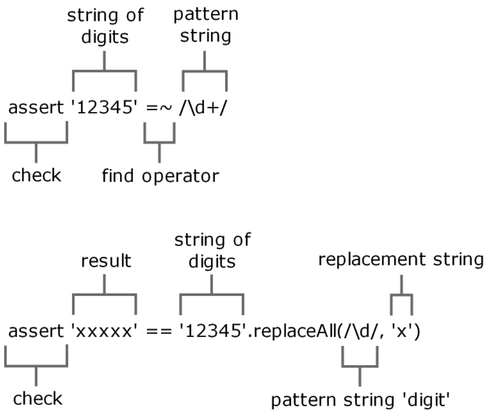

Figure 2.2 Regular expression support in Groovy through operators and slashy strings

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

2.3.5 Numbers are objects

Hardly any program can do without numbers, whether for calculations or, more often, for counting and indexing. Groovy *numbers* have a familiar appearance, but unlike in Java, they are first-class objects, *not* 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 ubiquitously used 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.

2.3.6 Using lists, maps, and ranges

Many languages, including Java, directly understand only 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 to use 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.

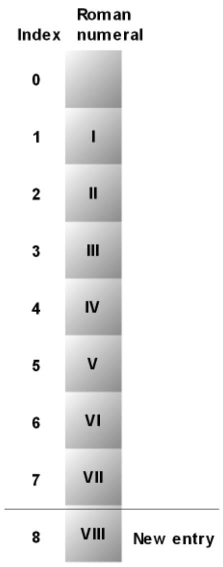

Figure 2.3 An example list where the content for each index is the Roman numeral for that index

The list is constructed such that each index matches its representation as a Roman numeral. Working with the list looks much like working with arrays, 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']   <— List of Roman numerals

assert roman[4] == 'IV'   <— List access

roman[8] = 'VIII'   <— 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. Later, in section 4.2, we will discuss more capabilities of the `list` datatype.

Simple maps

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

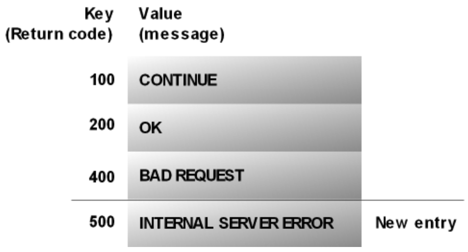

Figure 2.4 An example map where HTTP return codes map to their respective messages

The following example stores descriptions of HTTP[3] return codes in a map, as depicted in figure 2.4.

You 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'   ]
```

[3] 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.

```
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 the wealth of their Groovy 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 a notion of how to move from the start to the end point. 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.

2.3.7 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]`.

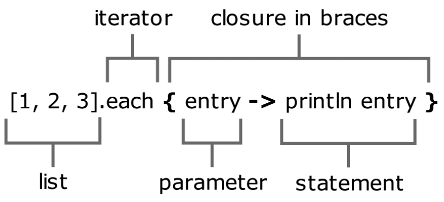

Figure 2.5 A simple example of a closure that prints the numbers 1, 2 and 3

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?[4]

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

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

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.

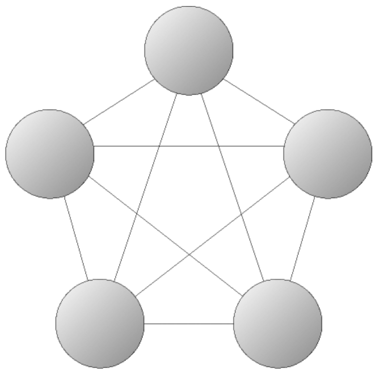

Figure 2.6 Five elements and their distinct connections, modeling five people (the circles) at a party clinking glasses with each other (the lines). Here there are 10 "clinks."

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

Listing 2.5 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
}

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
int totalClinks = 0;
for(int guestNumber = 1;
```

[5] 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.

```
        guestNumber <= partyPeople;
        guestNumber++) {
    int clinksWithGuest = guestNumber-1;
    totalClinks += clinksWithGuest;
}
```

Note that `guestNumber` appears four times in the Java code but only two times 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.

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

2.3.8 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*; not-*null* is treated like *true*. 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 2.6 Control structures

```
if (false) assert false      <-- if as one-liner

if (null)      <-- Null is false
{
    assert false      <-- Blocks may start on new line
}
else
{
    assert true
}

def i = 0
while (i < 10) {      Classic while
    i++
}                        Classic
assert i == 10           while

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

def list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for (j in list) {                                 for in
    assert j == list[j]                           list
}

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

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

The code in listing 2.6 should be self-explanatory. Groovy control structures are reasonably close to Java's syntax. Additionally, you will find a full introduction to Groovy's control structures in chapter 6.

That's it for the initial syntax presentation. You got your feet wet with Groovy and 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 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.[6]

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 identical at the *byte-code level*.

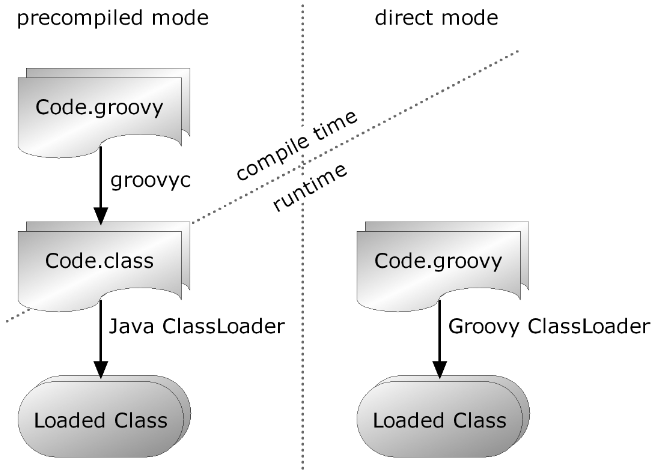

Figure 2.7 Groovy code can be compiled using `groovyc` and then loaded with the normal Java classloader, or loaded directly with the Groovy classloader

[6] We hope the Groovy programmers will forgive this oversimplification.

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

NOTE 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.

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

[7] 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.

Table 2.1 Various ways of determining sizes in the JDK

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

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 loads the class from the *.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 can *at runtime* read *.groovy files 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.

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.[8]

[8] This doesn't exclude *replacing* a class at runtime, when the .groovy file changes.

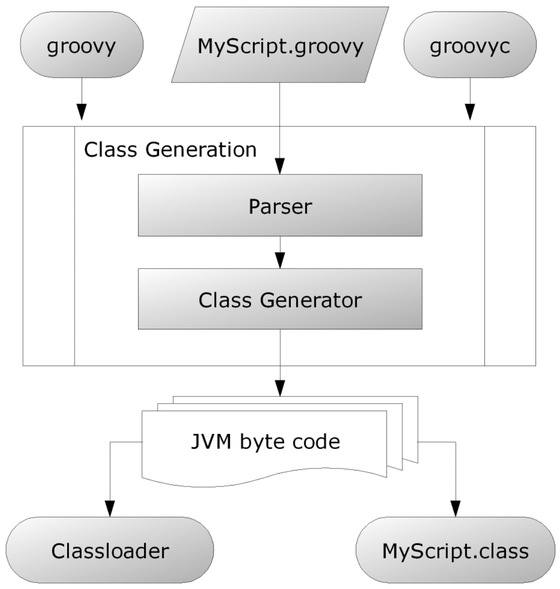

Figure 2.8
Flow chart of the Groovy bytecode generation process when executed in the runtime environment or compiled into class files. Different options for executing Groovy code involve different targets for the bytecode produced, but the parser and class generator are the same in each case.

Given this description, you can 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 the ability to seemingly modify classes at runtime—for example to add new methods. But as you just learned, Groovy generates classes once and cannot change the bytecode after it has been loaded. How can you add a method without changing the class? The answer is simple but delicate.

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[9]

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

[9] The actual implementation involves a few more redirections.

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

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

A second option of dynamic code is putting the code in a string and having Groovy evaluate it. You will see how this works in chapter 11. Such a string can be constructed literally or through any kind of logic. But beware: 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('os.version')
println code                    // Prints "1 + 5.1"
println evaluate(code)          // Prints "6.1"
```

Note that `code` is an ordinary string! It happens to contain `'1 + 5.1'`, which is a valid Groovy expression (a *script*, actually). Instead of having a programmer write this expression (say, `println 1 + 5.1`), 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` then be *executed*? The answer is simple. Remember the left-hand path in figure 2.7? Class generation can transparently happen at runtime. The only news is that the class-generation input can also be a *string* like `code` rather than a *.groovy file.

The capability 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 appears 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*.

JAVA

"... a clear and detailed exposition of what is groovy about Groovy. I'm glad to have it on my bookshelf."

—from the *Foreword* by James Gosling

Groovy IN ACTION

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

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

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

What's Inside

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

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

"Excellent code samples ... very readable."

—Scott Shaw, ThoughtWorks

"Top of my list."

—Samuel Pullara, VP Technology Strategy, Yahoo, Inc.

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

—John Wilson
The Wilson Partnership

"Great, logical focus on language features."

—Norman Richards
JBoss Developer, author of *XDoclet in Action*

"Destined to be the definitive guide. First rate!"

—Glen Smith, Bytecode Pty Ltd

"You want to learn Groovy? This book has all you need."

—Stuart Caborn, ThoughtWorks

54999

9 781932 394849

ISBN 1-932394-84-2

MANNING $49.99 US/$64.99 Canada