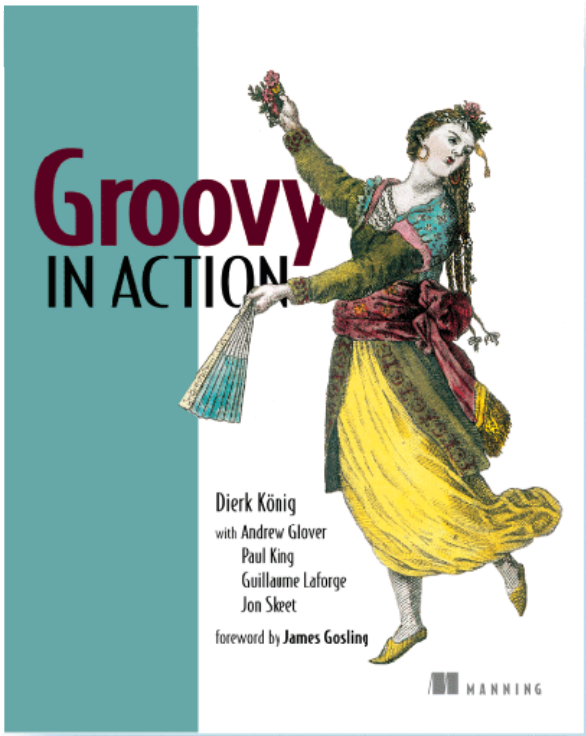
An Excerpt
Groovy in Action
By Dierk Koenig,
with Andrew Glover, Paul King,
Guillaume Laforge, and Jon Skeet
Forward by James Gosling



Excerpt of
Groovy in Action

Chapter 2:
Overture: The Groovy basics

Part 1 of 3

Click here for more information on *Groovy in Action*

This is the first installment of a three-part series based on chapter 2 of Groovy in Action *from Manning Publications. This chapter introduces the language in a high-level fashion. After reading this series, you'll have a solid initial understanding of Groovy fundamentals.*

This excerpt introduces you to the general appearance of Groovy code and its similarities to and differences from Java. It also introduces assertions, a built-in feature of Groovy that is used throughout the book to make the code examples self-checking.

Chapter 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
```

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

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 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.

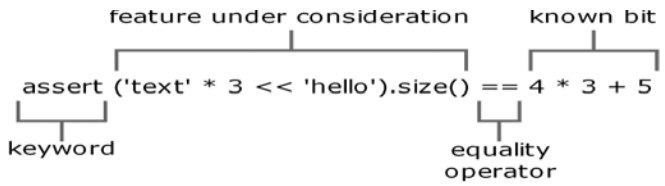


Figure 2.1• A complex assertion, broken up into its constituent parts

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.

This has been part 1 of a three-part series excerpted from the book Groovy in Action *from Manning Publications. The second installment will review the various datatypes for which Groovy has native language support. It will give you a first impression of what makes Groovy so special.*