Practical Groovy

Table of Contents

Chapter 1. Who this book is for

One of the struggles for any author is identifying who the target audience is. Novice or expert? Computer literate or not? In the end, a book cannot satisfy everyone and any attempt to do so will result in failure.

Practical Groovy is designed to be an introduction to the Groovy programming language for anyone that has prior knowledge of an object-oriented language, such as Java or Python. Familiarity with the Java platform will certainly help you, but it's not absolutely necessary. As long as you can install the Java Development Kit, you're good to go.

I envisage two main classes of existing developer using this book:

1. Dynamic language programmers that want to migrate to the Java platform while continuing to use features that they're used to.

2. Java developers that want some of the productivity promised by Groovy while staying on a familiar platform (including the core class library).

The book will take you through lots of practical examples so that you get a good understanding of how Groovy differs from other languages and how you might solve particular problems. It's best read while you are working on a computer so that you can try the examples out as you come across them. It also gives you the opportunity to experiment.

Chapter 2. First steps

In the tech world, it's fairly common to hear the phrase "yak shaving". This represents the kind of situation in which you end up doing a lot of setup work in order to achieve what you originally intended. In the case of Groovy, it normally entails installing the Java Development Kit if you haven't already done so, followed by Groovy itself. Fortunately, you can bypass all that by using the Groovy Web Console, which is a browser-based application for executing Groovy scripts.

You can find the app at the address http://groovyconsole.appspot.com/. You should see the page shown in figure 1.1.

The text editor component allows you to write Groovy that you can then execute. Try it now by entering the code

```
12 + 5
```

and clicking on the Execute script link. You'll see the result "17" displayed in the Result tab.

Let's now try the classic Hello World example:

```
println "Hello world!"
```

Now when you execute the script you'll see the "Hello world!" text displayed in the Output tab. What's the difference between the two tabs? The Result tab displays the value of the last expression in the script, whereas the Output tab displays anything that would normally go to the terminal or command prompt (stdout in C parlance). You can test this out by adding the "12 + 5" line to the end of the Hello World script: you should see something in both the Result tab and the Output tab.

At this point, you have two choices:

1. Continue with the Groovy Web Console for now and skip to section TODO
2. Read the next section on installing Groovy on your own computer

You will want to install Groovy at some point and it's usually a straightforward process. On the other hand, it is an extra step that you may not need just to get started with the language. If you want to be able to work offline or run scripts locally, then you should definitely install Groovy now. Otherwise either option is fine.

2.1. Installing Groovy

Before I explain how to install and set up Groovy, consider this: why would you need to install anything? This is an important question as you don't necessarily need a Groovy installation in order to

develop with Groovy. Build tools and IDEs can quite happily compile Groovy source files without one. What you absolutely must have, though, is a Java runtime installed.

The Java platform comes in several forms, but for the sake of simplicity I want to focus on two of them: the Java Runtime Environment (JRE) and the Java Development Kit (JDK). The first of these allows you to run Java applications and applets (those things that run in a browser and require the Java plugin). The JDK allows you to build Java applications, as well as run them, and it's the best option for a Groovy developer.

To get hold of the JDK, head to the download page on Oracle's website and follow the links. Remember, you only need the basic JDK for your platform. Once you have downloaded the package and installed the JDK (I recommend using an installation path without spaces), you just need to set up an environment variable:

```
JAVA_HOME=<path to JDK>
```

If you're unsure how to do this, a simple web search for "setting environment variable" combined with your platform will produce plenty of links to help out.

That's all you need as far as the Java side goes. So what about Groovy? You'll need to install it for the purposes of this book because you will need access to the associated tools, in particular the Groovy console. You also need a Groovy installation if you want to run standalone scripts, which comes in very handy.

You can install Groovy in a number of ways:

- On Windows, download and run the installer.
- On Mac OS X, you can use Homebrew or MacPorts: `brew install groovy` or `sudo port install groovy`.
- On all platforms that have Bash, including Cygwin, you can use the Groovy Environment Manager (GVM), which allows you to install multiple versions of Groovy as well as other tools such as Grails and Gradle.
- Alternatively, download the platform-agnostic binary distribution, unpack it, and manually set up the necessary environment variables.

My preferred option is GVM as it makes installing Groovy trivially easy and new versions of Groovy (including beta and other milestone releases) become available through it promptly. The fact you can have different versions of Groovy active in different terminals is an added bonus.

Whichever approach you take, you will end up with a `GROOVY_HOME` environment variable containing the location of the current Groovy installation. Your `PATH` environment variable will also include the path to the Groovy installation's 'bin' directory, since that's where all the useful tools are. You can verify the installation by running

```
groovy -e "println 'Hello'"
```

in a terminal. If everything is working, this will println the line "Hello" to the terminal. If you instead get an error message about the `groovy` command not being found, then check your `PATH` environment variable as it's probably not initialized properly.

So what does a Groovy installation give you? The main tools of interest are:

1. `groovy` - allows you to run Groovy scripts from the command line.
2. `groovysh` - starts an interactive shell for executing Groovy statements.
3. `groovyConsole` - starts a desktop version of the Groovy Web Console, with syntax highlighting in the editor pane.

You will start with the Groovy console as it's the easiest way to learn Groovy through practice. Just start it up with the command

```
groovyConsole
```

and you'll then see the window shown in figure 1.3 TODO.

The Groovy console has several options available, including loading and saving scripts as well as the standard cut, copy and paste. The most important options are

- View > Clear Output - clears the output pane of any existing text from previous script executions.
- View > Show Script in Ouput - enabled by default, you should uncheck this so that the output pane isn't polluted with the text of the script itself.
- Script > Run - executes the script.

I recommend you remember the keyboard shortcuts for clearing the output pane and executing the script as you'll be using them a lot.

You're now all set to start coding.

2.2. Groovy as a calculator

I'm sure your OS has a calculator app, as do most smartphones. Still, using Groovy as a calculator is a gentle way to introduce some of the basic syntax and semantics of the language. You already saw a simple addition. What happens if you now try

```
13 + 3 / 4
```

? In most programming languages you would end up with the result 13.75 (although Java would give you 13), and that's exactly what you get with Groovy. In other words, the division has occurred first. To do the sum first, use parentheses to control the order in which the operators are evaluated:

```
(13 + 3) / 4
```

The above results in 4. Groovy supports all the usual suspects when it comes to arithmetic operators, such as `-` (subtract), `*` (multiply), and `%` (modulo). Unlike Java, it also supports an exponentiation operator:

```
2 ** 8
```

This gives the result of two to the power of 8. If, like me, you've spent too much time in the past using Visual Basic, note that `^` does *not* perform an exponentiation. It's the XOR bitwise operator. You can see a complete list of operators in appendix A.

Let's now introduce some verification into the calculations. First, here is a simple equality comparison:

```
8 + 4 == 12
```

The result now is a word: `true`. This is a literal boolean value whose counterpart, as you'd expect, is `false`.

To verify that the expression does indeed result in `true`, you can prefix it with the keyword `assert`:

```
assert 8 + 4 == 12
```

When you run this, nothing different appears to happen. But what if the expression doesn't evaluate to `true`? Try

```
assert 8 + 4 == 11
```

The expression is obviously now false and this time you see an error message:

```
Assertion failed:
```

```
assert 8 + 4 == 11
         |   |
         12  false
```

```
at ConsoleScript6.run(ConsoleScript6:1)
```

What's interesting about this error is that it shows the values of all parts of the expression, including both the left hand and right hand sides. As you might imagine, this makes it relatively easy to diagnose problems in your code. Assertions are particularly useful in unit tests and other verification-based scenarios.

Let's now try out the Pythagorean Theorem. You do remember that from school, don't you? In case it's been too long, it states that the square of the hypotenuse of a right-angled triangle is equal to the sum of the squares of the other two sides. In code:

```
def a = 3
def b = 4
def result = Math.sqrt(a ** 2 + b ** 2)

assert result == 5
println result
```

This little example introduces some new concepts and syntax. First, it declares and uses a few local variables using the keyword `def`. Variables declared in this way can contain values of any type, not just numbers. In other words, the variables are untyped.

The second item of interest is the `sqrt()` method call. The syntax is absolutely identical to Java's and in this case you're seeing a *static* method. This is called on the class itself (`lava.lang.Math`) rather than on an instance of that class.

Last, but not least, the assertion fails to throw an error despite `result` being a floating point number. Groovy's `==` operator is largely value based so that `5` and `5.0` are considered the same despite being different types. This particular behavior is quite different to that of Java, which is very strict on equality.

Here's another example with some more method calls. It takes a hexidecimal string, converts it to an integer, divides that value by 2 (using integer division) and then prints out the hexidecimal representation of the result:

```
def n = Integer.parseInt("ED59", 16)
n = n.intdiv(2)
println Integer.toHexString(n)
```

The output is simply the string "76ac". Hexidecimal representations are fairly common in our line of work, so it's useful to know that you can easily transform them to and from integers.

The above example looks unassuming, but it highlights a common struggle with Groovy: where do

these methods come from? And where are they documented? To answer those questions, you first need to know what type you're dealing with. With static methods that's straightforward as you can see the type directly (`Integer` in the above example, which is short for `java.lang.Integer` - the fully qualified class name). It's not much harder for instance methods as you can always see the type of an object by calling the method `getClass()`:

```
println 10.getClass()
println 1.2.getClass()
println "Hello".getClass()
```

Executing this simple script results in the following output:

```
class java.lang.Integer
class java.math.BigDecimal
class java.lang.String
```

All three types are part of the core Java class library, which is fantastic if you already know Java. (And if you already know Java, you might be surprised at the floating point type - I'll talk about that soon TODO). What this means for you is that you'll need to bookmark the Java API documentation when coding in Groovy. You'll find the `parseInt()` and `toHexString()` methods described under `java.lang.Integer`. What you won't find is the `intdiv()` method.

What? Where does that come from then? One thing you have to remember is that one of Groovy's aims is to be as close to Java as possible while providing user-friendly features. What this means in practice is that Groovy extends the core Java class library with methods and properties that it thinks are particularly useful. `intdiv()` is one of those methods and you can find it nestled away in the Groovy JDK documentation.

The Groovy JDK is the name given to all the extra properties and methods that Groovy adds to the core Java classes. The layout of the documentation is similar to that of the Java API docs, and so you just need to select the Java class you're interested in to see what Groovy adds. Look under `java.lang.Number` (the super class of `Integer`) and you'll find `intdiv()`. Unfortunately it's not always obvious which type to look at for any given property or method, but it becomes easier as you get to know both Groovy and the Java class library.

There may be something nagging you by this point. All the method calls, such as `getClass()` and `sqrt()`, have parentheses. So what about `println()`? Isn't that a method call too? If so, where are the parentheses?

`println()` is indeed a method call, equivalent to `System.out.println()` in Java. Unlike any other method, though, it can be called as if it were a global function. It doesn't need to be called on an object or class like all other methods. As for the lack of parentheses, Groovy doesn't require them in certain circumstances. It's too early in your Groovy career to discuss exactly what those circumstances are, so for the moment, use parentheses for all method calls and only leave them out for `println()` calls.

One important point to be aware of is that the parentheses are required if you don't pass any arguments to `println()`:

```
println "Hello"
println()
println "Goodbye"
```

What happens if you don't have parentheses? Try it out:

```
println "Hello"
println
println "Goodbye"
```

You will now see an error message rather than the "Goodbye" text:

```
groovy.lang.MissingPropertyException: No such property: println for class: hello-goodbye
```

Note the class name of the exception: `MissingPropertyException`. Groovy is looking for a *property* called `println`, not a method. I'll come to properties soon, but for completeness, try this example where the parentheses are included but the method name is incorrect:

```
println "Hello"
print1n()
println "Goodbye"
```

This time, you will see a different error:

```
groovy.lang.MissingMethodException: No signature of method: hello-goodbye.print1n()
   is applicable for argument types: () values: []
Possible solutions: println(), println(), println(java.lang.Object),
   println(java.lang.Object), println(java.io.PrintWriter), print(java.lang.Object)
        at hello-goodbye.run(hello-goodbye.groovy:2)
```

Note how Groovy is stating that it can't find a method this time. It also rather handily displays a list of known methods that are a close match to the unknown method name just in case you have a simple typo or can't remember the correct name properly. You should always read the exception messages carefully as they are tremendously useful in determining why something isn't working.

Another interesting facet of the language is that you're seeing runtime errors for unknown methods and properties. This is because Groovy resolves properties and methods at runtime by default rather than at compile time. This can be changed, as you'll see in chapter TODO. You can also make use of this

runtime resolution to do interesting things such as creating builders (chapter TODO) or mock objects (chapter TODO).

Let's return to the world of numbers and look at some more examples. In the following script, I perform a floating-point multiplication repeatedly and time how long it takes:

```
def start = System.currentTimeMillis()  // <1>
def n = 100.0
for (i in 0..<100000) {             // <2>
    n = n * 0.95
}

println "n = ${n}"                       // <3>
println "Time taken: ${(System.currentTimeMillis() - start)}ms"
```

① Gets the current number of milliseconds since the Unix epoch (1st Jan 1970)

② Iterates over the integers 0 to 100,000

③ Displays the value of `n` within this string

Ignore the `for` loop and `println()` statements for the moment and just focus on the results of executing the script. First, the number it displays is ridiculously long. When you run it, you'll see what I mean. Second, it takes a while to finish - around 2.5 seconds on my laptop. Now add a `D` suffix to the literal `100.0` and run the script again:

```
...
def n = 100.0D
...
```

This time the number is significantly shorter and, more to the point, different. The script also finishes much faster - 33ms for me. What's going on? It all boils down to the number types being used. When you look at the classes behind the two number literals:

```
println 100.0.getClass()
println 100.0D.getClass()
```

you will see that the first value is of type `java.math.BigDecimal`, while the second one is a `java.lang.Double`. `Double` allows for fast calculations at the cost of accuracy, while `BigDecimal` is accurate at the cost of speed. In fact, `Double` cannot represent the final result because it's too small.

As you can guess from running these scripts, Groovy defaults to `BigDecimal` for floating point literals, which ensures accuracy. This behavior may bother some of you, but it's not a big deal. As you can see, it's easy to turn floating point literals into `Double`s through a simple suffix if you have the need for speed.

Integer numbers also show some interesting effects. Consider this example for calculating factorials:

```
def factorial(n) {                          // <1>
    return n == 1 ? 1 : n * factorial(n - 1)  // <2>
}

println factorial(10)
```

① Declares a new function called "factorial"

② Uses the ternary operator ('<condition> ? <value1> : <value2>') for a conditional result

A quick work on the implementation before we look at the behavior of this script. The easiest way to implement the factorial algorithm is via recursion, and for that you need functions. As you can see from the example, Groovy allows you to define functions in a script using the syntax

```
<return type> <method name>(<type> <arg>[, <type> <arg>,...]) {
    <method body>
}
```

This is also the basic syntax for defining methods within classes, as you'll see later. As with the local variables earlier, the `factorial()` function doesn't specify an explicit type for its return value, using `def` instead. Once defined, you can then call the function from anywhere inside the script.

You might notice that `factorial()` doesn't specify a type for its argument. Nor does it use `def`. That's because method arguments are a special case that don't require `def` if they are untyped. If you forget this, don't worry: Groovy happily accepts `def` in front of method arguments. It's just unusual amongst more experienced Groovy developers.

Executing the factorial script behaves as you'd expect, displaying the value 3628800 (everyone knows the factorial of 10, right?). But what if I pass a floating point number to the `factorial()` function instead?

```
...
println factorial(10.5)
```

Ouch! Now when you run this you'll see an ugly `StackOverflowError`. That's because the recursion only stops when `n` becomes 1, which never happens if the initial argument is not an integer. So if the function only works with integers, can we specify that as a constraint? Indeed we can. Change the signature of the method to

```
def factorial(int n) {
    ...
}
...
```

The argument is now *typed*. If you attempt to call the `factorial()` method with an argument of 10.5 now, you'll get a `MissingMethodException` because Groovy can't find a method whose signature accepts floating point numbers. You're still seeing a runtime error though, as Groovy doesn't perform a compile-time check even though we have specified the types.

Primitive vs object types

I guess now is as good a time as any to explain why `int` rather than `Integer`. Java has the concept of primitive types and object types. Primitive types, such as `int`, `double`, etc., are not objects and so you can't call methods on them. Nor can they have a value of `null`. On the plus side, they allow for faster calculations.

Groovy has inherited primitive types from Java, but it treats them differently. In fact, the only difference between the primitive and object types is that the former can't be `null`. You can access properties and methods on primitive types just as you can on object types. Because of that, I tend to use the primitive types. It's more habit than anything else. That said, it makes sense for `factorial()` to use the primitive type as the algorithm can't handle `null` values anyway.

Are you happy with the implementation of `factorial()` as it stands? I'm not. To understand why, try using a value of 17:

```
...
println factorial(17)
```

Were you expecting a negative number? Whether you were or not, that's what happens and it's certainly not correct. The reason for this is that `int` and `Integer` can only store integers up to a size of 2^{31}. If your value goes beyond that, then you end up with something called *integer overflow*.

You can force Groovy to use a wider integer type by changing the specified type. Long integers are 64 bit, so that will allow us to calculate larger factorials:

```
def factorial(long n) {
    ...
}
...
```

Unfortunately, it's not long before you hit integer overflows again. Just try using a value of 21! The

result is negative yet again. This is one occasion where the `java.math.BigInteger` type is particularly useful. It can handle integers of almost any size. As with `BigDecimal`, it's not as fast as using the core types, but it's speedy enough for many cases:

```
def factorial(BigInteger n) {
    return n == 1 ? 1 : n * factorial(n - 1)
}

println factorial(234)
```

You can declare types for local variables and method return values, not just method arguments. Groovy will automatically coerce values to the required type. If a particular coercion isn't supported, then you get a runtime error. Type coercion has a lot of subtleties to it, so there's a dedicated section for it in chapter TODO.

Chapter 3. Working with types

Numbers are an important part of programming. Ultimately, computers *only* deal with numbers. As humans, though, we prefer some higher level types such as strings. In this chapter, I'll show you how to use and interact with strings, collections, and maps. I'll also introduce some other concepts such as fields and properties.

3.1. The joy of strings

History has been recorded in the form of text for thousands of years - in stone, on paper, and now electronically. In that time, words have lost none of their power or usefulness. That's why a basic type representing text is such a fundamental part of any programming language. And of course that type should be able to handle any language.

Groovy uses Java's `java.lang.String` class which is able to represent text in any written language supported by Unicode, or more specifically the 16-bit Unicode Transformation Format (UTF-16). If you're interested, you can find a full list of supported scripts/alphabets on the Unicode website. I'll be using the latin alphabet almost exclusively in this book, but you can try this short script in the Groovy console to confirm support for other languages:

```
println "I live in London"
println "          "
println "                                          "
println "                        "
```

The above consists of English, Chinese (Simplified), Thai, and Greek (using Google Translate of course - I only wish I could write in so many diverse languages!). Just bear in mind that you need a text editor or IDE that supports UTF-8 if you want to use non-Latin characters in Groovy source code. This is because the Groovy compiler assumes UTF-8 by default.

It's quite possible to write a book about character encodings and internationalization, but it's out of scope for a Groovy language introduction. So from here on in, I'll be sticking to examples based on English. That will certainly make life easy for the next example which will involve some text analysis.

3.1.1. Strings are sequences

What is text? It's inherently a sequence of characters, which means you can evaluate its length, reorder it, extract substrings and count the number of vowels or consonants in it. It also means you can iterate over the characters.

Let's see how this works in Groovy. We start with a piece of text (a string), evaluate the properties we just described (length, etc.) and then print that information out:

```
def text = "Jack Rabbit Slims Twist Contest"
println "Text length: " + text.size()        // <1>
println "First character: " + text[0]        // <2>
println "Last character: " + text[-1]

final vowels = "aeiou"                       // <3>
def vowelCount = 0
for (ch in text) {                           // <4>
    if (vowels.contains(ch.toLowerCase())) {
        vowelCount++
    }
}

println "Number of vowels: " + vowelCount
```

① Concatenate strings with the `+` operator

② Access individual characters

③ Declare a constant using `final`

④ Iterate over the characters of the string

This example introduces several properties of strings:

- Concatenation with `+`

  You can build up strings this way with any values, including numbers, lists and so on. The only requirement is that the concatenation starts with a string.

- Text length with `size()`

  Java is annoying because you need to use different fields or methods to get the lengths of arrays, lists, and strings. In Groovy, you can just use the `size()` method for all types of sequence. You can find this method documented in the Groovy JDK entry for String.

- Array indexing to extract characters

  You can use the syntax of `[index]` on any type of sequence to pick out a particular element. The first element of a sequence is at index 0. If you want to index from the end of a sequence, use a negative number. So -1 represents the last element, -2 the second to last, and so on.

  One thing to note is that the array index operator returns a single character string in this case, rather than a `java.lang.Character`. You can verify this by adding a line `println text[0].getClass()` to the script. I'll discuss the `Character` type shortly as Groovy is noticeably different from Java on this score.

- Looping with `for`

The standard Groovy `for` loop works on any type of sequence, just like the array index operator. And similarly, each iteration gives you a single character string to work with instead of a `Character`. If you want to be more explicit you can include the type:

```
for (String ch in text) {
    ...
}
```

This does of course raise the question of when to explicitly specify the type of the iteration variable. It's a question that always crops up with Groovy because of its optional typing, hence why I discuss it more thoroughly in chapter [TODO] once you know most of the basics. In this particular case, it's easy to infer the type of `ch` and so the type isn't necessary. But some people may be more comfortable with it being explicit, and that's fine too.

I just want to finish off this section by identifying the source of the string methods used in the example. I've already mentioned that `size()` is provided by the Groovy JDK, but so is `toLowerCase()`, which returns a new string with all letters lower-cased. The `contains()` method is different because it is provided by the JDK itself.

This sort of thing can be frustrating: how are you supposed to know where to find these methods in the API docs? The best advice is to stay calm and just look at both the Java API and Groovy JDK docs when searching for a method. Alternatively, use an IDE that will give you auto completion: Eclipse, IntelliJ and NetBeans all have great Groovy integration. As you become more familiar with the APIs, you'll learn where the various methods come from.

Before moving on, there is one improvement I'd like to make to the script. I like things to be formatted so that they are more readable. The summary of the text analysis just doesn't cut it at the moment as the values are misaligned. We can easily remedy that in Groovy using one of its methods related to padding. In this case, we want to pad the labels so that the values all appear left-aligned:

```
def text = "Jack Rabbit Slims Twist Contest"
final padding = 20
println "Text length: ".padRight(padding) + text.size()
println "First character: ".padRight(padding) + text[0]
println "Last character: ".padRight(padding) + text[-1]
...
```

You could also try right-aligning the labels by using the `padLeft()` method instead.

3.1.2. String expansion

As you've just seen, string concatenation works just fine. That said, it does add noise to your code, particularly when there is more text than expressions. An alternative approach in Groovy is to use string expansion, by embedding expressions inside `${}` place holders:

```
...
println "I found ${vowelCount} vowels in the text"
```

I consider this to be more readable than

```
...
println "I found " + vowelCount + " vowels in the text"
```

It's also easier to keep track of the spaces you need. Did I remember to include the space after 'found' and the one before 'vowels'? It's not so clear with string concatenation.

You will often see an alternative form (without the curly braces) for embedding variables:

```
println "I found $vowelCount vowels in the text"
```

It's a little less noisy, but the expression doesn't stand out so well. I tend to recommend to newcomers that they only use the former syntax (with curly braces) for consistency and to avoid some confusing edge cases with the latter approach. Ultimately it's a style thing and up to you or your team.

So what happens when you want to include an actual dollar symbol in the string? Try this code snippet in the Groovy console:

```
println "I sent them a $1000 check by post"
```

I see an error message:

```
illegal string body character after dollar sign;
   solution: either escape a literal dollar sign "\$5" or
   bracket the value expression "${5}" at line: 1, column: 1
```

It's pretty clear that Groovy's not happy about that $ in the string. One solution is to escape it using a backslash:

```
println "I sent them a \$1000 check by post"
```

It works, but isn't pretty. Also imagine if you have a much larger string with lots of $ symbols. What are the chances that you're going to remember to escape all of them? In such cases, I prefer an alternative string literal using single quotes:

```
println 'I sent them a $1000 check by post'
```

The only difference between the single-quote and double-quote versions is that the latter supports embedded expressions. It's also important to understand that there is no memory or performance difference between the two. A double-quote string without any embedded expressions is still a plain `java.lang.String`. The decision to use one over the other then becomes one of personal preference. I prefer to use double quotes in most cases because I can easily insert an embedded expression at a later date without also changing the quotes. Others like the extra information tied to using single quotes, i.e. it is just a plain string.

3.1.3. Multi-line strings

This is the kind of thing that people have long discussions and arguments over, so let's move swiftly on before we end up in the same boat. Imagine you want to include a block of text in your code - perhaps it's an email template or long, formatted message you want to display to users. The easiest way to do this in Groovy is with a multi-line string literal:

Listing 2.1

```
def sayHello(name) {
    println """Dear ${name},

Thank you for signing up to our product. We hope you enjoy!

Kind regards,

Customer Services
"""
}

sayHello("Peter")
```

So simply by using three quotes at the beginning and end, you're allowed to put line breaks in the text. This is much nicer than using the `\n` new line character. Also notice how the above example includes an embedded expression: that's because it uses double quotes. You can also use three single quotes (`'''`) instead, in which case the expression doesn't get evaluated.

So far, so simple. But what if you don't like aligning the text in the first column of the source file? Perhaps you'd prefer to indent it like so:

```
def sayHello(name) {
    println """Dear ${name},

        Thank you for signing up to our product. We hope you enjoy!

        Kind regards,

        Customer Services
        """
}

sayHello("Peter")
```

Unfortunately, the spaces at the beginning of each line will end up in the output because they are part of the text. All is not lost, though! The Groovy JDK provides some extra methods that are ideal for such scenarios: `stripIndent()` and `stripMargin()`.

The first of these remove any leading whitespace that is common to all lines. The second, `stripMargin()`, removes all leading characters up to a specific character (`|` by default). We're going to use `stripIndent()` here, but there is one thing we need to be wary of: every line has to have the leading whitespace for it to be removed. And the code as it stands doesn't have leading whitespace for the first line!

To solve that, we're going to use a *line continuation*:

```
def sayHello(name) {
    println """\                    // <1>
        Dear ${name},               // <2>

        Thank you for signing up to our product. We hope you enjoy!

        Kind regards,

        Customer Services
        """.stripIndent()           // <3>
}

sayHello("Peter")
```

① `\` followed by a new line cancels the line break

② The first line of text now has leading whitespace

③ Removes the leading whitespace from each line

I almost always use a line continuation for the first line of multi-line strings, just to ensure nice

alignment. The main issue that typically arises from this technique is the presence of spaces or tabs after the backslash. For line continuations to work, the backslash *must* be the last character of that source code line.

3.1.4. Characters and type coercion

As you've seen, Groovy treats strings as if they are sequences of single-character strings. This seems a little strange as the Java class library does have a specific type representing a single character: `java.lang.Character`. Does this mean that you shouldn't use `Character` from Groovy?

The simple fact is that it would be hard not to use `Character` when so many Java APIs that you might want to use rely on them. For example, if you want to check whether a character is an alphabetic letter, you can use the static `Character.isLetter()` method. It takes a `Character` as its argument. To understand the problem better, consider this example:

```
def text = "Jack Rabbit Slims Twist Contest"
if (Character.isUpperCase(text[0])) {
    println "The text starts with a capital letter!"
}
```

Running this results in another one of those pesky missing-something exceptions:

```
groovy.lang.MissingMethodException: No signature of method: static
java.lang.Character.isUpperCase() is applicable for argument types: (java.lang.String)
values: [J]
Possible solutions: isUpperCase(), isUpperCase(char), isUpperCase(int), toUpperCase(),
toUpperCase(char), toUpperCase(int)
    at ConsoleScript27.run(ConsoleScript27:2)
```

As the exception says, there is no `isUpperCase()` method that takes a string. So how do you get a `Character` in Groovy? The solution lies in a multi-purpose conversion mechanism.

Groovy provides two conversion options, one implicit and the other explicit. The implicit version consists of assigning a value of one type to a local variable declared as another type. Applying this to the previous example, we get

```
def text = "Jack Rabbit Slims Twist Contest"
Character ch = text[0]                          // <1>
if (Character.isUpperCase(ch)) {
    println "The text starts with a capital letter!"
}
```

① Explicitly declare the local variable as the required type

This approach also works for variables declared in `for` loops:

```
def text = "Jack Rabbit Slims Twist Contest"

def upperCount = 0
for (Character ch in text) {                            // <1>
    if (Character.isUpperCase(ch)) {
        upperCount++
    }
}

println "There are ${upperCount} upper case letters"
```

① Explicitly declare the type of the loop variable

The second, explicit, approach involves a new operator: `as`. This is particularly useful when introducing a new variable just adds noise to the code. The previous example where we introduced a new local variable would be better coded as

```
def text = "Jack Rabbit Slims Twist Contest"
if (Character.isUpperCase(text[0] as Character)) {          // <1>
    println "The text starts with a capital letter!"
}
```

① Use `as` to coerce a value to a different type

Whichever approach you use, just be aware that it doesn't just magically convert all types to all other types. The built-in support has limited knowledge and is mostly used for:

- Converting a single-character string to a `Character`
- Upgrading numbers, for example from `Integer` to `Long`
- Converting a list to an array (I'll talk about this a little later)
- Creating a set from a list
- Converting pretty much anything to a `String`, since every type has a `toString()` method

You can't, for example, assign a string with more than one character in it to a `Character` variable. You'll get a `GroovyCastException`. That's good because automatic coercions are very useful but can cause productivity-sapping surprises when too prevalent. And remember that coercion does not happen automatically for method arguments, even if they are explicitly typed.

Object vs primitive types

Groovy inherits Java's type system, which means that you get both primitive and object types. For example, `char` and `Character`. These two types are interchangeable in Groovy as you can call methods on primitive types. The main difference is that object types can have a `null` value.

I tend to prefer the primitive types because they are shorter and I rarely want to allow `null` values for them. The available primitive types (and corresponding object types) are:

Primitive type	Object type
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`

From this point on, I'll use primitive types unless an object type is required for some reason.

Strings aren't the only form of sequences. What if you want a shopping list or a simple sequence of numbers? A list is a more generic form of sequence available in Groovy and we look at that next alongside other types of collection.

3.2. Collections, ranges and maps

When programming, we often need to deal with collections of objects, not just single instances. That's why most languages have built-in support for lists (an ordered collection or sequence) and other types of collection. As with strings, it's the Java class library that provides these types in Groovy. That doesn't mean that you get the same experience as if you're developing in Java. In fact, Groovy brings a greater level of consistency and expressiveness.

We'll start with ordered collections, which are represented by the `java.util.List` interface. In the following example, I create a list of words (strings) and then proceed to iterate over them to calculate the shortest, longest and average word lengths. Before you critize the implementation (there are better ways to do this), remember that I'm trying to introduce new concepts and APIs gradually!

```
final words = ["orange", "sit", "test", "flabbergasted", "honorific"]   // <1>

def minWordLength = Integer.MAX_VALUE                                   // <2>
def maxWordLength = 0
def totalLength = 0
for (w in words) {
    minWordLength = Math.min(minWordLength, w.size())
    maxWordLength = Math.max(maxWordLength, w.size())
    totalLength += w.size()
}

println "Min word length:  ${minWordLength}"
println "Max word length:  ${maxWordLength}"
println "Avg word length:  ${totalLength / words.size()}"               // <3>
```

① A list literal containing strings

② `MAX_VALUE` is a static field on `java.lang.Integer`

③ `size()` gives the lengths of strings *and* lists

The first thing that will strike you if you come from Java is how easy it is to create a literal list of values. It's simply a list of comma-separated values bounded by square brackets. We can put numbers in there, dates, or any other type. We're not even forced to use the same type for each element, so we can create a list of mixed strings and numbers for example. That said, collections commonly contain objects of the same type.

The `for` loop also works exactly the same as for strings. If you're trying the example out in an IDE, you might even notice that it's not worried about the `w.size()` method (IDEs with Groovy support tend to underline methods that they can't be sure exist on a particular object, typically because they can't be sure what the type of that object is). That's because Groovy infers that the list only contains strings, so therefore `w` must be a string too! This is just a taster of how Groovy straddles the usual line between dynamic and static (type-checked) languages. We'll be delving deeper into that topic later on in chapter TODO.

The previous example shows you how to iterate over a list, but what if you want to access an element at a specific index? As with strings, you can use the array index operator:

```
def words = ["orange", "sit", "test", "flabbergasted", "honorific"]

assert words[0] == "orange"
assert words[-1] == "honorific"
assert words[words.size() - 1] == "honorific"                           // <1>
```

① The array index can be an expression, as long as it evaluates to an integer

At this point, I can almost hear some Java developers drawing breath to point out a flaw in this

example. Surely I should be using the `equals()` method when comparing strings? Fortunately that's not the case with Groovy. The `==` operator performs a value comparison rather than an object identity one. In other words, it does what any non-developer would expect it to do.

Sometimes a list literal isn't sufficient and you need to build one dynamically. Consider a method that creates a list of the powers of two, i.e. $f(x) = 2^x$:

```
List powersOfTwo(int n) {
    def result = []
    for (i in 0..n) {                           // <1>
        result << 2 ** i                        // <2>
    }

    return result
}

assert powersOfTwo(2) == [1, 2, 4]
assert powersOfTwo(8) == [1, 2, 4, 8, 16, 32, 64, 128, 256]
println "Done!"
```

① Iterate over squential integers using a range

② Append a new value to the list

In this example, we start with an empty list and then add the relevant numbers to it via the `<<` (left shift) operator. Note that we're modifying the list directly rather than creating a new list with the extra element. If you'd prefer to use a non-mutating approach, you can use `+` instead:

```
...
    for (i in 0..n) {
        result += 2 ** i                        // <1>
    }
...
```

① Shorthand for `result = result + (2 ** i)`

Yes, this approach has a cost in terms of execution speed and memory usage (you're creating a new list and copying the existing elements on each iteration), but in many cases it's not significant within its context.

The `+` operator can also be used to merge lists, similar to the way it's used to concatenate strings:

```
def l1 = [1, 2, 3]
def l2 = [4, 5, 6]

assert l1 + l2 == [1, 2, 3, 4, 5, 6]
assert l1  == [1, 2, 3]
println "Done!"
```

The more memory-efficient, mutating approach requires an explicit method call rather than an operator:

```
def l1 = [1, 2, 3]
l1.addAll([4, 5, 6])

assert l1 == [1, 2, 3, 4, 5, 6]
println "Done!"
```

I have noticed that some people try to use the `<<` operator for this, but it doesn't work. What you end up with is a 4 element list whose last element is also a list: `[1, 2, 3, [4, 5, 6]]`. Try it!

You can find the `addAll()` method and others defined on `java.util.List`. It's also worth checking the `java.util.Collection` and `java.util.Iterable` interfaces too, since `List` inherits from them.

When it comes to sequences, you should always try to use lists where possible as they have the richest API and widest support. They also enable you to take advantage of the Java 8 stream support. Unfortunately, you may still come across APIs that use another type of sequence: arrays.

3.2.1. Coping with arrays

Many years ago, before Java reached version 1.2, the only built-in support for sequences (other than strings) was the array. Some people still use them for reasons of efficiency, although thats not a common case. The main reason you'll encounter them is through using some API, often in the Java class library.

Let's start with the `String.split()` method. I frequently use this to convert a string into a sequence of words, but as you can see from the documentation, it returns an array rather than a list. So what magic do you need in order to process the array? As you can see from the following example, in some cases you don't need to do anything:

```
def words = "Jack Rabbit Slims Twist Contest".split(" ")        // <1>

def minWordLength = Integer.MAX_VALUE
def maxWordLength = 0
def totalLength = 0
for (w in words) {
    minWordLength = Math.min(minWordLength, w.size())
    maxWordLength = Math.max(maxWordLength, w.size())
    totalLength += w.size()
}

println "Min word length:  ${minWordLength}"
println "Max word length:  ${maxWordLength}"
println "Avg word length:  ${totalLength / words.size()}        // <2>
```

① Returns a String array

② `size()` method works on arrays

This code looks no different to the example earlier where we started with a literal list of words. Groovy allows you to treat an array as if it were a list. So you can iterate over it and also use the `size()` method. Many other Groovy JDK methods work on both lists and arrays too. Yet there are limits to this approach that you will become familiar with.

For example, you cannot append to an array. The following will result in a `MissingMethodException` for `leftShift()`, a clear sign that the method only applies to lists, not arrays:

```
def words = "Jack Rabbit Slims Twist Contest".split(" ")
words << "Test"
```

That exception is particularly interesting because it hints at how operators such as `<<` are handled in Groovy. Don't worry, I'll be coming back to that a little later.

In fact, the uniformity that Groovy provides across arrays and lists all happens within the Groovy JDK. Arrays in Java have no methods for you to invoke and they only have one significant field: `length`. So the best place to look to find out whether you can use a particular method on an array is the Groovy JDK doc for http://docs.groovy-lang.org/docs/next/html/groovy-jdk/?java/lang/Object.html[`Object[]`].

Alternatively, you can banish any doubt or confusion by converting the array to a list as soon as possible:

```
def words = "Jack Rabbit Slims Twist Contest".split(" ") as List    // <1>
words << "Test"
```

① Use `as` to convert an array to a list

That `as` operator is already proving remarkably useful and will continue to do so. And I think this is the safest approach when dealing with arrays that are returned to you. Even if the API suddenly changes and returns a list, your code will continue to work (the `as` operator will just return the value as is).

I've covered the case of a method returning an array, but there is still the question of how to handle methods that require an array as an argument. The solution is simple and relies on a technique I've alread shown you. Do you remember how we got a `Character` from a `String`? You can either declare a local variable with the required type or use the `as` operator. Here's an example using `as`:

```
def proc = Runtime.getRuntime().exec(["java", "-version"] as String[])
def out = new StringWriter()
def err = new StringWriter()
proc.waitForProcessOutput(out, err)

println err
```

As you've probably worked out, the array type is the one with the `[]` suffix. So you can have `int[]`, `Object[]`, and so on. One thing you can't do is initialize an array using the Java syntax:

```
int[] words = new int[] { "orange", "sit", "test", "flabbergasted", "honorific" }
println words
```

You'll get a compilation error if you try. In fact, this is one of the rare cases where standard Java syntax is not valid Groovy.

That's pretty much all you really need to know about arrays. I'd like to finish here with just a brief look at the `MissingMethodException` I mentioned earlier. The full message looks like this:

```
groovy.lang.MissingMethodException: No signature of method:
[Ljava.lang.String;.leftShift()
is applicable for argument types: (java.lang.String) values: [Test]
        at ConsoleScript9.run(ConsoleScript9:2)
```

Look carefully at the method signature. What type does it think `leftShift()` is being invoked on? Yes, it's `[Ljava.lang.String;`. That looks a bit weird, right? This is the stringified form of the associated array class. You can easily identify arrays by that leading `[`. The `L` and trailing `;` represent an object type. Here are some more examples:

```
int[]      =>  [I
Integer[]  =>  [Ljava.lang.Integer;
```

I bring this up because it's important to recognise what types are involved when you're trying to

diagnose problems in your code, and Java doesn't make it too easy when it comes to arrays!

Although sequences are common, they aren't the only form of data structure. Sometimes you don't care about order but do want to ensure that you don't have duplicate elements. That's when a set comes in handy. Or you may want a list that is cheap to add elements to. It's useful to know how to work with other collection types.

3.2.2. Collection types

I've already mentioned that lists in Groovy implement `java.util.List`. That's an interface, so what type of list are you actually getting when you use the list literal? It's easy to find out in the Groovy console using the following:

```
println([].getClass())
```

Hopefully you'll see `class java.util.ArrayList` in the output pane. This is possibly the most commonly used list implementation in Java as it's cheap to access elements at an arbitrary index (otherwise known as random access). You can find out more about its characteristics in the Java API docs.

Before I talk about other list implementations, I'd like to point out a common gotcha with the previous example. In its case, the parentheses for `println()` are required. Otherwise you get a `MissingPropertyException` for `println`. Why? Because Groovy interprets the square brackets in this:

```
println [].getClass()
```

as the array index operator, not a list literal. As in `people[0].getClass()`. If you get into this kind of situation, make sure you fully digest the exception message. It often reveals the source of the problem right away. In this case, the exception makes it clear that Groovy thinks that `println` is a property, not a method. The quick insertion of parentheses will quickly resolve such cases.

Back to list types. Given that there are different implementations, the key question is this: how do you use an alternative implementation to `java.util.ArrayList`? The answer depends on whether you're starting with an existing list or not. Let's go back to our list of words, but this time make it a *linked list*:

```
def words = ["orange", "sit", "test", "flabbergasted", "honorific"] as LinkedList

println words.getClass()
```

Once again, it's the `as` operator that comes to our rescue. You can of course explicitly declare a local variable as type `LinkedList` for the same effect.

Array lists and linked lists are the two main types of list provided by the core Java class library. You might notice a couple of other implementations lurking in the `java.util` package too: `Vector` and `Stack`.

They're pretty much legacy, superseded by `ArrayList` and queues, and you can safely ignore them. If you have specialist requirements, consider third-party libraries, such as Google's Guava, that provide alternative list implementations.

Java does provided greater variety when it comes to `java.util.Queue` and `java.util.Deque` (double-ended queues) implementations, although Groovy defaults to `LinkedList` if you coerce a list to `Queue`. A more distinct collection type is the set:

```
def uniqueNumbers = [3, 3, 1, 1, 2, 2] as Set

assert uniqueNumbers.size() == 3
println uniqueNumbers
println uniqueNumbers.getClass()
```

As you can see, a java.util.Set doesn't allow duplicate elements. They typically don't care about order either. However, Groovy's default implementation (`java.util.LinkedHashSet`) will reliably print the numbers in the same order as they are declared in the list literal. That's because it retains *insertion order*, so when you iterate over the set, it does it in the order that elements were added to the collection.

A more common requirement, especially for numbers and strings, is to use the elements' *natural sort order*. You can easily create such a set by coercing to the `java.util.SortedSet` interface:

```
def uniqueNumbers = [3, 3, 1, 1, 2, 2] as SortedSet

println uniqueNumbers
println uniqueNumbers.getClass()
```

You will now see the set printed out as `[1, 2, 3]` with a class of `java.util.TreeSet`.

This isn't the only way to get the collection type you want, particularly if you want to start with an empty one. Since the collections are normal Java classes, you can instantiate them directly using the `new` keyword:

```
def uniqueNumbers = new HashSet()
uniqueNumbers.addAll([1, 1, 2, -1, 1])

println uniqueNumbers
```

Interestingly, when I run this script I see the numbers printed in numeric order: -1, 1, 2. Trust me, that's a coincidence. The `HashSet` implementation gives no guarantees as to its iteration order.

There's just one more type connected with collections that I want to cover: the map, also known as an associative array.

3.2.3. Maps

I remember my early C++ days when it was hard to find an existing hash map implementation. I even remember trying to write my own. So it was with some relief that I saw Java came with its own map classes, which I could use with abandon. I haven't looked back.

Perhaps surprisingly, the `java.util.Map` interface doesn't extend `java.util.Collection`, even though you can think of it as a collection of key-value pairs. For example, consider this script which prints the keys and value of all the entries in a map:

```
def epics = [VMW: "VMware", MSFT: "Microsoft", AAPL: "Apple"]

for (e in epics) {
    println "${e.key}: ${e.value}"
}
```

A map literal looks a lot like a list literal, with the square brackets and comma-separated elements. The key difference is that you have a colon between each key and its value. When you then iterate over a map, each element is an instance of `java.util.Map.Entry`. Note that `Entry` is defined within the `Map` interface, making it a nested interface.

A closer look at the `Entry` class shows that there are no `key` or `value` fields there, so why does `e.key` work? The answer lies in Groovy's concept of properties. Don't worry, I'll explain these as soon as I've finished with maps.

Iterating over a map is all very well, but one of the main reasons to use such a data structure is so that you can quickly look up the value for a given key. Groovy allows you to do this through the array index operator, using a key instead of an integer. You use the same operator to add or overwrite individual entries:

```
def epics = [VMW: "VMware", MSFT: "Microsoft", AAPL: "Apple"]
def searchKey = "MSFT"

println epics[searchKey]                                    // <1>

epics[searchKey] = "Micro$oft"
epics["GOOG"] = "Google"                                    // <2>

println epics
```

① Fetches the value for a given value (returns `null` if the key doesn't exist)

② Adds another map entry

Note how I need to use string literals with the array index operator but not in the map literal. That's because the literal is a special case in which the key is assumed to be a string literal by default. The

example will work just the same if you put quotes around the keys like so:

```
def epics = ["VMW": "VMware", "MSFT": "Microsoft", "AAPL": "Apple"]
println epics
```

It's really just a convenience because string keys are so common in map literals. Of course, not all maps use strings for their keys. What happens if you want your keys to be numbers? That's easy because number literals (integers and floating point) work as is:

```
def numberWords = [1: "One", 2: "Two", 3: "Three"]
println numberWords[2]
```

The downside comes when you want to use a variable (or more specifically its value) as a key. This is a surprisingly rare scenario, but as an example, imagine we have a method that takes a string as an argument and returns a map with that string as a key and its length as the value:

```
def stringAttributes(String str) {
    return [(str): str.size()]                              // <1>
}

println stringAttributes("Hello world!")
```

① Use the string value as the key, not the literal `"str"`

When you run the script you should see the output

```
[Hello world!:12]
```

These basics will get you through the examples in rest of the book, so I want to go back to the question of why `e.key` works when there is no field on the `Map.Entry` class with that name. It's time to talk about one of the most important concepts in Groovy: properties.

3.2.4. Properties

In object-oriented languages, objects can have both state and behavior. That means fields and methods if you're writing Java. The trouble with fields is that you can't override them, for example to evaluate their values on demand. So Groovy introduces an additional form of state: the property. The basic idea is that properties are accessed through two methods whose signatures follow a specific convention:

```
<property type> get<PropertyName>()
void set<PropertyName>(<property type> val)
```

These are known as the *getter* and *setter* methods, or simply getters and setters. So in the case of the `Map.Entry` class, you have a `getKey()` method that corresponds to a property named `key`, and `getValue()` and `setValue()` methods that represent a `value` property.

So how do you map the method names to the equivalent property names? The rules are based on the JavaBeans specification, but it's probably easiest to understand through some examples. Here's a table mapping method names to property names:

Method	Property name
`getKey()`	`key`
`getFullName()`	`fullName`
`getLine1Address()`	`line1Address`
`isHidden()`	`hidden` (only valid for boolean properties)
`getURLScheme()`	`URLScheme`

As you can see, the basic rule is to remove the leading "get" and then lower case the first letter of what's left. The only case in which this rule doesn't hold is when the first two letters of what's left are both upper case. The property name is the samme as the method name in this situation (minus the "get"). It's also worth pointing out that boolean properties can use a getter or an `is<PropertyName>()` method.

The `Map.Entry` example raises another question. What happens when there is a getter but no setter, as is the case with `Entry.getKey()`? To demonstrate, I'm going to bring in the `File` class and try to change the name of the file:

```
def f = new File("test.txt")
println f.name

f.name = "new.txt"
println f.name
```

Running this will print out "test.txt" as expected, but you'll see an exception following on swiftly:

```
groovy.lang.ReadOnlyPropertyException: Cannot set readonly property: name for class:
java.io.File
        at ConsoleScript9.run(ConsoleScript9:4)
```

So if you have a getter without a corresponding setter, you end up with a read-only property. This can be particularly useful for properties that are calculated on demand and have no corresponding field to store the state.

You can of course still access the methods as methods. You don't have to use property syntax. The

above example then becomes

```
def f = new File("test.txt")
println f.getName()

f.setName("new.txt")
println f.getName()
```

This time, though, you'll get a `MissingMethodException`. Otherwise the script behaves as before.

I generally prefer property syntax over using the methods directly because the code looks cleaner and more readable, which I find very valuable. One of the few time I prefer using a getter method explicitly is with the ubiquitous `getClass()`, which tells you the type of an object. The reason for that is because of what happens when you use property syntax on a map:

```
def map = [:]                                                    <1>
println map.class
println map.getClass()
```

① This is an empty map. The colon distinguishes it from an empty list.

You'll see that the script first prints `null` before then displaying the full type of the map on the next line. That's because Groovy adds special behavior to `Map` that overrides the default property access. So the second line of the script looks for a key named "class", doesn't find it, and returns `null`. I've been caught out by this more times than I can recount, so I just use `getClass()` explicitly.

Speaking of property syntax and maps, let's take a look at a previous map example, replacing the array index operator with property access:

```
def epics = [VMW: "VMware", MSFT: "Microsoft", AAPL: "Apple"]
println epics.MSFT

epics.GOOG = "Google"
println epics
```

Whether you use the array index operator or property syntax depends on your own preferences. I typically use the former if I know that I'm dealing with a map. However, property syntax can be useful if the target object could be a map or another type entirely. I'll explain how this works when I dive into the dynamic nature of standard Groovy. Other developers prefer the property syntax because it's less noisy and easier to type (my little finger certainly struggles with the square brackets on a US keyboard).

You've now been introduced to the core types in Groovy, which will set you in good stead for future examples. It's hard to code without strings, collections, or maps! I just want to finish off this chapter

with a more complete example that you can play with, based on Pig Latin.

3.3. Chapter example

If you're not familiar with Pig Latin, it's a technique for obfuscating English words so that they aren't instantly recognisable. The rules I'll follow are fairly straightforward:

- If a word begins with a consonant, I move the starting consonants to the end of the word and then add an "ay" suffix.

  ```
  For example, "happy" becomes _appyhay_ and "glove" becomes _oveglay_.
  ```

- If a word begins with a vowel or a silent letter, then I simply add "way" as a suffix.

  ```
  For example, "egg" becomes _eggway_ and "eight" becomes _eightway_.
  ```

You can find out more information about Pig Latin on Wikipedia. It's mostly just a game that people play. So how do we go about implementing such a thing?

Let's start with a script that has a method for doing the conversion plus a bunch of assertions to verify that it works correctly:

```
String pigLatinize(String text) {
    return ""
}

assert pigLatinize("happy") == "appyhay"
assert pigLatinize("egg") == "eggway"
assert pigLatinize("hello earth") == "ellohay earthway"
```

As it stands, running this script will result in an assertion error. It's kind of obvious because the method always returns an empty string. I'm just trying to take the Test Driven Development (TDD) approach!

The logic for the conversion is deceptively simple (if we ignore the hard cases):

- Split the text into individual words
- Convert each word
  - If the word starts with a series of consonants, place them at the end
  - Add an 'ay' suffix

- Recombine the words into a single string

Before we start, though, consider how we are going to do the split and merge. We ideally want to retain all the characters between words and make sure that they get merged back in at the right places. Another thing to consider is whether we want to handle hyphens or other characters that can occur in words. Dealing with strings is rarely easy.

Let's take a quick and dirty approach: split the text on word boundaries using a regular expression. That will at least retain the characters between words, ensuring that they appear in the result. We may still have trouble with some bits of text, but t will work as a simple demonstration. You can experiment with other approaches as you see fit! Here's my suggested implementation:

```
String pigLatinize(String text) {
    def wordsAndSeparators = text.split(/\b/)                // <1>
    def pigLatinWords = []
    for (str in wordsAndSeparators) {
        if (Character.isLetter(str[0] as char)) {            // <2>
            pigLatinWords << pigLatinizeWord(str)
        }
        else {
            pigLatinWords << str
        }
    }
    return pigLatinWords.join("")                            // <3>
}

String pigLatinizeWord(String word) {
    def leadingConsonants = leadingConsonants(word)
    return (word - leadingConsonants) + (leadingConsonants ?: 'w') + "ay"
}

String leadingConsonants(String word) {
    def vowels = ["a", "e", "i", "o", "u"] as Set
    def firstVowelIndex = 0
    for (ch in word) {
        if (ch in vowels) {
            break                                            // <4>
        }
        firstVowelIndex++
    }

    return word.substring(0, firstVowelIndex)
}

assert pigLatinize("happy") == "appyhay"
assert pigLatinize("egg") == "eggway"
assert pigLatinize("hello earth") == "ellohay earthway"
println "Done!"
```

① Breaks text into a list of words using the JDK method `String.split()`

② Only pig latinize words that start with a letter

③ Merge the list of words into a single string

④ Break out of the loop once we hit a vowel

Have fun with the example! The only features that I haven't covered so far are:

- `break` can be used to prematurely stop a `for` loop.

- `?:` is a special operator which uses the value on the left hand side if it's not `null` or an empty string. Otherwise you get the right hand side instead. It's useful for default values: if this expression is already a valid value, use that, otherwise use this default value. It's often referred to as the Elvis operator.

- `/ /` is a special string delimiter like `' '` and `" "`. It's behavior is identical to double-quoted strings except you don't need to escape the "\" character. It's particularly useful with regular expressions.

In the next chapter, I want to take the foundations set out in the first two chapters and use them to start doing some practical work with scripts, making use of other Java types and libraries.

Chapter 4. Doing odd jobs (scripting)

As software developers, we typically build applications and systems. And yet we often find ourselves doing miscellaneous other tasks that involve processing text files, fixing databases, generating reports, etc. Those tasks may even need to be repeated in the future. Such tasks can be laborious and error prone when performed manually, so I like to write scripts to handle them.

In the past, I'd break out Sed and Awk for text processing and Ruby/Python for anything more involved. But these are not tools that I use day to day and I'd end up spending a lot of time trying to remember their syntaxes and (in the case of Ruby & Python) their class libraries. That's not helpful when you're trying to *save* yourself time.

Groovy's an ideal scripting language for Java developers because it makes many of the things you need to do in scripts easy compared to Java, while using the same class library and a very similar syntax. You don't have a big context switch going between Groovy and Java. Even if you're an avid fan of a different scripting language, Groovy's worth considering simply due to the JVM's cross-platform support, which is handy in multi-platform environments.

In this chapter, I'll show some common use cases for Groovy scripting that will also help you to become familiar with more of the standard API. I'll also stop along the way to discuss any language structures that appear, with a particular focus on an important feature called closures.

4.1. Working with the filesystem

A lot of scripting needs involve files: finding them, parsing them, creating them, moving them around, and so on. Java's API has been traditionally weak in this area, which is why you'll find libraries like Commons IO that provide a lot of extra utility and convenience. That changed with Java 8, which introduced some significant enhancements to the file APIs. Not everyone will be able to use Java 8 though, so I will give examples for both Java 6 and Java 8. Even if you're able to use the latter, you will certainly find it useful to learn the old API because so many Groovy projects use them and will probably continue to use them.

Let's start with the basic mechanism of reading and writing files. Imagine you have a comma-separated values (CSV) file containing the first and last names of authors along with the country and date of birth. Your first job is to print out all the names of the authors sorted alphabetically by their surnames (or family names). Here's a quick script to do that using Java 8, which you can run in the Groovy console:

```
import java.nio.charset.Charset                                  <1>
import java.nio.file.Files
import java.nio.file.FileSystems

def lines = Files.readAllLines(
        FileSystems.default.getPath("authors.csv"),              <2>
        Charset.forName("UTF-8"))

def authorNames = []
for (l in lines) {
    def authorDetails = l.split(/\s*,\s*/)
    authorNames << "${authorDetails[1]}, ${authorDetails[0]}"
}

authorNames = authorNames.sort()
for (name in authorNames) {
    println name
}
```

① Most classes need importing before you can reference them

② Create a file path representing the location of the CSV file

There's much to talk about here, starting with the core classes we're using to work with the filesystem. The `FileSystems` class provides access to the default `FileSystem` of the current platform (Windows, Mac OS X, Linux, etc.), which in turn allows you to construct file paths represented by the `java.nio.file.Path` interface - a contract of behavior without an implementation. Those file paths are then used by the methods of the `Files` class to perform file operations, such as copying, reading, and listing directories. `Charset` is simply a representation of a character encoding such as ISO-8859-1 and UTF-8.

Why do we import these classes? It's to help avoid name clashes. Classes have a name and a package, also known as a namespace in other languages. This means you can have classes with the same name as long as they are in different packages. The result is that you have to tell Groovy (and Java) which class you are referring to in your code, and that's what the `import` statement does.

The `readAllLines()` method assumes that the target file is text with line breaks in it and produces a list of the lines of text. One interesting aspect of this particular method call is that I explicitly specify a character encoding. If you don't provide one, the method assumes the text is encoded using the default platform encoding, which varies between Windows, Mac OS X and Linux. I like to use UTF-8 everywhere, but that may not be appropriate for you. It largely depends on who creates the files in the first place. Yes it's a headache if you're dealing with any non-ASCII characters, but you need to factor it in to your coding right from the start.

The rest of the script uses features I introduced in the previous chapter. It simply breaks each line of text into its component parts, removing the commas in the process. The first name and last name for

each entry are then combined into a single author name string, with the last name first. That means we can perform a natural order sort on the resulting list to get the names in last name order before then printing them all.

Hopefully that's all straightforward for you, even though the code for reading the file seems more verbose than is perhaps necessary. So what happens if you don't have access to a Java 8 or newer JVM? In that case, you can use the old `java.io.File` class instead:

```
def lines = new File("authors.csv").readLines("UTF-8")

def authorNames = []
...
```

I've only shown the file reading code because that's the only difference from the previous example. What's striking is that I've somehow managed to replace 4 lines of code with just one. A big reason for this is the lack of imports, since I'm only using a single class this time, `File`, and Groovy doesn't require you to import it. In fact, there are several core packages that don't need to be imported because they're so commonly used. Here's a short list of those exceptions:

- `java.lang.*`
- `java.io.*`
- `java.net.*`
- `java.util.*`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `groovy.lang.*`
- `groovy.util.*`

These packages cover such classes as `File`, `String`, `List`, and `URL`, which means that your scripts become a lot cleaner without those `import` statements. One thing that newcomers may find confusing is how sub-packages are treated. Consider the classes associated with regular expressions, which I'll talk more about later in this chapter. They reside in the package `java.util.regex`, which looks like a sub-package of `java.util`. In fact, packages aren't hierarchical except in the way we developers interpret them. The concrete result is that you have to explicitly import a class like `java.util.regex.Pattern` because its not covered by the `java.util.*` auto import.

Another factor in the reduced amount of code is that `File` is a simpler API. And that's one reason why you will continue to see it, even with code that exclusively runs on Java 8 or above. One thing to bear in mind is that in this case, the `readLines()` method is provided by the Groovy JDK extensions rather than Java. I recommend that you peruse the Groovy JDK documentation for `File` because you'll find plenty of useful methods to try out.

This all raises an obvious yet important question: Which API should you use? I would argue that you should move over to the new NIO classes as soon as you can. Although `File` is generally simpler and less verbose, which is fine for scripts with simple needs, the NIO API has several advantages. First, it's easier to unit test as `Path` is an interface and you can implement your own mock `FileSystem` (or you can use an existing one such as JIMFS). Second, it integrates with Java 8 streams, allowing you to do things like lazily walk a directory tree.

One way to reduce the clutter in the Java 8 version of the CSV script is to make use of some special `import` features. You can both alias a class, giving it a shorter name, and explicitly import static methods, meaning you don't have to qualify them with the class name. Here's a modified version of the script using these techniques:

```
import static java.nio.file.Files.readAllLines              <1>

import java.nio.charset.Charset as Ch                       <2>
import java.nio.file.FileSystems as FS

def lines = readAllLines(FS.default.getPath("authors.csv"), Ch.forName("UTF-8"))

def authorNames = []
...
```

① Allows us to call `readAllLines()` without qualifying with the class name

② Aliases the class name to something shorter

As you can see, the code for reading the lines of text from the file is now more readable, which is always good for aiding comprehension of your code. You'll find static imports in Java as well, but the aliasing is specific to Groovy.

I now want to move away from the kinds of scripts that you can run in the Groovy console to something a bit more useful: command line scripts. You can still write them in the Groovy console, but you must first save them to a file before you can run them from the command line.

The first example harks back to the days of Apache Ant, a venerable build tool from the Java world. I often used its Filter task as a simple template engine, replacing occurrences of `@TOKEN@` with whatever parameter values I needed. It's not hard to reproduce this behavior in a Groovy script, as you'll see from the following example:

```
import java.nio.file.FileSystems as FS

if (args.size() != 1) {
    println "Incorrect number of arguments."
    println ""
    println "USAGE: processTemplates TMPL_PATH"
    System.exit(1)
}

final tokens = [                                          // <1>
    NAME: "Whizzbang",
    CURRENT_DATE: new Date()]
final templatePath = FS.default.getPath(args[0])
final sourceText = templatePath.getText("UTF-8")

def processedText = sourceText
for (entry in tokens) {
    processedText = processedText.replace(                // <2>
            "@${entry.key}@",
            entry.value.toString())
}

final outFile = FS.default.getPath("processed.txt")
outFile.setText(processedText, "UTF-8")                   // <3>
```

① Hard code the replacement tokens

② Perform the token substitution using a standard JDK method

③ Write the resulting text to a new (hard-coded) file

You should by now be familiar with most of the code in the script. The key new feature is the `setText()` method, which allows you to easily write text to a file. If the file doesn't exist, it gets created. Otherwise, its content is overwritten by the string you provide. The method also closes the file automatically, so you don't have to worry about leaving file handles open.

This works well for simple cases, but you often want to write to a file in chunks. Imagine you want to read records from a database table, perhaps process them, and then generate a CSV file containing one line per record. You could create the CSV in memory as a string and then write it out using `setText()`, but this is an inefficient use of memory and could take noticeably longer than writing each line to the file as you go. It depends on how many records you have.

Fortunately, Groovy provides several convenience methods for writing to a file on an as-you-go basis. I'm using `withPrintWriter()` in the next example to do exactly that, while also ensuring that the underlying file is closed regardless of whether my code throws an exception or not. The script takes a path to a directory and then generates a CSV file containing the filename, file size, and creation date for each file in that directory.

```
import static java.time.format.DateTimeFormatter.ISO_OFFSET_DATE_TIME

import java.nio.file.Files
import java.nio.file.FileSystems as FS
import java.nio.file.Path
import java.time.ZoneId                                          // <1>

if (args.size() != 1) {
    println "Incorrect number of arguments."
    println ""
    println "USAGE: filesReport DIR_PATH"
    System.exit(1)
}

final dirPath = FS.default.getPath(args[0])
final files = Files.list(dirPath)

final outFile = FS.default.getPath("files.csv")
outFile.withPrintWriter("UTF-8") { writer ->                     // <2>
    for (path in files) {
        writer.println("${path.fileName}, ${Files.size(path)}, ${getCreationDate(path)}")
    }
}

println "Written report to ${outFile}"

String getCreationDate(Path path) {
    final creationInstant = Files.getAttribute(path, "creationTime").toInstant()
    return ISO_OFFSET_DATE_TIME.format(creationInstant.atZone(ZoneId.systemDefault()))
}
```

① Use the Java 8+ Date & Time API

② Generate the CSV file, with one line per source file

This example doesn't have many lines, but it does introduce two important features:

1. The Java 8+ Date and Time API

2. Closures

Let's start with the new API.

4.2. Dates and time

Before Java 8, developers either got by with the `java.util.Date` and `java.util.Calendar` classes, or they used a third-party library called Joda Time. The latter was often the only choice for even moderately

complex date and time work because the core JDK classes were awkward to use and underpowered.

The Date and Time API brings coherence to working with dates in Java. Simple things are often easier to accomplish than with the old `Date` class, while you also gain greater control with more complex requirements. Still, the `Date` class should work just fine for the file creation dates in our example. So why use the `java.time.*` classes? For two reasons. First, the file NIO API gives us a `java.time.Instant` value rather than a `java.util.Date`. Second, I think you should prioritize the Date and Time API over `java.util.Date` for your own code because it's a significant improvement over the older API. For example, the core classes are immutable and thread safe, unlike `Date` and `Calendar`.

Figure xxx shows you the most relevant classes of the Date and Time API for right now. The `Instant` class represents a fixed point in time, independent of calendars and dates. If people around the globe retrieved the current time as an `Instant` at the same time, they should all get the same value, Of course, this is unlikely as there will almost certainly be variations between the system clocks on their computers. But that's the ideal.

The `java.time.LocalDateTime` class is different in that it represents a (Gregorian) calendar date and time, such as New Year's Day (1st January for any given year). This is not a fixed point in time because it depends on the time zone you're interested in. New Year's Day hits Australia well before the west coast of America. To correlate a date and time to an `Instant`, you need to use either `java.time.ZonedDateTime` or `java.time.OffsetTimeZone`, both of which incorporate the time zone information.

It's often hard work dealing with calendars and keeping track of time zones, but we're fortunate that the API makes life easy for our simple use case. The idea is to store the file creation date in the CSV file as an ISO 8601 date rather than an instant. To do that, we want to format the `Instant` we get from the `Files` class. The date formatters require dates rather than instants, though, so we use the `Instant.atZone()` method to perform the conversion using the system's current time zone. We could also perform the conversion using the UTC time zone if we wanted. It all depends on whether we're interested in which time zone the file was created in. For server-side work, I'd recommend almost always using UTC.

The Date and Time API brings more to the table than just these classes. Durations and periods allow you to readily calculate deadlines and reminders among other things. Overall, it's a richer API than we had previously in the JDK without it being laboriously hard to work with.

I now want to go onto the second feature the script introduces: closures.

4.3. Closures

Developers start using Groovy for a variety of reasons. I started because it was the language of the Grails web framework. But if we could narrow down why people then stayed with Groovy to a single language feature, I'm convinced it would be closures. It's fairly easy these days to explain the basics of them because most of the popular languages have comparable features:

- Ruby code blocks

- Javascript functions
- Java 8 lambdas
- Python functions and lambdas
- C# lambda expressions and anonymous functions
- Scala functions

These aren't all exactly the same feature, but they do allow for higher-order functions. If you're not familiar with this term, it means that functions can themselves be arguments to other functions. Consider the example of filtering a list of words. You might want all the words that start with the letter 'p'. You might instead be interested words longer than 5 letters. Or perhaps you just want to remove all swear words using a dictionary. The only difference between each of these requirements is the filter constraint. Does the word start with a particular letter? Is it longer than a particular value? The algorithm for filtering the list is always the same, it's only the constraint that changes.

That constraint can be implemented as a standalone predicate function, i.e. a function that returns `true` or `false`. Let's see what this looks like in Groovy code:

```
final swearWords = ["flabbergasted"] as Set
final words = ["orange", "sit", "profit", "test", "flabbergasted",
        "honorific", "proposal"]

println words.findAll({ word -> word.startsWith("p") })
println words.findAll({ word -> word.size() > 5 })
println words.findAll({ word -> !swearWords.contains(word) })
```

The `findAll()` method is a Groovy JDK extension that performs collection filtering. In each of the above calls, it effectively applies the given predicate function (in this case a closure) to each word in the list and only retains the word if the function returns `true`. You can run the example to see the result.

The basic form of a closure is the following:

```
{ <args> -> <function body> }
```

So the curly braces delimit the closure literal, similar to the way square brackets delimit lists and maps. This can be a bit confusing because methods, loops, and conditions use curly braces as well. I generally ask myself this: are the curly braces part of an assignment or a method *call*? If so, then I have a closure. Assignments are easy to pick out:

```
def predicateFn = { word -> word.startsWith("p") }
```

Method calls tend to be harder to identify because Groovy supports several syntaxes. I'll explain shortly, but first, one question you may be asking yourself is where the `word` argument comes from. How does it work? It's important to understand that a closure's argument list behaves just the same as a method's. In other words, `word` only exists within the context of the closure itself. It's the responsibility of the `findAll()` method to pass each word as the argument when calling the closure.

To see what I mean, you can try invoking the closure yourself:

```
def predicateFn = { word -> word.startsWith("p") }
predicateFn.call("petals")
```

Executing this in the Groovy console will show a result of `true`, because "petal" starts with a 'p'. What happens if you try to call the closure with more than one argument? The following results in a `MissingMethodException` for `call()` because the number of arguments don't match:

```
def predicateFn = { word -> word.startsWith("p") }
predicateFn.call("petals", "badarg")
```

If it's easy to identify a closure that's assigned to a variable, why is it harder when the closure is an argument of a method call? Let's look at the first `findAll()` call from the list filtering example:

```
words.findAll({ word -> word.startsWith("p") })
```

It's clear that `findAll()` is a method call because of the parentheses and the closure is defined within the parentheses. This makes the closure definition relatively obvious. But this is an uncommon syntax. The following are all more common alternatives:

```
words.findAll() { word -> word.startsWith("p") }
words.findAll { word -> word.startsWith("p") }
words.findAll { word ->
    word.startsWith("p")
}
```

Of these, the second is probably the most common and it doesn't even have any parentheses at all! What's going on? This is simply syntax sugar for a very specific case. If the closure is the last (or only) argument of a method, it can be defined after the parentheses. If it's the only argument, it's common to dispense with the parentheses entirely.

Given this information about closures, can you identify the one in the earlier example where we created a CSV file from a directory listing? If you remember, the script writes the lines of text to the file using the `withPrintWriter()` method:

```
...
final outFile = FS.default.getPath("files.csv")
outFile.withPrintWriter("UTF-8") { writer ->
    for (path in files) {
        writer.println("${path.fileName}, ${Files.size(path)}, ${getCreationDate(path)}")
    }
}
...
```

The closure begins with `{ writer`. Under the hood, `withPrintWriter()` creates or opens the target file and creates a `java.io.PrintWriter` for it using the character encoding specified by the first argument. It then calls the second argument - the closure - passing the newly created writer as it's argument. When the closure returns or throws an exception, `withPrintWriter()` closes the file. For Java developers, it's worth noting that this is the Groovy alternative to Java's *try-with-resources* statement, as that syntax isn't supported by Groovy.

A good way to learn and understand something is through plenty of examples. Two (filtering and writing to a file) aren't really sufficient. To rectify that, I'm going to revisit some examples from earlier chapters and try to rewrite them using closures where appropriate. This should give you an idea of an alternate approach to thinking about coding problems and how to solve them.

4.4. Revisiting earlier examples

A lot of the examples you saw in the previous chapter have more concise and readable implementations once you start thinking in terms of higher order functions. Take the example for counting the number of vowels in a string. Instead of reaching immediately for a loop to solve the problem, think about what we want to achieve: counting characters. Which characters? Vowels. This is a similar problem to filtering a word list, except this time the algorithm is counting. The predicate simply determines whether a particular character should be counted or not. Is there a method available to do this?

If you check out the Groovy JDK documentation, you'll find a `count()` method for `CharSequence`. The problem is that it only counts occurences of a specified string. What we need is the `Iterable.count(Closure)` method (the `groovy.lang.Closure` type is one way to identify a method that accepts closure arguments). Since `CharSequence` is not an `Iterable`, we first have to convert it to one using `iterator()`. We can then perform the count, as demonstrated here:

```
def text = "Jack Rabbit Slims Twist Contest"

final vowels = "aeiou"
final vowelCount = text.iterator().count { vowels.contains(it.toLowerCase()) }

println "Number of vowels: " + vowelCount
```

What was originally 5 lines of a loop is now a single line. The example is also easier to reason about because the code is no longer cluttered with the mechanics of how to count the vowels. The focus here is *what* we want to achieve, not *how*.

Implicit closure argument

One new element introduced by the previous example is the lack of an explicit argument for the `count()` closure. There's no that separates the argument list from the closure body. This is a special syntax that's useful for very short closure implementations, typically one-liners. If you don't specify an argument list, an implicit argument called `it` becomes available.

The implicit variable form only makes sense for methods that pass a single argument to the closure, such as `each()`, `collect()`, and `count()`. It also works with methods that pass no arguments at all, but then `it` will be `null`.

One thing you'll quickly realise is how useful closures are in the world of collections. Consider the first example of section 3.2 [xcheck] calculates some statistics about a list of words using a loop. Although the implementation is efficient in terms of there only being a single loop, it's not obvious at first glance what it's trying to achieve.

A better approach in terms of code comprehension is to use some Groovy JDK extension methods for calculating the maximum, minimum and sum of values in a collection. The key point is that we're not after the maximum word in this case - you could interpret this as the last word alphabetically perhaps - but the longest word. That's where closures come in, as you can see from this alternate implementation of that example:

```
final words = ["orange", "sit", "test", "flabbergasted", "honorific"]

println "Min word length:  ${words.min { it.size() }.size()}"         // <1>
println "Max word length:  ${words.max { it.size() }.size()}"
println "Avg word length:  ${words.sum { it.size() } / words.size()}"
```

① `min()` and `max()` return the relevant collection element, i.e. a word in this case, so we need to call `size()` again to get its length

In other words, we can specify exactly what property of the values in the collection we're interested in by passing a closure to `min()`, `max()`, and `sum()` that extracts the relevant information. If we were interested in the number of vowels in the words instead, we could implement a `vowelCount()` method and use that from the closure:

```
...
println words.sum { vowelCount(it) }
```

The last example I want to revisit is the method for calculating the power of twos. I originally showed you an example based on a for loop, but let's now take a different tack. Consider what result we want: a list of elements of the form 2^x, where `x` is a range of numbers 0 to `n`. Given a sequence of numbers (`0..n`), applying the function 2^x to each of the numbers in that sequence will generate a new sequence containing the required powers of two.

In code form, this looks like

```
List powersOfTwo(int n) {
    return (0..n).collect { 2 ** it }
}
```

The `collect()` method is a general way of mapping the elements in one sequence to a new sequence. For example a list of words to a list of their lengths. Or sequence of integers to a sequence of their doubles. The actual mapping is defined by the closure, which acts as the mapping function.

If you're new to this style of programming, you may wonder how useful such a method could be in your day-to-day development. I think you'll find you use it regularly, along with many of the other closure-based extensions to the JDK. That's certainly been my experience, particularly as we often work a lot with collections.

There is one complication to this whole area of Groovy. Java originally had nothing like the `collect()` method prior to Java 8, but that all changed with the introduction of lambda functions and the Streams API. You now have an alternative. Here's a Groovy version of the powers of two method using that API:

```
import java.util.stream.Collectors

List powersOfTwo(int n) {
    return (0..n).stream().map { 2 ** it }.collect(Collectors.toList())
}
```

There are several important differences between this example and the previous one:

- You have to convert a collection to a stream before you can use the functional methods
- Java's `map()` method is equivalent to Groovy's `collect()`
- Use Java's `collect()` method to convert a stream back to a collection
- Java's streams are lazy (unlike collections), so you can use them to work with infinite sequences
- You can use a Groovy closure wherever Java 8 allows a lambda function

So which API should you use? The Groovy Collections extensions or Java 8's Streams API? If you have the option, I think the Streams API is more flexible and has more standard names for many of its

methods (`map()` vs `collect()`, `filter()` vs `findAll()`, etc.). They are also type safe, which can be very useful if you like to use the `@TypeChecked` or `@CompileStatic` annotations that I discuss later [xcheck] or like your IDE to know what the types of the closure arguments are without explicitly declaring them. The main cost is the small additional complexity of converting to and fro between collections and streams.

Ultimately, both APIs provide an extra avenue of expressiveness for your code. If both satisfy a set of given requirements, just use the one that you're most comfortable with.

I hope I've given you a good idea of just what you can achieve with Groovy when it comes to scripting, and how easy it can be. I've touched on the filesystem and date APIs and explored collections a bit further with closures. All that's left for you to do is explore the API documentation yourself now that you have the information you need to navigate those waters. I'm also hoping that you are at least starting to get a grasp of what closures are and how they can help. You will find extensive use of them throughout this book and in most Groovy code you'll come across, so it's an important topic.

In the next chapter I want to move beyond simple scripting and see how you can incorporate Groovy into a project, such as an command line or web application.