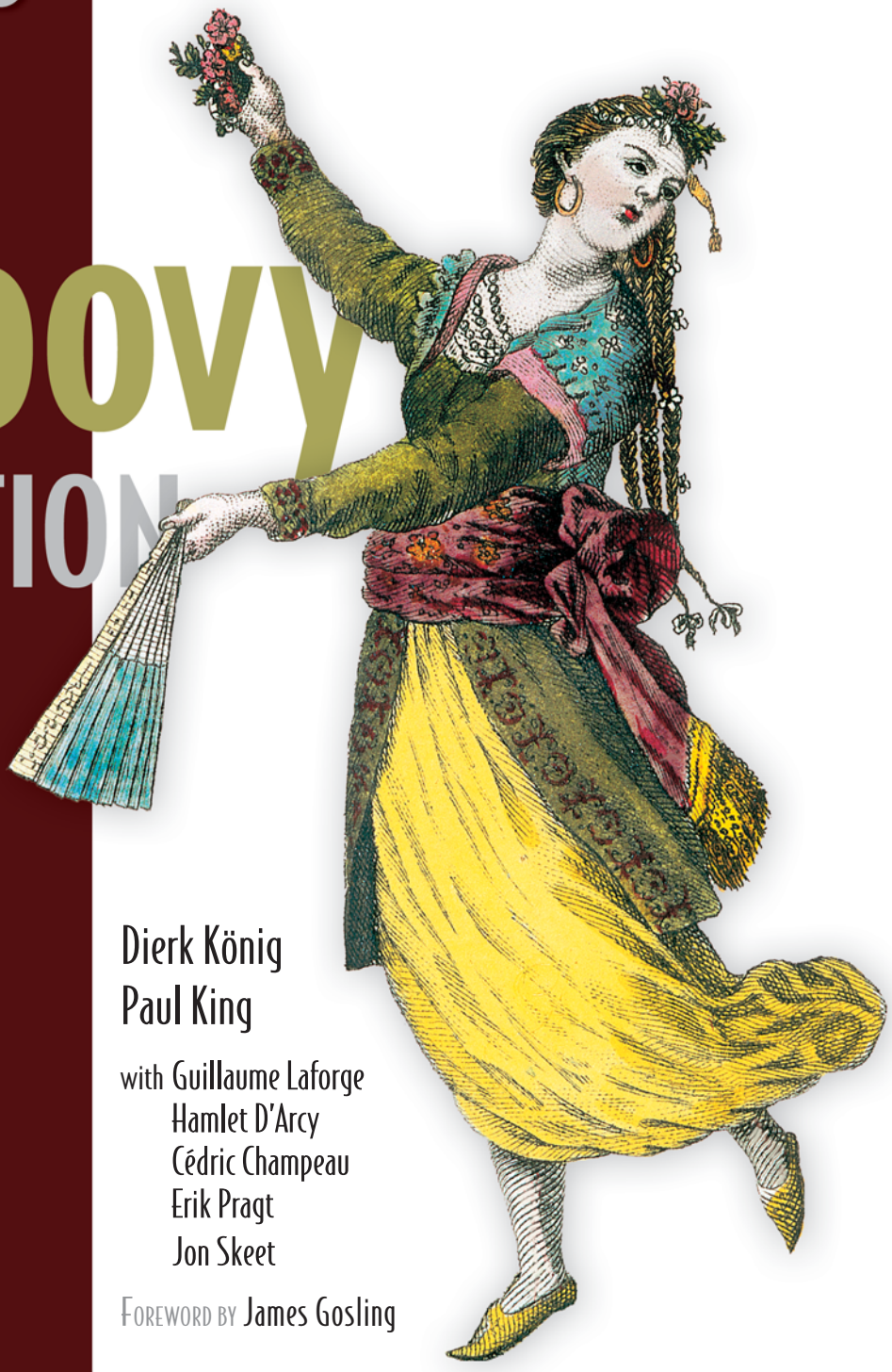
SAMPLE CHAPTER

SECOND EDITION

Groovy IN ACTION

COVERS GROOVY 2.4

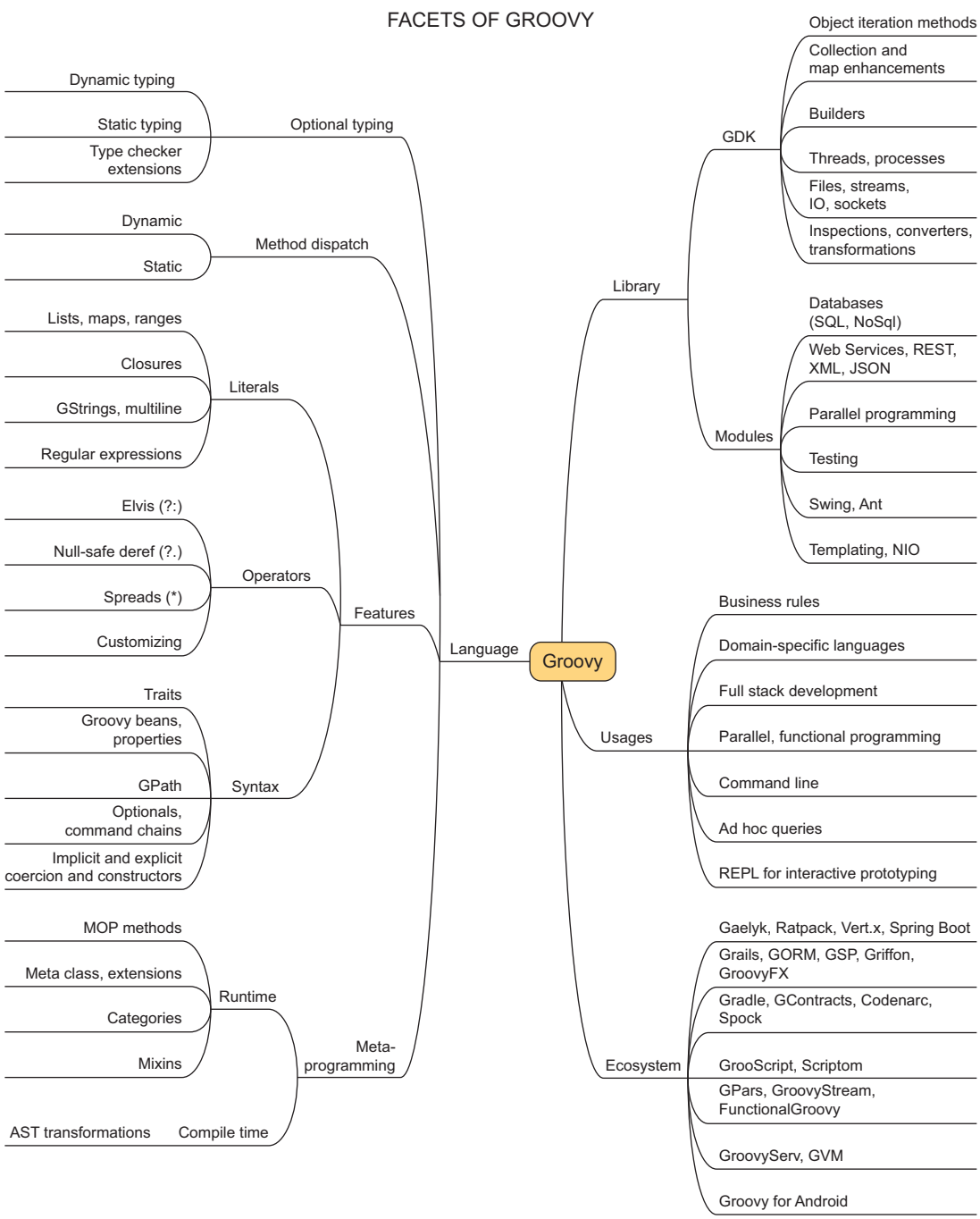


Dierk König
Paul King

with Guillaume Laforge
Hamlet D'Arcy
Cédric Champeau
Erik Pragt
Jon Skeet

FOREWORD BY James Gosling

MANNING


FACETS OF GROOVY
Groovy
Language
Features
Optional typing
Dynamic typing
Static typing
Type checker extensions
Method dispatch
Dynamic
Static
Literals
Lists, maps, ranges
Closures
GStrings, multiline
Regular expressions
Operators
Elvis (?:)
Null-safe deref (?.)
Spreads (*)
Customizing
Syntax
Traits
Groovy beans, properties
GPath
Optionals, command chains
Implicit and explicit coercion and constructors
Meta-programming
Runtime
MOP methods
Meta class, extensions
Categories
Mixins
Compile time
AST transformations
Library
GDK
Object iteration methods
Collection and map enhancements
Builders
Threads, processes
Files, streams, IO, sockets
Inspections, converters, transformations
Modules
Databases (SQL, NoSql)
Web Services, REST, XML, JSON
Parallel programming
Testing
Swing, Ant
Templating, NIO
Usages
Business rules
Domain-specific languages
Full stack development
Parallel, functional programming
Command line
Ad hoc queries
REPL for interactive prototyping
Ecosystem
Gaelyk, Ratpack, Vert.x, Spring Boot
Grails, GORM, GSP, Griffon, GroovyFX
Gradle, GContracts, Codenarc, Spock
GrooScript, Scriptom
GPars, GroovyStream, FunctionalGroovy
GroovyServ, GVM
Groovy for Android

Groovy in Action
Second Edition

by Dierk König
Paul King
with
Guillaume Laforge
Hamlet D'Arcy
Cédric Champeau
Erik Pragt
and Jon Skeet

Chapter 2

brief contents

PART 1 THE GROOVY LANGUAGE.....1

1 ■ Your way to Groovy 3
2 ■ Overture: Groovy basics 28
3 ■ Simple Groovy datatypes 54
4 ■ Collective Groovy datatypes 91
5 ■ Working with closures 117
6 ■ Groovy control structures 145
7 ■ Object orientation, Groovy style 164
8 ■ Dynamic programming with Groovy 200
9 ■ Compile-time metaprogramming and AST transformations 233
10 ■ Groovy as a static language 294

PART 2 AROUND THE GROOVY LIBRARY.....341

11 ■ Working with builders 343
12 ■ Working with the GDK 401
13 ■ Database programming with Groovy 445

14 ■ Working with XML and JSON 506
15 ■ Interacting with Web Services 543
16 ■ Integrating Groovy 561

PART 3 APPLIED GROOVY 603

17 ■ Unit testing with Groovy 605
18 ■ Concurrent Groovy with GPars 650
19 ■ Domain-specific languages 676
20 ■ The Groovy ecosystem 732

2 Overture: Groovy basics

This chapter covers

- 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 weave euphonious patterns that are revisited, extended, varied, and combined later in the performance. In a way, overtures are the whole symphony *en miniature*.

In this chapter, we introduce many 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 jumpstart 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's almost indistinguishable from Java and transition smoothly into a more lightweight, suggestive, idiomatic style as your knowledge of the language grows. We'll 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.

Groovy is *indentation-unaware,* but it's 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. Groovy currently doesn't support multiple initialization and iteration statements in the classic `for(init1,init2;test;inc1,inc2)` loop. As you'll 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 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 includes the following:

- 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.
- Use annotations to generate invisible code, the so-called AST transformations that are described in chapter 9.
- Introduce new datatypes together with their operators and expressions.
- A backslash at the end of a line escapes the line feed so that the statement can proceed on the following line.
- Additional parentheses force Groovy to treat the enclosed content as an expression. We'll use this feature in section 4.3 when we cover more of the details about maps.

Overall, Groovy looks like Java, except more compact and easier to read thanks to these additional syntax elements. 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's shorter and more *expressive*. Compare the Java and Groovy code for encoding a string for use in a URL. For Java:

```
java.net.URLEncoder.encode("a b", "UTF-8");
```

For Groovy:

```
URLEncoder.encode 'a b', 'UTF-8'
```

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're 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 doesn't 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'll 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 that are optional in Groovy too:

- In chapter 7, we'll talk about optional *return* statements.
- Even the ubiquitous dot becomes optional when the chaining method is called. For example, in combination with optional parentheses, the following code is legal in Groovy: `buy best of stocks`, which is short for `buy(best).of(stocks)`. Chapter 7 has the full description of these so-called command chains.
- 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 your belt, we can look at the principal tool you're going to use to test each new piece of the language: assertions.

2.2 Probing the language with assertions

If you've worked with Java 1.4 or later, you're 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, for performing tasks such as checking preconditions at the beginning and postconditions and invariants at the end of a method, or for 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'll 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. The following listing makes some simple assertions.

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 or Scala but unlike Java, the `==` operator denotes *equality*, not *identity*. The parentheses were left out as well, because they're 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 nothing was revealed about the *type* of `x`. The `def` keyword means "dynamically typed."

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

This is the typical style 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've seen before, it's optional when the statement ends with the current line.

[1] Groovy's meaning of truth encompasses more than a simple Boolean value, as you'll see in "The Groovy truth" in chapter 6.

What happens if an assertion fails? Let's see![2] For example:

```
def a = 5
def b = 9
assert b == a + a
```

Expected to fail

prints to the console (yes, really!):

```
Assertion failed:

assert b == a + a
       | |  | | |
       9 |  5 | 5
         |    10
       false
```

Expression retained

Referenced values

Subexpression values

```
        at snippet22_failing_assert.run(snippet22_failing_assert.groovy:3)
```

Pause and think about the language features required to provide such a sophisticated error message. You'll see more examples of Groovy's "power assert" feature when we discuss unit testing in chapter 17.

Assertions serve multiple purposes:

- They can be used to reveal the current program state, as they're used in the examples in this book. The one-line assertion in the previous example 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, at this point, it's 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 example

One real-life example of the value of assertions is in your hands right now (or on your screen). This book is constructed such that all listings and the assertions they contain are maintained outside the actual text and linked into the text via file references. 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. If an assertion fails, the whole process stops with an error message.

The fact that you're reading a production copy of this book means the production process wasn't stopped and all assertions succeeded. This should give you confidence in the correctness of the Groovy examples provided. For the first edition, we did the same with MS Word using Scriptom (chapter 20) to control MS Word, and AntBuilder (chapter 11) to help with the building side. As we said before, the features of Groovy work best when they're used together.

[2] This code is one of the few listings that isn't executed as part of the book production.

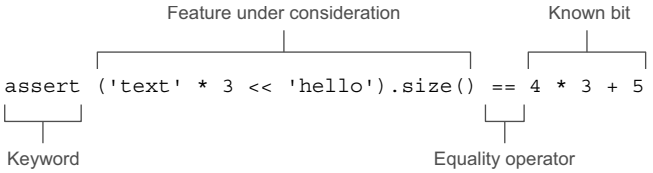


Figure 2.1 A complex assertion, broken up into its constituent parts

Most of the 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'd expect the result to be given your description, and then looking at what you've said the result will be, as checked at runtime by the assertion. Figure 2.1 breaks up a more complicated assertion into its constituent parts.

This is an extreme example—you'll 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 you're trying to demonstrate, and there's code that's trivial and can be easily understood without knowing the details of the topic at hand.

In case assertions don't convince you or you mistrust an asserted expression in this book, you can usually replace it with output to the console. A hypothetical 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 isn't a common way of presenting code in books,[3] 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've explained the tool you'll be using to put Groovy under the microscope, you can start seeing some of the features in use.

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 toward the goal of writing useful Groovy code in the shortest possible amount of time. Instead, we'll present

[3] This was a genuine innovation in the first edition of this book, which was found so useful by other authors that they copied the concept. We don't mind. Everything that advances our profession is welcome.

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'll 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 (OOP), 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'll refer to it in the next section.

The code isn't 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 like this:

```
> groovy myfile.groovy
```

Note that this is very different from Java. In Groovy, you're executing the source code! An ordinary Java class is generated for you and executed behind the scenes. But from a user's perspective, it looks like you're executing plain Groovy source code.[4]

[4] Any Groovy code can be executed this way as long as it can be run; that is, it's either a script, a class with a main method, a `Runnable`, or a Groovy or JUnit test case.

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'll 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. You create a `new` instance and call the getter method on the object by using Java's *dot* syntax. Then you 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 you're able to invoke the method `getTitleBackwards` before it's 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 you 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 isn't 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 you'll see next.

2.3.3 GroovyBeans

JavaBeans are ordinary Java[5] classes that expose *properties*. What is a property? That's not easy to explain, because it's 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`.

[5] This is prior to Java 8 where a new concept of properties as first-class citizens comes bundled with JavaFX 8.

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)
- Simplifying registration of event handlers together with annotations that declare a property as *bindable*

The following listing shows how the `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
}

def groovyBook = new BookBean()

groovyBook.setTitle('Groovy in Action')
assert groovyBook.getTitle() == 'Groovy in Action'

groovyBook.title = 'Groovy conquers the world'
assert groovyBook.title == 'Groovy conquers the world'
```

Property declaration

Property use with explicit getter calls

Property use with Groovy shortcuts

We also demonstrate how to access the bean in the standard way with accessor methods, as well as in 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'll learn more about this construction in chapter 7.

Also note that `groovyBook.title` *is not* a field access. Instead, it's a shortcut for the corresponding accessor method. It'd work even if you'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

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. Listing 2.5 declares an immutable `FixedBean` class, calls the constructor in two different ways, and asserts that you have a standard implementation of `equals()` that supports comparison by content. With the help of a little `try-catch`, you assert that changing the state isn't allowed.

Listing 2.5 Defining the immutable `FixedBean` and exercising it

```
import groovy.transform.Immutable

@Immutable class FixedBook {
    String title
}

def gina  = new FixedBook('Groovy in Action')
def regina = new FixedBook(title:'Groovy in Action')

assert gina.title == 'Groovy in Action'
assert gina == regina

try {
    gina.title = "Oops!"
    assert false, "should not reach here"
} catch (ReadOnlyPropertyException expected) {
    println "Expected Error: '$expected.message'"
}
```

AST annotation.

Positional constructor.

Named-arg constructor.

Standard equals().

Not allowed!

It must be said that proper immutability isn't easily achieved without such help and the annotation does actually much more than what you see in listing 2.5: 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're indispensable for *concurrent programming*: an increasingly important topic that we'll cover in chapter 18.

The `@Immutable` annotation is only one of many that can enhance your code with additional characteristics. In the next section we'll briefly cover the `@Grab` annotation, in chapter 8 we'll look at `@Category` and `@Mixin`, and in chapter 9 we'll cover the full range of other annotations that come with the GDK.

Most Groovy annotations, like `@Immutable`, instruct the compiler to execute an AST transformation. 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 on 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 metaprogramming and isn't limited to the transformations that come with the GDK. You can also provide your own transformations!

2.3.5 Using grapes

Before continuing we should cover one of the other annotations that you'll see in numerous places in the rest of the book. The `@Grab` annotation is used to explicitly define your external library dependencies within a script. We sometimes use the term *grapes* as friendly shorthand for our external Groovy library dependencies. In the Java world, you might store your dependent libraries in a `lib` directory and add that to your classpath and IDE settings, or you might capture that information in an Ivy, Maven, or Gradle build file. Groovy provides an additional alternative that's very

handy for making scripts self-contained. The following listing shows how you might use it.

Listing 2.6 Grabbing external libraries

```
@Grab('commons-lang:commons-lang:2.4')
import org.apache.commons.lang.ClassUtils

class Outer {
  class Inner {}
}

assert !ClassUtils.isInnerClass(Outer)
assert ClassUtils.isInnerClass(Outer.Inner)
```

Here the use of the commons `lang` library is declared. It's used to make some assertions about two classes, ensuring that one of them is an inner class. At compile time and runtime that library will be downloaded if needed and added to the classpath. More details about `@Grab` and numerous related annotations can be found in appendix E.

2.3.6 Handling text

Just as in Java, character data is mostly handled using the `java.lang.String` class. But 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're familiar with the concept of regular expressions, you'll 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'll 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 example ensures that the string contains a series of digits; the second example replaces every digit with an `x`.

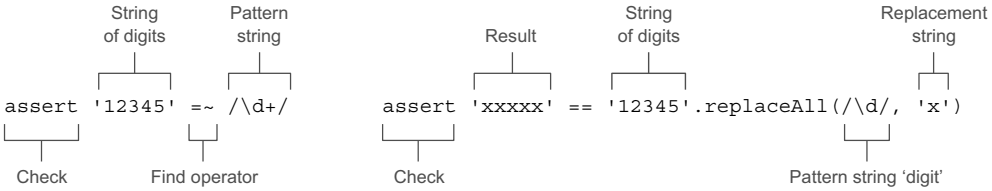


Figure 2.2 Regular expression support in Groovy through operators and slashy strings

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.

2.3.7 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're 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. For example:

```
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, you can use the `plus` method, but you 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'll learn more about how Groovy handles numbers in chapter 3.

2.3.8 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's 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'll call the *subscript operator*. In Groovy the same syntax can 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 aren't 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 example in figure 2.3 declares a list of Roman numerals and initializes it with the first seven numbers.

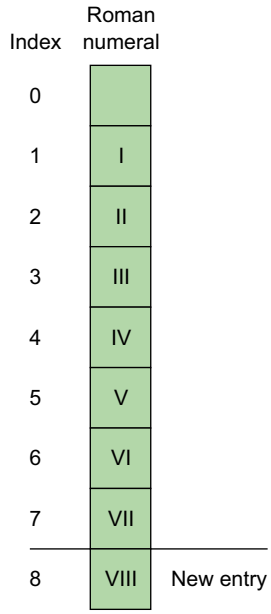


Figure 2.3 An example list where the content for each index is the Roman numeral for that index

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

```
def roman = ['', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII']
assert roman[4] == 'IV'

roman[8] = 'VIII'
assert roman.size() == 9
```

List of Roman numerals

List access

List expansion

Note that there was no list item with index `8` when you assigned a value to it. You 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; 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 example in figure 2.4 stores descriptions of HTTP[6] return codes in a map.

---

[6] The server returns these codes with every response. Your browser typically shows the mapped descriptions for codes above `400`.

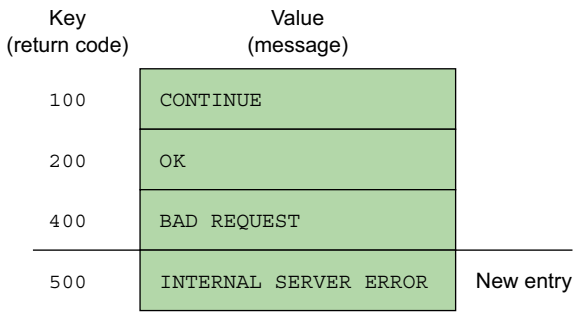


Figure 2.4 An example map where HTTP return codes map to their respective messages

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's 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)
assert x.size() == 10
assert x.from == 1
assert x.to == 10
assert x.reverse() == 10..1
```

These examples are limited because we're 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 uses.

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

2.3.9 Code as objects: closures

The concept of *closures* isn't 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, the sole purpose of which is to implement an appropriate single-method interface. The instances of those types can subsequently 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 and, since Java 8, lambdas and method references to address these issues. Although similar in function, Groovy closures are much more versatile and powerful when it comes to reaching out to the caller's scope and putting closures in a dynamic execution context. Groovy allows closures to be specified 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's 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 braces, like any other code block. It optionally has a list of identifiers to name the parameters passed to it, with an `->` marking the end of the list.

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

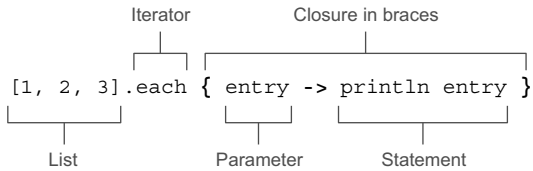


Figure 2.5 A simple example of a closure that prints the numbers 1, 2, and 3

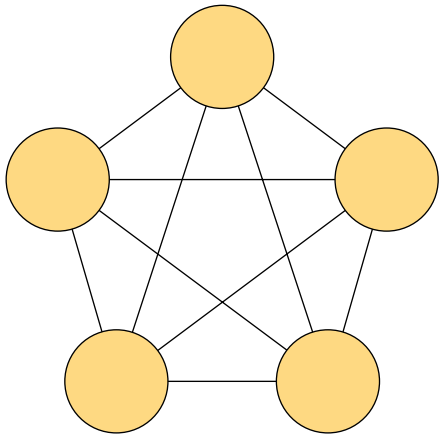


Figure 2.6 Five elements and their distinct connections, modeling five people (the circles) at a party clinking glasses with each other (the lines). Here there are 10 clinks.

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 whatever is passed to the closure, namely the parameter 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?[7] Figure 2.6 sketches this question for five people, where each line represents one clink.

To answer this question, you 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. You 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 2.7 calculates the number of clinks. You keep a running total of the number of clinks, and when each guest arrives, you add the number of people already present (the guest number – 1). Finally, you test the result using Gauss' formula[8] for this problem—with 100 people, there should be 4,950 clinks.

Listing 2.7 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
```

Modifies outer scope

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

[8] Johann Carl Friedrich Gauss (1777–1855) was a German mathematician. At the age of seven, 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 differing reports on how the teacher reacted.

How does this code relate to Java? In Java, you'd have used a loop like the following code 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. But while code duplication itself is bad, it's even more so an indicator for a lack of modularity! Groovy gives you more means to separate your code into its independent concerns such as how to walk a data structure and what to do at each step.

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's pretty tricky to do in Java, even with lambdas in Java 8.[9]

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

2.3.10 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. The following listing gives an overview.

[9] Java pours "syntax vinegar" over such a construct to discourage programmers from using it.

Listing 2.8 Control structures

```
if (false) assert false

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

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

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

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

list.each() { item ->
    assert item == list[item]
}

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

The if as one-liner

null is false

Blocks may start on new line

Classic while

The for in range

The for in list

The each method with a closure

Classifer switch

The code in listing 2.8 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's 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'll examine how Groovy classes enter the Java environment to start with, how Groovy augments the existing Java class library, and 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've compared the appearance of the source code. But the connection to Java is much stronger. Behind the scenes, all Groovy code runs inside the JVM, and follows 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's a `MyClass.groovy` file, it'll 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. (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*.

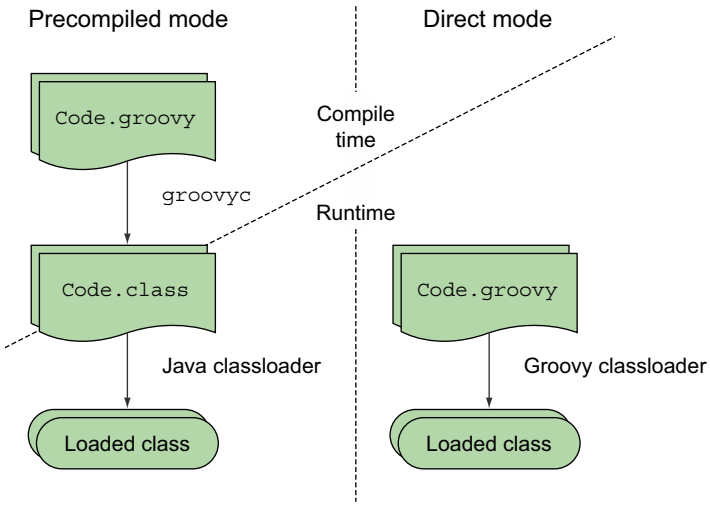


Figure 2.7 Groovy code can be compiled using `groovyc` and then loaded with the normal Java classloader, or loaded directly with the Groovy classloader.

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're the same thing, there's no gap to bridge. In the 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 know 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,[10] and it provides significant benefits in consistency, power, and expressiveness.

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's available on everything that's of some size: strings, arrays, lists, maps, and other collections. Behind the scenes, they're 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'd 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.

[10] 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'll use this acronym anyway, because it's conventional in the Groovy community.

Table 2.1 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

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

To help you understand how Groovy objects can leverage the power of the GDK, we'll next sketch how Groovy objects come into being.

2.4.3 Groovy compiler 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.

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 isn't 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).

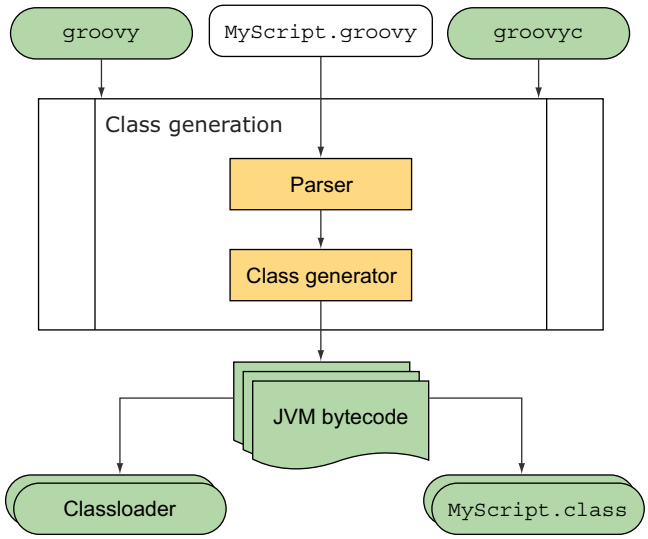


Figure 2.8 Flowchart of the Groovy bytecode generation process when executed in the runtime environment or compiled into *.class files. Different options for executing Groovy code involve different targets for the bytecode produced, but the parser and class generator are the same in each case.

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 you have a Groovy script stored in a file named `MyScript.groovy`, and you run it via `groovy MyScript.groovy`. The following are the class-generation steps, as shown in figure 2.8:

1. The file MyScript.groovy is fed into the Groovy parser.
2. The parser generates an 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 also 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 don't change while running.[11]

[11] 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'll 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's not the dynamic typing that makes a dynamic language dynamic. It's 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 this case recognizes that there's no corresponding method in the bytecode of `Album` and therefore relays the call to its `missingMethod` handler. This knows about the naming convention of Grails' dynamic finder methods and fetches your favorite albums from the database.

But because 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 this:[12]

```
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's no difference.

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

Calling the `MetaClass` for every method call seems to imply a considerable performance hit, and, yes, this flexibility comes at the expense of runtime performance. But this hit isn't quite as bad as you might expect, because 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. When you need near-Java performance, you can even use `@CompileStatic` (see chapter 10) and the generated code is no longer calling into the `MetaClass`.

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')` isn't known at compile time but the compiler has to

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

compile it anyway. Now if you've mistyped the method name by accident, a compiler cannot warn you. In fact, compilers have to accept almost any method call that you throw at them and the code will fail at runtime.[13] But don't 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're doing. It'll warn you on method calls that it cannot resolve and, in the preceding case, it even gives you code completion and refactoring support for Grails's dynamic finder methods.

A way of using dynamic code is to put the source in a string and ask Groovy to evaluate it. You'll see how this works in chapter 16. 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 + 51.0'
assert 52.0 == evaluate(code)
```

Note that `code` is an ordinary string! It happens to contain `'1 + 51.0'` when running the code with Java 7,[14] which is a valid Groovy expression (a *script,* actually). Instead of having a programmer write this expression (say, `println 1 + 51.0`), 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 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's a general-purpose programming language in itself.

GROOVY CAN BE STATIC

Does the dynamic support within Groovy worry you? Do you think it might add performance penalties to your execution? Or do you worry that you might have reduced IDE support when writing your programs? We already told you not to despair because of the excellent tool support available even for Groovy in its most dynamic form. But if you still aren't reassured, you can force the Groovy compiler to do strict type checking (with elaborate type inference) by using the `@TypeChecked` annotation for pieces of code that you know to be free of dynamic features. The type checking mechanism is extensible so you can even provide stricter type checking than available in Java if you want.

---

[13] That is, the code fails at unit-test time, right?

[14] You should expect 49.0 if running using JDK5, 50.0 using JDK6, and 52.0 if using JDK8.

To see a glimpse of this feature, examine the following class definition:

```
class Universe {
  @groovy.transform.TypeChecked
  int answer() { "forty two" }
}
```

If you try to compile this you'll get a compilation error:

```
[Static type checking] - Cannot return value of type java.lang.String
on method returning type int
```

Without the `@TypeChecked` annotation, the code would fail at runtime with a `GroovyCastException`. Chapter 10 has all the details.

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 you to keep the features you're trying to demonstrate and the results of using those features close together within the code. It also lets you automatically verify that the 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'll cover in depth in chapter 8. 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've 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'll leave the surface and dive into the deep sea of Groovy. This may be unfamiliar, but don't worry. We'll return to sea level often enough to take some deep breaths of Groovy code *in action*.

JAVA

Groovy IN ACTION Second Edition

König • King

In the last ten years, Groovy has become an integral part of a Java developer's toolbox. Its comfortable, common-sense design, seamless integration with Java, and rich ecosystem that includes the Grails web framework, the Gradle build system, and Spock testing platform have created a large Groovy community.

Groovy in Action, Second Edition is the undisputed definitive reference on the Groovy language. Written by core members of the Groovy language team, this book presents Groovy like no other can—from the inside out. With relevant examples, careful explanations of Groovy's key concepts and features, and insightful coverage of how to use Groovy in-production tasks, including building new applications, integration with existing code, and DSL development, this is the only book you'll need.

What's Inside

- Comprehensive coverage of Groovy 2.4 including language features, libraries, and AST transformations
- Dynamic, static, and extensible typing
- Concurrency: actors, data parallelism, and dataflow
- Applying Groovy: Java integration, XML, SQL, testing, and domain-specific language support
- Hundreds of reusable examples

Some experience with Java or another programming language is helpful. No Groovy experience is assumed.

Authors **Dierk König, Paul King, Guillaume Laforge, Hamlet D'Arcy, Cédric Champeau, Erik Pragt,** and **Jon Skeet** are intimately involved in the creation and ongoing development of the Groovy language and its ecosystem.

Technical editor: **Michael Smolyak**

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/GroovyinActionSecondEdition

"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, Creator of Java

"Groovy lies between light scripting and heavier enterprise languages—this book will help you master the sweet spot."
—Rick Wagner, Red Hat

"The most valuable Groovy resource, written by the most valuable members of the Groovy community."
—Vladimir Orany
Metadata Consulting Ltd.

"The long-awaited and excellent successor to the first edition."
—David McFarland
Instil Software Ltd.

Free eBook
SEE INSERT

ISBN 13: 978-1-935182-44-3
ISBN 10: 1-935182-44-7
55999
9 781935 182443

MANNING $59.99 / Can $68.99 [INCLUDING eBOOK]