Groovy

Succinctly

by Duncan Dickinson

Syncfusion | Technology Resource Portal

Groovy Succinctly

By

Duncan Dickinson

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Stephen Haunts
Copy Editor: John Elderkin
Acquisitions Coordinator: Morgan Weston, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

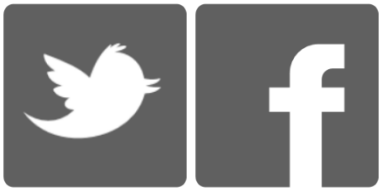
As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Duncan Dickinson (@dedickinson) is an IT systems analyst working in Brisbane, Australia. He started as a developer and has since worked in a range of industries, including the university sector—particularly with systems supporting research data and exploring new models for delivering IT systems. He enjoys work related to agile development, deployment pipelines, and DevOps.

Duncan's interest in approaches to learning and development led him to complete two postgraduate degrees in education and to work as a high school teacher for several years. Returning to the IT profession, he has enjoyed blending technical work with exploring changes that improve the ability of IT teams to deliver useful solutions.

Duncan is the author of The Groovy 2 Tutorial.

Acknowledgments

Groovy is not only a language, it is also an open source community of people contributing in all kinds of ways. I'd like to thank the Groovy community for their ongoing contributions and their friendly nature.

If you're interested in what's going on in the world of Groovy, the Community page on the Groovy website is a good starting point.

I'd also like to thank Syncfusion for giving me the opportunity to write this e-book and share Apache Groovy with all those who like to solve problems.

Introduction

Welcome to Groovy Succinctly! If it's your first time checking out the Groovy language, then welcome to Apache Groovy!

The term "succinct" works well for this e-book, as I'll introduce you to the Groovy programming language in a manner that gets you solving problems quickly. What's more, you'll find that the Groovy language is indeed a succinct one, lacking much of the boilerplate and unnecessary ceremony that make many other languages difficult to read and that require the programmer to become involved in minutia that serves the language but not the solution at hand.

Learning a language involves an investment in time. Of course, it can sometimes seem as though a new language appears every month (this isn't necessarily a bad thing), so why spend your time learning Groovy? Perhaps the following points will encourage you to give Groovy a spin and demonstrate why I have made Groovy my go-to language:

- Groovy has existed for more than ten years. It's not a new language, and it is backed by a friendly open source community.
- Groovy runs on the Java Virtual Machine (JVM), a mature platform for running software, and it can take advantage of the huge number of existing Java libraries.
- Groovy code can be written as a single-file script or a full-blown application.
- Groovy is used across a wide range of systems, including the Gradle build tool, the Grails web application framework, and the Jenkins Workflow plugin. What's more, Groovy makes it easy to write your own Domain Specific Language (DSL).

This book will introduce you to Groovy over the following primary sections:

- **Chapter 1 Getting Started** guides you through installing Groovy.
- **Chapter 2 Language Essentials** provides you with an introduction to a subset of the Groovy language that will get you working quickly.
- **Chapter 3 Solution Fundamentals** describes approaches to common programming activities such as working with files and databases.
- **Chapter 4 Data Streams** focuses on techniques for processing data using Java's new Streams library.
- **Chapter 5 Integrating Systems** demonstrates how Groovy can work with Apache Camel to help you integrate disparate systems.
- **Chapter 6 Larger Applications** introduces the Gradle build tool in order to go beyond scripts.

- **Chapter 7 Next Steps** will provide you with a number of key Groovy-related technologies that you can pursue as you go beyond the basics.

Who is this book for?

This book is intended for people with at least a basic level of programming skills. Although this book is not a “how to program” text, I have tried to pitch the text and code samples so that readers new to programming or who come from another programming language can get useful, working programs going quickly. Experience in Java is not necessary, and I don’t spend any time comparing Java and Groovy approaches—there are already several books and blogs that discuss this.

In these days of DevOps and automation, I’m mindful that the term “programmer” is much less specific than it used to be. My hope is that network engineers or systems administrators will find Groovy useful in their daily work, so don’t feel excluded if you’re not a full-time programmer. If you can use the command line on your Windows, OS X, or Linux system, you should find this instruction easy to follow.

Groovy code can be anything from a single script to a full-blown application, and it is an ideal language for quickly preparing code to work with a database, processing JSON from a website, and quickly prototyping integration between two systems. I also use Groovy to create desktop user interfaces with JavaFX and websites with Grails. For larger development work, I turn to Gradle and define my build files (and the whole application) in Groovy. Of late, I've been using Groovy to define my Jenkins pipelines.

By giving Groovy just a small amount of your time, you might (re)discover your desire to automate those daily chores that use up your precious time.

Code examples

All of the nontrivial code examples can be found in the accompanying Bitbucket repository: https://bitbucket.org/syncfusiontech/groovy-succinctly. If you’re comfortable with Git you can clone a copy for your own endeavors or download the repository to your local system.

In order to aid you in trying out the sample code, I've aligned many of the code listings with their associated code file in the Appendix: Code Listings section.

The Groovy Console packaged with the Groovy installation will be fine for trying out the code examples in this e-book. A basic text editor such as Atom (with the language-groovy package) will also serve you well, but you’ll need to switch between the editor and the command line in order to run the code. I use JetBrains’ IntelliJ IDEA as my IDE of choice for larger projects, and the Community Edition is free if you want to try that. The NetBeans IDE also provides support for Groovy.

Note: The Eclipse IDE's support for Groovy has slipped since development on the Groovy/Grails Tool Suite™ was discontinued. If you're an Eclipse user, try checking the Groovy-Eclipse project.

For the sake of conciseness, much of the code here will be focused on looking at a specific types of data. Most of the sample data is centered on a theme of weather readings such as maximum and minimum temperatures and rainfall. Because I come from Australia, these measurements use Celsius (temperature) and millimeters (rainfall), but the examples can be easily reimagined using other units. Weather recordings are interesting because they map well to large data processing and align with recording equipment we would include in the Internet of Things. At their heart, the examples are about numbers and how Groovy makes it easy to process data.

Chapter 1 Getting Started

Installing Groovy

In order to work through the example code in this book, you'll need to get two items running on your system:

- The Oracle Java SE Development Kit (JDK) 8
- Apache Groovy 2.4.5

Installing these should be quite easy for both Microsoft Windows and Apple OS X users.

Java JDK 8

The downloads for Java 8 are available via the Oracle website. Make sure that you download the variation for your operating system and that you've selected the JDK and not the Java Runtime Environment (JRE).

Note: While Groovy is happy running on Version 7 of the JDK, the examples in the Data Streams section rely on libraries found in JDK Version 8.

The JDK 8 installation is straightforward, so I won't take you through the process here—just follow the installer's instructions. Additional help can be found on the Start Here page.

Check your installation by typing **`java -version`** in the command prompt (Windows) or the terminal (OS X). You should see output similar to the following:

```
java version "1.8.0_31"

Java(TM) SE Runtime Environment (build 1.8.0_31-b13)

Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
```

Groovy 2.4.5

The Groovy download page provides access to the various Groovy distributions.

Windows users should start with the Windows installer, as it will guide you through the installation process.

OS X users can use the excellent SDKMAN! tool for installing not only Groovy but a number of other useful systems such as Gradle and Grails. The basic process for installing Groovy using SDKMAN! via the console is provided in Code Listing 1: Installing Groovy with SDKMAN! Refer to the Install page of the SDKMAN! website for more information.

Code Listing 1: Installing Groovy with SDKMAN!

```
$ curl -s get.sdkman.io | bash
$ source "$HOME/.sdkman/bin/sdkman-init.sh"
$ sdk install groovy 2.4.5
```

Once you've installed Groovy, check your installation by typing **`groovy -version`** in the command prompt (Windows) or the terminal (OS X). You should see output similar to the following:

```
Groovy Version: 2.4.5 JVM: 1.8.0_31 Vendor: Oracle Corporation OS: Mac OS X
```

Hello, world

Now that you have JDK 8 and Groovy 2.4.5 installed, you're ready to try out a small Groovy script. We'll use the Groovy Console to write the script and run it. Windows users will find a link to the Groovy Console in their start menu. OS X users can access the Groovy Console by running **`groovyConsole &`** in the terminal. You should now see the Groovy Console on your screen (refer to the screenshot in Figure 1).

The top half of the Groovy Console window is a text editor with syntax highlighting—this is where you write your code. The lower half of the window displays the output of your Groovy code.

Let's get coding:

1. In the text editor section, type: **`println 'hello, world'`**. The "**`println`**" element is a function that displays (prints) the supplied text (**`'hello, world'`**) to the screen.

2. Go to the Script menu and click Run.

3. The output of your script should appear in the lower half of the window.

The screenshot in Figure 1 displays the code in the text editor section and its output in the lower half.

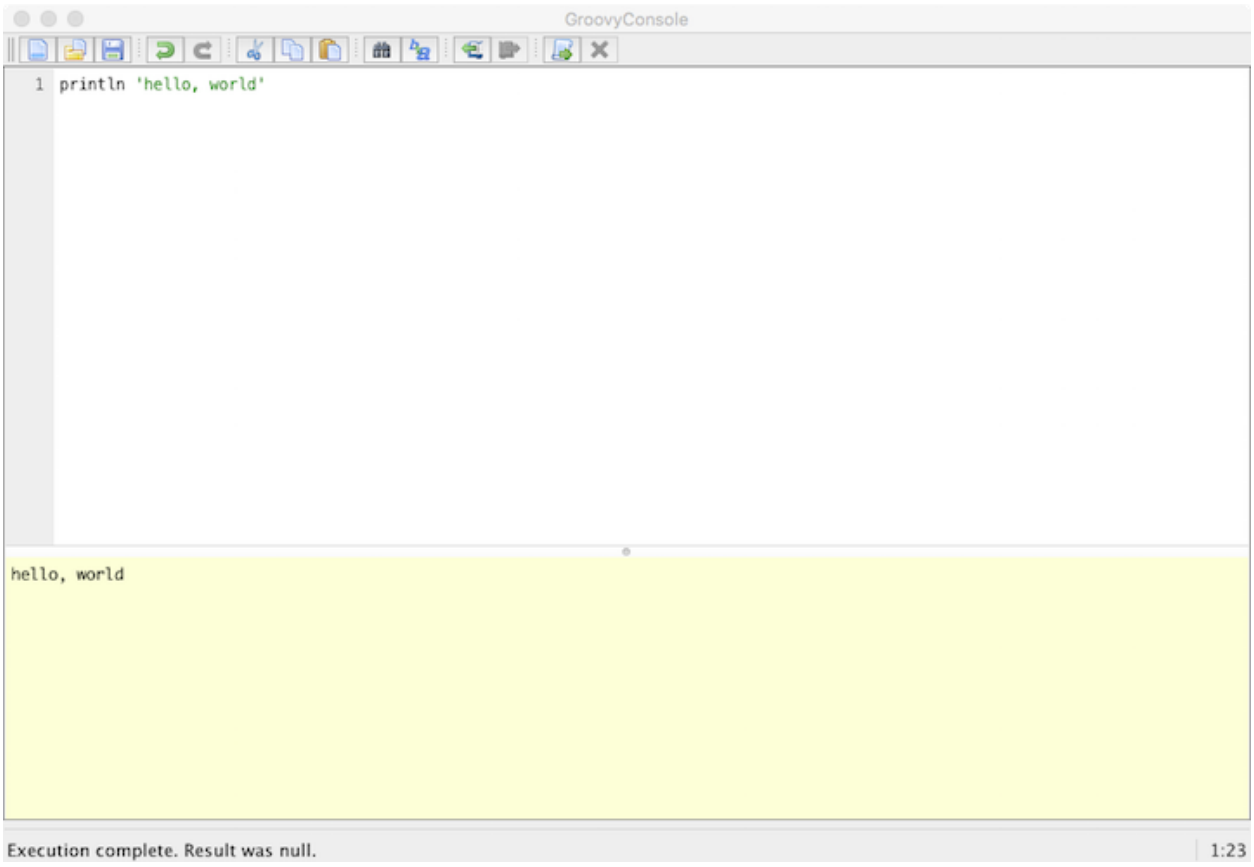


Figure 1: The Groovy Console

You can save the script by selecting Save from the File menu. Save your script using the filename **`HelloWorld.groovy`**.

Open your terminal/command prompt and change the directory to denote where you just saved your script. You can now run your script with the command **`groovy HelloWorld.groovy`**.

The examples provided in the rest of this e-book, with the exception of the Chapter 6 Larger Applications chapter, will work in the Groovy Console. This is a great tool for getting started. Those who wish to use an alternative text editor can use the **`groovy`** command-line tool.

Chapter 2 Language Essentials

In order to get working in Groovy, you need to learn only a surprisingly small subset of the language. This section will outline all of the Groovy syntax you'll need in order to work through the examples in the following sections. Use this chapter to get a feel for the language and pick up enough syntax to get started.

Comments

While they don't actually direct Groovy to do anything, comments are extremely useful in helping explain your code to others. Groovy supports two main types of comments:

- Line comments start with two forward slashes (**//**) followed by a very brief piece of text. They can appear on a line by themselves or at the end of a statement.
- Block comments open with a forward slash and an asterisk (**/***) and are closed by the reverse combination (***/**). These comments usually span multiple lines, and it is customary to start each line with an asterisk (*****) that lines up with the opening and closing asterisks.

Code Listing 2: Comments provides examples of line and block comments.

Code Listing 2: Comments

```
//This is a line comment

def answer = 1 + 1 //This is another line comment

/*
 * This is a block comment.
 * They can be useful for larger bodies of text
 */
```

Values

Groovy handles a range of data values:

- Booleans
- Numbers
- Strings
- Lists

- Ranges
- Maps

Booleans

A Boolean value can be either **`true`** or **`false`**.

Numbers

There are two primary types of number in Groovy:

- Integers (whole numbers): **`1`**, **`83`**, **`401`**, **`-56`**, **`-1000`**
- Decimal numbers: **`3.14`**, **`-273.15`**

Note: A handy option for large numbers is to use an underscore (`_`) to help break up the segments: `1_000_000`.

Strings

A string is a piece of text surrounded by single quotes (**`'...'`**). We used a basic string earlier when we typed **`'hello, world'`**. If we need to use a single quote in a string, it must be delimited using the backslash (**`\`**), e.g. **`'It\'s Jim here'`**.

For longer strings we use three single quotes (**`'''`**) to open and close the text.

Code Listing 3: Longer Strings

```
'''\
groovy makes it easy
less boilerplate to create mess
solve the problem quick
'''
```

Note: The backslash in the top line of the haiku indicates that the first line continues into the second, avoiding a blank line at the top of the string.

Groovy also provides a special type of string, called a Groovy String, surrounded by double quotes (**`"..."`**) for small strings and three double quotes (**`"""..."""`**) for a multiline string. The Groovy String is used for interpolation of variables—we'll look at this shortly.

Lists

A list holds zero or more values. A list value is surrounded by square brackets (**`[ ]`**), and each item is separated by a comma (**`,`**):

- An empty list: **`[]`**

- A list of numbers: **`[ 0, 4, 3, 7, 2, 4, 8 ]`**
- A list of strings: **`[ 'temperature', 'rainfall', 'humidity' ]`**

Groovy is a relaxed language and doesn't demand that all of the list elements be of a similar kind. This means that you can mix different types of items in a list: numbers, booleans, strings, etc.

Lists are said to be zero indexed because the first item in the list has an index of 0. In the list **`[ 2, 4, 8 ]`**, the number 2 is at index 0, and the number 4 is at index 1.

Ranges

Similar to lists, ranges define a series of numbers. A range is defined by providing a starting number, two full stops (**`..`**), and a final number. This format allows a range to increment or decrement:

- Counting up: **`0..10`**
- Counting down: **`10..0`**

Essentially, ranges are a quick way to prepare a list without explicitly providing each value. This is illustrated in a basic assertion: **`assert 1..3 == [ 1, 2, 3 ]`**

Maps

A map is a structure that consists of key-value pairs. Similar to a list, a map is surrounded by square brackets (**`[ ]`**), and each item is separated by a comma (**`,`**). Each item in the map is a key-value pair and is declared with the key first, followed by a colon (**`:`**), then the value. A map's keys are unique and are usually strings.

Here are some sample map values:

- An empty map: **`[:]`**
- A basic map: **`[ day: 'Monday', rainfall: 3.4, maxTemp: 31 ]`**
- Using quoted keys: **`[ 'day': 'Monday', 'rainfall': 3.4, 'maxTemp': 31 ]`**

As demonstrated in the basic map example, the key doesn't have to be quoted, provided the key isn't one of Groovy's reserved keywords; this helps readability. However, this doesn't work for strings that include elements such as punctuation, and you'll need to use quoted strings, as demonstrated in Code Listing 4: Map Keys.

Code Listing 4: Map Keys

```
[ 'Maximum temperature': 32,
  'Minimum temperature': 18 ]
```

Variables

Variables are used to hold values for use through a program. In order to declare a variable, the **def** keyword is followed by the name of the variable. Code Listing 5: Declaring Variables provides a series of variable declarations.

Code Listing 5: Declaring Variables

```
def greeting
def dailyRainfall
def personalDetails
```

Tip: You'll note that I've used an approach to formatting variable names in which the first word is in lowercase but any subsequent words start with an uppercase letter—this is called lower camel case.

A variable will start off with a value on many occasions, and we do this in the declaration by using the assignment operator (**=**). This is especially useful in the case of lists and maps because the empty form of each (**[ ]** and **[:]**) advises Groovy how we intend to use the variable. Code Listing 6: Assigning Values at Variable Declaration provides a number of examples.

Code Listing 6: Assigning Values at Variable Declaration

```
def greeting = 'hello, world'
def pi = 3.14
def honest = true

def emptyList = [ ]
def dailyRainfall = [ 0, 4, 3, 7, 2, 4, 8 ]

def emptyMap = [ : ]
def weatherDetails = [ day: 'Monday', rainfall: 3.4, maxTemp: 31 ]

def countdown = 10..0
```

Note: In this book I'm primarily using Groovy's ability to handle variables in a dynamic fashion. However, Groovy also lets me be explicit about variable types.

Groovy Strings

As we saw in the section on Values, a Groovy String is surrounded with double-quotes (**"..."**) and provides variable interpolation. This means that we can use variable names, prefixed with the dollar (**$**) symbol, in the string, and Groovy will substitute in the variable's value. Code Listing 7: Groovy String in Action demonstrates this using a multiline Groovy String.

Code Listing 7: Groovy String in Action

```
def language = 'Groovy'
def groovyHaiku = """
run $language on jvm
a multiplatform solution
for fun and profit
"""
println groovyHaiku
```

Groovy String interpolation allows for more complex expressions by enclosing the expression in the dollar symbol and curly braces (**${ }**), as we can see in Code Listing 8: Expressions in Groovy Strings.

Code Listing 8: Expressions in Groovy Strings

```
assert "Total score: ${10 + 9 + 10 + 8}" == "Total score: 37"
assert "Area: ${Math.PI * 3**2}" == "Area: 28.274333882308138"
```

Note: I'll describe the `assert` statement shortly—it's just a check to make sure a statement is producing (returning) the correct value.

Lists

Once a list has been declared and items added, we can use index notation to access list items. As demonstrated in Code Listing 9: Accessing List Items, the desired index is placed within square brackets (**[ ]**). Remember that lists are zero-indexed so index 4 is actually the fifth list item. As a handy extra, Groovy lets you use a negative index to access items for the end of the list—e.g., an index of -2 indicates the second-to-last item in the list.

Code Listing 9: Accessing List Items

```
def dailyRainfall = [ 0, 4, 3, 7, 2, 4, 8 ]
assert dailyRainfall[4] == 2
assert dailyRainfall[-2] == 4
```

Maps

Values stored in a map are accessed using dot-point notation such as **`dailyRainfall.Friday`**, which returns the value stored against the **`friday`** key in the **`dailyRainfall`** map. Groovy Strings is particularly useful when providing a key because we can resolve the value of a variable that holds the key (e.g., **`dailyRainfall."$day"`**). Code Listing 10: Accessing Map Items demonstrates these approaches as well as using a multiline declaration for the map, a readable approach to presenting the key-value pairs.

Code Listing 10: Accessing Map Items

```
def dailyRainfall = [ monday    : 7,
                      tuesday   : 5,
```

```
                        wednesday: 19,
                        thursday : 6,
                        friday   : 11,
                        saturday : 8,
                        sunday   : 0 ]

assert dailyRainfall.friday == 11

def day = 'friday'
assert dailyRainfall."$day" == 11
```

Types

Let's avoid going into too much detail on variable types —we can achieve a lot with only a cursory understanding. However, there are occasions in which we need to manipulate a value's type so that the variable stores the value correctly. We often encounter this when we read text from a file—all of the input comes to us as a string, and we may need to tell Groovy to change the value to a number. This is achieved through a process known as "casting," and in Groovy we use the **as** keyword followed by the required type.

Groovy also provides a range of default methods that let us convert to a specific type. These methods are named using the "**to**" prefix followed by the type name. For example: **toString()**, **toInteger(),** and **toBigDecimal()**. These methods are very useful when we convert between number types and between strings and numbers.

Code Listing 11: Casting to Another Type

```
def highway = 66.toString()
def lucky = '13'.toInteger()
pi = '3.14'.toBigDecimal()

def days = [ 'monday', 'tuesday', 'monday' ] as Set
assert days == [ 'monday', 'tuesday' ] as Set
```

In Code Listing 11: Casting to Another Type, we see this order of actions:

1. The number **66** is converted to a **String.**
2. The string **'13'** is converted to an **Integer** (an **Integer** is a "whole number").
3. The string **'3.14'** is converted to a **BigDecimal** (the BigDecimal type is a type that makes it easy to deal with decimal numbers).
4. A list of days is cast to a **Set** (this removes all duplicate list items).

Some of the vast library of types available for use will be discussed in the Classes and objects section.

Assertions

An assertion checks that a statement is true. If the statement is found to be false, Groovy halts processing and displays an error. This is helpful for the code examples in this e-book because it helps illustrate an expected outcome. You should be careful with asserts in a Groovy script or application—they will cause your program to halt, but they are very useful in checking your program. Code Listing 12: Some Correct Assertions provides a few basic uses of **`assert`**.

Code Listing 12: Some Correct Assertions

```
assert 1 + 1 == 2
assert 10 * 2 == 20
assert 10 ** 2 == 100
```

To set up an assertion, the **`assert`** keyword is followed by a statement that evaluates to a Boolean (true or false) result. For example, if I run **`assert 1 + '1' == 2,`** my code will fail and Groovy will display the error shown in Code Listing 13: A Failed Assertion Example.

Code Listing 3: A Failed Assertion Example

```
Assertion failed:

assert 1 + '1' == 2

         |     |

         11    false
```

The output of the assertion error is very useful, and we can see that **`1 + '1'`** returns **`11`** and not **`2`**. Groovy illustrates what went wrong:

- **`1 + '1'`** mixed a number and a string, so Groovy concatenated the two into a single string (**`'11'`**).
- When **`'11'`** was compared for equality (**`==`**) with **`2`**, the statement was found to be **`false`**.

Operators

We just saw that the equality operator (**`==`**) is used to compare two values and return true when they are equivalent. The inequality operator (**`!=`**) provides the opposite functionality and returns true when the two values are different.

The numeric operators are known by most school students and are listed in Table 1.

Table 1: Numeric Operators

Group	Operators	Examples
Additive operators	Plus (`+`), minus (`-`)	`1 + 2 == 3` `100 - 9 == 91`

Multiplicative operators	Multiply (`*`), divide (`/`), modulo (`%`)	`10 * 2 == 20`
		`10 / 2 == 5`
		`10 % 3 == 1`

Power operator	`**`	`10 ** 2 == 100`

Note: Groovy supports operator overloading, which allows you to define how the various operators work. This is useful for your own custom classes, but it can also modify existing operator implementations.

A number of other operators are available, including:

- Relational operators: used to compare two operands and return a Boolean (true or false) result—e.g., `assert 10 > 9 == true`.
- Conditional operators: used to evaluate the "truth" of an expression, returning `true` if the condition is met or `false` if it is not.
- The increment and decrement operators: used with numbers, they return the next or previous value respectively. These operators can be used as a prefix (evaluated before the expression) or as a postfix (evaluated after the expression).

Table 2 provides a list of these commonly used operators.

Table 2: Other Operators

Group	Operators	Examples
Relational	Greater than (**>**) Less than (**<**) Greater than or equal to (**>=**) Less than or equal to (**<=**) Spaceship (**<=>**)	**10 > 5** **5 < 10** **9 >= 9** **8 <= 11**
Conditional	And (**&&**) Or (**||**) Ternary (**? :**)	**true && true == true** **true || false == true**
Increment & Decrement	Increment (**++**) Decrement (**--**)	**10++** **++10** **8--**

Several of the operators mentioned in the table above need more context. First, the spaceship operator compares two operands and returns a result as follows:

- If both operands are the same, zero (0) is returned.
- If the left operand is less than the right operand, negative one (-1) is returned.
- If the left operand is greater than the right operand, positive one (1) is returned.

This seems a little obtuse, even when seen in action in Code Listing 14: Results from the Spaceship Operator. For the most part, the spaceship operator is used when performing a sort—something we'll come across in the Chapter 4 Data Streams chapter.

Code Listing 14: Results from the Spaceship Operator

```
assert 10 <=> 10 == 0
assert 9 <=> 10 == -1
assert 11 <=> 10 == 1
```

The next item of interest is the ternary (or conditional) operator (**? :**), so named because it has three operands (**op1 ? op2 : op3**). The first operand (op1) is a boolean condition. If **op1** is true, then **op2** is returned; otherwise, **op3** is returned. Code Listing 15: The Ternary Operator shows two example uses, but we'll also explore the ternary operator in later chapters.

Code Listing 15: The Ternary Operator

```
def witness = true ? 'honest' :'liar'
assert witness

assert 10 > 2 ? 'Ten is greater' :'Two is greater' == 'Ten is greater'
```

I didn't list the left shift operator (**<<**) in Table 2: Other Operators, but it is very useful and adapts nicely to the context in which it is used:

- Adding an item to a list:
    - **[2, 4, 6] << 8 == [2, 4, 6, 8]**
- Adding an item to a map:
    - **[day: 'Sunday', rainfall: 8.1] << [maxTemp: 32] == [day: 'Sunday', rainfall: 8.1, maxTemp: 32]**

While you can use the left shift operator to concatenate strings, it's often neater to use Groovy String's ability to interpolate variables. Code Listing 16 depicts and example of this.

Code Listing 16: Easy String Concatenation

```
def str1 = 'hello,'
def str2 = 'world'

assert "$str1 $str2" == "hello, world"
```

Classes and objects

Everything we interact with in Groovy is an object, so let's sort out a few key concepts:

- A class describes a structure (type) for use in programs. For example, **Integer**, **String**, and **BigDecimal** are classes provided as part of the Java library and can be used by Groovy programmers.
- A class declaration includes the data and methods that are available to individual instances of a class.
    - These instances are referred to as "objects," and the act of creating a new object is called "instantiation."

    - An object's data consists of variables that keep track of the object's state and are commonly referred to as "fields" or "properties."
    - An object's methods provide functionality to interact with the object's data.
- Classes can also provide data and methods that can be accessed directly.
    - These are commonly referred to as "class fields" and "class methods" and are marked with the **`static`** modifier.
    - For example, the **`Math`** class provides a field called **`PI`** which contains a value for the mathematical concept of Pi (π).
- Class and object fields can be configured as constants to prevent their value from being changed.
    - These are marked with the **`final`** modifier.
    - For example, the **`Math.PI`** field is marked as final so that we can't decide on a different interpretation of this well-known value.

Note: I'll be straight with you—we're not going to write many of our own classes. We can achieve a lot without taking that path. I've mentioned aspects of syntax here to aid you when looking at the documentation for existing classes.

Let's turn this theory into some real code!

Code Listing 17: Using Class Fields and Methods demonstrates that we use the class name (e.g., **`Math`**) followed by a full stop (**`.`**), then the name of the field (**`PI`**) or method (**`max`**) we wish to use. Use of the full stop is similar to what we saw with accessing map keys and is referred to as "dot notation."

Code Listing 17: Using Class Fields and Methods

```
println Math.PI
println Math.max(8, 9)
```

*Code Listing 18: Instantiating an Object demonstrates the use of the **new** keyword to create an instance (object) of the **Random** class. The new object is then stored in the **random** variable, and, through that variable, we can access the object's methods (e.g., **nextInt**).*

Code Listing 18: Instantiating an Object

```
def random = new Random()
println random.nextInt(100)
```

Calls to class and object methods can require arguments and can return a result. For example, **`Math.max(8, 9)`** has two arguments (8 and 9) and returns the greater of the two (9). The call to **`random.nextInt(100)`** takes one value (100) and returns a random integer between 0 (inclusive) and 100 (exclusive).

Individual arguments are separated by a comma (**`,`**). Use of the parentheses enclosing the arguments is optional in Groovy, and Code Listing 19: Arguments without Parentheses demonstrates a few things:

- The **`Math.max 8, 9`** call is the same as **`Math.max(8, 9)`**.
- **`println`** is actually a method that can be passed a string and is often called without parentheses.
- A method call with no arguments (such as **`println()`**) requires the parentheses.

Code Listing 19: Arguments without Parentheses

```
def highest = Math.max 8, 9
println 'hello, world'
println()
```

A special type of method, referred to as a "constructor," is called when instantiating a new object. The call to **`new Random()`** is a call to the no-argument constructor provided by the **`Random`** class. Some classes provide constructors that accept arguments because these help define the object when it's instantiated.

In Code Listing 20: A Basic Groovy Class, I have declared a class that describes a weather station. The **`class`** keyword commences the declaration, followed by the class name (**`WeatherStation`**). Note the use of upper camel case for the class name. The class definition is enclosed in curly braces (**`{ }`**).

The **`WeatherStation`** class has a number of properties:

- **`id`**, **`name`** and **`url`** properties are declared using **`def`** but could be given a specific type (e.g., **`String`**).
- **`latitude`**, **`longitude`**, and **`height`** properties are each declared as **`BigDecimal`**.
- **`tempReadings`** is actually defined with **`def`**, but this can be omitted when using the **`final`** keyword. By using **`final`**, I indicate the **`tempReadings`** will be set to a list but can't be changed. This means that other code can't change **`tempReadings`** to hold another value, but the contents of the list can be changed.

Code Listing 20: A Basic Groovy Class

```
class WeatherStation {
    def id
    def name, url
    BigDecimal latitude, longitude, height
    final tempReadings = []

    String toString() {
        "$name(#$id): $latitude lat, $longitude long, $height height"
    }

    def addTempReading(temp) {
        tempReadings << temp
    }

    def getAverageTemperature() {
        tempReadings.sum() / tempReadings.size()
    }
}

def davisStation = new WeatherStation(id: 300000, name: 'DAVIS',
        latitude: -68.57, longitude: 77.97, height: 18.0,
        url:
'http://www.bom.gov.au/products/IDT60803/IDT60803.89571.shtml')

davisStation.addTempReading 5
davisStation.addTempReading 3
davisStation.addTempReading 7
davisStation.tempReadings << 2

println davisStation.toString()
println davisStation
println "$davisStation"
println "Station url: ${davisStation.getUrl()}"
println "Average temperature: ${davisStation.averageTemperature}"
```

`WeatherStation` also has a number of instance (object) methods:

- The **`toString()`** method customizes the standard method provided on all Groovy objects. It returns a **`String`** value (hence **`String`** appears before the method name) and has no parameters. This method returns a human-readable description of the object. Groovy methods return the result of the last statement which, in this case, is a Groovy String.

- The **`addTempReading(temp)`** method declares a single parameter (**`temp`**) and appends it to the **`tempReadings`** list.

- The **`getAverageTemperature()`** method has no parameters but returns the average of the **`tempReadings`** list.

You'll notice that the **def** keyword is prefixed to the last two method declarations. This is used in much the same manner as when we declare a dynamic variable and indicate that the method may return a value of any type. Additionally, we need some syntax to indicate that we're declaring a method when we don't want to specify the return type (as opposed to **String toString()**)—**def** performs that role. You may be wondering how to recognize what a method will return if it is declared with **def**. This can often be deduced by the method name (e.g., you'd expect **getAverageTemperature** to return a number) or its documentation. In the case of most nontrivial code, you're likely to explicitly provide the return type of the method.

***Note:** `WeatherStation` consists of no class methods or class fields—they're not needed for this example.*

The **WeatherStation** class has now been declared, so we're able to create instances of it. Remember that the **new** statement is used to create an instance of a class and the **new WeatherStation** statement creates a new instance to be held by the **davisStation** variable. Groovy helps us here with classes and generates a constructor for us that takes a set of key/value pairs for the object properties. If this looks like the map notation to you, you're absolutely right—we can pass the constructor a map containing keys with the same name as the object properties and their associated value. Groovy will tidy the syntax for us by removing the need to enclose the map in square brackets (**[ ]**). You don't need to set all of the properties, only the ones that get you started.

Now that I've instantiated the object, I can add some temperature readings using the **addTempReading** method: e.g., **davisStation.addTempReading 5**. You'll notice that I can also append a value using **davisStation.tempReadings << 2** and could have, in fact, not bothered to provide an **addTempReading** method at all!

I can call the **toString()** method in a few ways. The first method is explicit: **println davisStation.toString()**. The second call (**println davisStation**) demonstrates that Groovy determines the **println** method needs the String version of the object and therefore calls the **toString()** method. Similarly, the third method demonstrates that interpolating an object in a Groovy String causes the **toString()** method to be called. This gives you a number of options, and you'll find that each approach is useful in different contexts.

Because I never declared a **getUrl()** method, the **davisStation.getUrl()** call may seem a little odd. Groovy provides getters and setters for all of the properties, which avoids the boilerplate **getX** and **setX** methods you see in Java code. All of the properties I have declared in **WeatherStation** are automatically given an associated **set** and **get** method. In fact, if I'd accessed the property with **davisStation.url**, it would have called the **getUrl()** method. This means that I could explicitly provide the **getUrl()** method if I want to—perhaps to verify that the URL is correctly formed.

Lastly, accessing **davisStation.averageTemperature** will look like I'm accessing a property rather than calling a method. This is another useful Groovy trick: methods with a **get** prefix and no parameters or a **set** prefix with one parameter can be accessed as if they're properties. Hence, **davisStation.averageTemperature** will actually call the **getAverageTemperature** method. This features works not just for our own Groovy methods but for those you'll find in existing Java libraries as well.

The small **WeatherStation** example demonstrates that Groovy will let us declare classes in a very succinct manner. Furthermore, when accessing objects, we can use a property approach rather than have **get** and **set** calls throughout our code.

Enumerations

An enumeration is a special type of class used primarily to provide a set of constants. This is a different concept than **Math.PI** in that the constants form a set of values. These values may form a series in which one constant follows another, but that's not a requirement. A good example of a series is the Month enumeration that lists the twelve months of the year.

The constants within an enumeration are referred to in the same manner as class fields, e.g., **Month.FEBRUARY**. Enumeration constants make comparisons such as equality (**==**) easier because they aren't strings and therefore don't suffer from mismatched case or spelling.

We'll mainly utilize existing enumerations in this book, but Code Listing 21: A Basic Enumeration demonstrates an enumeration with two constants (**RAINFALL** and **TEMPERATURE**). You'll note that we don't use the **new** keyword when assigning an enum constant to the variable—this is because they're more like class constants. You can imagine that the **ReadingType** enum would be useful when you come across two readings and want to ensure that they're a measurement of the same aspect of the weather.

Code Listing 21: A Basic Enum

```
enum ReadingType {
    RAINFALL, TEMPERATURE
}

def readingType = ReadingType.RAINFALL
```

Closures

Closures are similar to methods, but they aren't bound by a class definition. Groovy's closures are first-class functions because they can be passed as method arguments, returned from methods, and stored in variables. This is a very powerful concept—one we'll make a lot of use of in the coming chapters.

The basic syntax for a Groovy closure is one or more statements enclosed in curly braces (**{ }**). Code Listing 22: A Basic Closure shows a variable called **square** set to be a closure that can then be called using **square(4)**. The really interesting thing here is that the closure is more like a value (such as a number or a string), and we can pass the closure around to other closures and methods—more about this shortly.

Code Listing 22: A Basic Closure

```
def square = { it**2 }
assert square(4) == 16
```

You may have also noticed the use of a variable named **it**. By default, closures have a single parameter (**it**), and the **square** closure will find the square of the argument passed to **it**: **{ it**2 }**.

Note: An argument is the value passed to the parameter of a function (i.e. a method or a closure).

As we saw in methods, closures return the result of their last statement. Often provided in a one-line form, this allows closures to be written in a very succinct manner. However, we can also use the **return** keyword to explicitly designate a value to be returned. We'll see an example of this momentarily.

Tip: Most closures are likely to be made up of only a small number of statements—it's best to keep them small and easy to read.

While a closure has a parameter named **it** by default, we can also explicitly designate a parameter. In Code Listing 23: A Single-Parameter Closure you'll see an extension to the curly brace syntax with the addition of an "arrow" (**->**). The parameter list is provided to the left of the arrow, and the body (statements) of the closure is provided to the right. A single parameter (**num**) is declared in the closure assigned to the **triple** variable.

Code Listing 23: A Single-Parameter Closure

```
def triple = { num -> num * 3 }
assert triple(3) == 9
```

Note: When a closure is defined with a parameter list, the default parameter (`it`) is no longer available.

A closure can have more than one parameter, and Code Listing 24: A Two-Parameter Closure demonstrates the use of the comma (**,**) to list the parameters. In the case of the **max** closure, the greater of the two arguments is returned.

Code Listing 24: A Two-Parameter Closure

```
def max = { arg1, arg2 ->
    arg1 > arg2 ? arg1 :arg2
}
assert max(8, 9) == 9
```

Finally, a closure can be declared with no parameters to the left of the arrow so as to define a closure without access to any parameters, including the default **it** parameter.

Code Listing 25: A Closure with No Parameters

```
def noArgClosure = { -> println 'hello, world' }
noArgClosure()
```

The previous examples have been short and simple—a sound approach to closures. However, closures can be more complicated, and Code Listing 26: A Quadratic Equation Closure accepts three arguments and returns a list containing the two possible solutions for the quadratic equation. You'll also note the explicit use of the **return** keyword to designate the value being returned from the closure.

Code Listing 26: A Quadratic Equation Closure

```
def quadratic = { a, b, c ->
    def denominator = 2 * a
    def partialNumerator = Math.sqrt((b**2) - (4 * a * c))
    def answer1 = ((-1 * b) + partialNumerator) / denominator
    def answer2 = ((-1 * b) - partialNumerator) / denominator
    return [ answer1, answer2 ]
}

assert quadratic(1, 3, -4) == [ 1, -4 ]
```

That implementation of the quadratic equation was rather step-by-step and could be rewritten in a number of ways. In Code Listing 27: A Revised Quadratic-Equation Closure, I've simplified the quadratic closure (now called **quadratic2**) to simply declaring a list containing the two results from the calculations. I also broke out part of the numerator calculation into its own closure (**quadraticPartialNumerator**).

Ordinarily, a single call to **quadratic2** would cause two calls to **quadraticPartialNumerator**, but I've appended **.memoize()** to the latter's declaration. The **memoize** method[1] creates a cache which holds a list of results for previously submitted parameters. Once a result is calculated, it is stored, and any future call to **quadraticPartialNumerator** with the same values for **a**, **b**, and **c** will avoid the calculation and take the result from the cache. There's no reason you can't also memoize **quadratic2.**

Code Listing 27: A Revised Quadratic-Equation Closure

```
def quadraticPartialNumerator = { a, b, c ->
    Math.sqrt((b**2) - (4 * a * c))
}.memoize()

def quadratic2 = { a, b, c ->
    [ ((-1 * b) + quadraticPartialNumerator(a, b, c)) / 2 * a,
      ((-1 * b) - quadraticPartialNumerator(a, b, c)) / 2 * a ]
}

assert quadratic2(1, 3, -4) == [ 1, -4 ]
```

[1] For more information, check out the Groovy doc for Closures.

Tip: It's best to present readable (maintainable) code as the initial strategy. Certain efficiency gains will be obvious but don't go overboard too early—you can always profile and refactor your code later if it runs slowly.

Closures as arguments

Just as numbers and strings can be passed as arguments to methods and closures, Groovy lets you also pass closures as arguments. This is not only useful, but it can really help keep the code succinct.

We've seen that arguments can be optionally enclosed in parentheses, but how do we include a closure as a parameter? Groovy gives us a number of approaches, and we'll explore these through a call to the extremely useful **`each`** method that's available on lists and maps. The **`each`** method is passed as a closure that is called for each item in the list/map.

The first approach to passing a closure parameter is to simply pass in a variable to which a closure has been assigned. For example, in Code Listing 28: Closure Variable as Parameter, the **`printer`** variable is passed to the **`each`** method, which results in each item in **`dailyRainfall`** being displayed to the screen, one item per line.

Code Listing 28: Closure Variable as Parameter

```
def dailyRainfall = [ 0, 4, 3, 7, 2, 4, 8 ]
def printer = { println it }
dailyRainfall.each printer
```

Pre-assigning closures to variables can get annoying, especially if a closure is only to be used once. Luckily, we can provide the closure inline to the method call. Code Listing 29: Passing Parameters illustrates three ways in which the **`each`** method can be called:

1. The first approach passes the closure block within the method call's parentheses: **`({ println it })`**.

2. The second approach places the closure block outside the method call's parentheses: **`(){ println it }`**. This is useful if the method call takes other arguments.

3. The final approach is to declare the closure inline. This is the idiomatic approach used in most Groovy code.

Code Listing 29: Passing Parameters

```
def dailyRainfall = [ 0, 4, 3, 7, 2, 4, 8 ]
dailyRainfall.each ({ println it })
dailyRainfall.each (){ println it }
dailyRainfall.each { println it }
```

When called against a list, the **each** method passes the supplied closure a single parameter (a list item), and this is accessible inside the closure through the **it** variable. However, some functions pass multiple parameters to the supplied closure, and we need to provide a closure with declared parameters. For example, when the **each** method is called against a map, the supplied closure can declare two parameters: one to hold the current map item's key and the other to hold the associated value.

In Code Listing 30: Calling the **each** Method on a Map, the closure has two parameters (**key** & **value**) that help produce an informative display. The parameters can be named in any manner that best describes their use, so the example could have used **day** and **rainfall** as parameter names.

*Code Listing 30: Calling the **each** Method on a Map*

```
def dailyRainfall = [ monday   : 7,
                      tuesday  : 5,
                      wednesday: 19,
                      thursday : 6,
                      friday   : 11 ]

dailyRainfall.each { key, value ->
    println "$key had ${value}mm of rain"
}
```

Note: You can call the `each` method on a map without providing a closure with two parameters. In such a case, the `it` variable will contain the key-value pair.

Summary

That was a whirlwind tour of a specific subset of the Groovy language, but it gives us enough information to produce some very useful code. We've seen how to use basic values (numbers, strings, lists, and maps) as well as declare our own types (classes). Groovy gives us optional typing so that we can take the dynamic approach, declaring variables using the **def** keyword, or by explicitly providing the required type. In fact, we can mix both approaches as we see fit. Closures play a major role in Groovy, and they're first-class citizens in the language, so we can pass them throughout our application as needed.

Now we'll build on this base knowledge by exploring the wealth of existing classes and their functionality. If you want to find out about the rest of the Groovy language, please dip into the Groovy Language Documentation.

Chapter 3 Solution Fundamentals

In Chapter 2 Language Essentials, we covered enough of the key Groovy syntax that you'll be able to read through Groovy code and get an inkling of what it is doing. In this section, we'll build on that understanding and bring in a range of libraries that will help you get the job done.

Libraries

Java provides a wide range of code libraries that Groovy can easily access. The built-in Java libraries are referred to as the Java API (Application Programming Interface), and you'll find them documented in the API Specification. The first thing you'll see is a list of packages such as **`java.lang`** and **`java.time`**. Packages provide a method of grouping similar classes so that they can be easily located. If you find yourself writing larger applications in Groovy, you'll want to start arranging your code into packages, and the Java Tutorial can help you with this.

Note: Code that doesn't declare itself to be in a package (such as our scripts) is allocated to the `default` package.

Groovy modifies some of the Java libraries so as to make them Groovier—this is referred to as the Groovy Development Kit (GDK). The GDK is documented on the Groovy website, and it's worth looking at the GDK documentation prior to checking the Java API's because you'll see where Groovy gives you extra functionality. The Groovy API provides another set of libraries that really let us flex Groovy's versatility.

Tip: Several classes in the org.codehaus.groovy.runtime package add in a number of methods to Java classes. Checking out documentation for classes such as `StringGroovyMethods`, `DefaultGroovyMethods`, and the `typehandling` subpackage can help when you are not certain where a method came from.

Outside of the core Java and Groovy libraries, you'll find a vast array of existing libraries that people have written to overcome various problems. These libraries are offered through open source or commercial licenses and will often spare you from having to write a lot of a codebase yourself. Finding a useful library can be tricky for newcomers, but here are a few sites that will help you get started:

- Maven Central is a key repository for Java libraries and the default for a number of tools.
- MVNRepository provides an alternative interface for searching Maven Central.
- Bintray's JCenter provides a collection of libraries from across a number of repositories, including Maven Central.

These repositories provide a basic model for describing a library using three identifiers:

- **groupId**: designates a project or an organization.

- **artifactId**: designates a specific library from the project/organization.
- **version**: designates the version of the library.

We use a library's coordinates when we want to include the library in our code. The coordinates use the following pattern: *groupId*:*artifactId*:*version*. The coordinates allow us to be specific about what we want to use. For example, we'll look at the Apache Commons CSV library shortly, and the coordinate of **`org.apache.commons:commons-csv:jar:1.2`** specifies version 1.2 of that library.

Once you know the coordinates, using libraries is easy in Groovy scripts, thanks to the Grape dependency manager. Grape provides the **`Grab`** annotation that allows us to include the Apache Commons CSV library in our scripts through a one-liner: **`@Grab('org.apache.commons:commons-csv:1.2')`**. This tells the Groovy compiler to download the requested library and make it available to your code.

Note: Annotations perform a variety of functions within code. The 'at' symbol (@) is used when calling the annotation.

The **`import`** keyword is used to make library classes available to our code. You'll see this used in most of the code listings, and you'll soon get the hang of the usage, but let's look at a few examples:

- **`import java.nio.file.Paths`**: will import the **`Path`** class from the **`java.nio.files`** package.
- **`import java.nio.file.*`**: imports all classes in the **`java.nio.files`** package.
- **`import static org.apache.commons.csv.CSVFormat.RFC4180`**: will import a class field (**`RFC4180`**) from the (**`CSVFormat`**) class and allow us to use it directly.
- **`import static java.nio.file.Paths.get as getFile`**: imports a class method (**`get`**) from the **`Paths`** class. The **`get`** method could be available to call directly in our code, but I've set an alias (**`getFile`**) to use instead (I think that "**`get`**" is a little too generic.)

Groovy imports the following packages/classes by default:

- java.lang.*
- java.util.*
- java.io.*
- java.net.*
- groovy.lang.*
- groovy.util.*

- java.math.BigInteger
- java.math.BigDecimal

There's no harm in explicitly importing those default items but, over time, you'll find that you simply take them for granted.

Next, I'll demonstrate how to use a range of existing Java and Groovy libraries in your code.

Reading and writing data

Data comes to us from a range of sources such as files, web resources, and databases. Groovy can read and write these sources with ease.

The source code for this chapter is located in the *fundamentals* directory of the source code.

Files

Java's **`java.nio.file`** package provides a range of classes for working with files. The **`Path`** class, which represents an object in a file system, is central to this approach. **`Path`** objects help us write code that doesn't get too caught up in how various operating systems manage files, which is essential for a cross-platform system. Groovy extends Java's **`Path`** class so that there's even less to do.

We get an introduction to the sequence of a number of concepts in Code Listing 31: Reading a File:

1. In order to access a **`Path`** object, we call the **`get`** class method provided by the **`Paths`** class:
    a. In order to use the **`Paths`** class, we import it using **`import java.nio.file.Paths`**.
    b. The call to **`Paths.get`** is passed the name of the desired file (**`'demo.txt'`**) as an argument and returns a **`Path`** object that is assigned to the **`file`** variable.
2. The code then demonstrates two approaches to reading the file:
    a. The **`text`** property contains all of the text in the file.
    b. The **`withReader`** method uses a buffer for reading a file's content, which is handy with large files. The call to **`withReader`** is passed a closure that is used to process the file. Once the closure has completed, the file is closed.

Code Listing 31: Reading a File

```
import java.nio.file.Paths

def file = Paths.get('demo.txt')

println 'Reading a file with using the .text attribute:'
print file.text

println 'Reading a file via a Reader:'
file.withReader { reader ->
    print reader.text
}
```

When looking at this code, you might find it helpful to refer to the documentation for the various classes:

- The Java API documentation for the Paths and Path classes.
- Groovy's supplements to the Path class, including the **`withReader`** method.

If you flick through the documentation, you'll notice that **`Path`** has no **`text`** property. However, Groovy enhances **`Path`** with the **`getText`** method, which leads to an interesting "trick" provided by Groovy: a method starting with **`get`** that has no parameters can be read as a property, as we saw in **`file.text`**. The name of the property is determined by dropping the "get" prefix and using lowercase for the first character in the remaining string.

This approach utilizes a common Java convention around getters and setters. Traditionally, the method to read an object's field will use the "get" prefix followed by the name of the field. An associated setter is the provided to change the value of a field and uses both the "set" prefix and a single parameter for the new value.

Tip: When you start writing your own classes, you'll find that Groovy can automatically provide getters and setters, which will save your code from a huge amount of boilerplate.

Code Listing 32: Writing a File demonstrates the use of a setter as a property by writing a haiku to a file through a very easy statement: **`Paths.get('haiku1.tmp').text = haiku`**. This statement gives us access to the *haiku1.tmp* file (even though it doesn't exist), then set its content through the **`text`** property. Almost too easy. The code also demonstrates two other approaches:

- Using the **`withWriter`** method that, much like **`withReader`**, uses buffering to help with larger files and closes the file once the closure has completed.
- Using the `Files` class to create a temporary file to which you can write.

The **`Files`** class provides a number of useful class methods for working with filesystems, including:

- Uses **exists**, which determines if a file exists.
- Checks for filesystem objects:
    - **isDirectory**
    - **isExecutable**
    - **isReadable**
    - **isWritable**
    - **isRegularFile**

While the examples provided in this e-book don't perform checks such as **Files.exists**, you will probably want this in production code.

Code Listing 32: Writing a File

```
import java.nio.file.Files
import java.nio.file.Paths

def haiku = '''\
groovy makes it easy
less boilerplate to create mess
solve the problem quick
'''

Paths.get('haiku1.tmp').text = haiku

Paths.get('haiku2.tmp').withWriter { writer ->
    writer.write(haiku)
}

def tempFile = Files.createTempFile('haiku', '.tmp')
tempFile.write(haiku)
println "Wrote to temp file: $tempFile:"
```

Code Listing 33: Appending a File demonstrates adding to a new or existing file. I've taken a slightly different approach to **Paths.get** by using **import static**, which allows for a specific class member such as the **get** method to be made available directly in the code. By doing this, I could have called **get('pets.tmp')**, but I decided to alias the method by using **as getFile** in the import. I chose this option solely because it makes the method a bit clearer in a larger script.

Once the **getFile** alias has been established, I can add items to a file in one of two ways:

- Through the use of **withWriterAppend**.
- Through the use of the **<<** operator to append in much the same was as appending to a list.

Code Listing 33: Appending a File

```
import static java.nio.file.Paths.get as getFile

def list = ['cat', 'dog', 'rabbit']

getFile('pets.tmp').withWriterAppend {writer ->
    list.each { item ->
        writer.write "$item\n"
    }
}

def list2 = ['fish', 'turtle']
def file = getFile('pets.tmp')
list2.each { pet ->
    file << "$pet\n"
}
```

Web resources

The **toURL** method available on strings makes working with URLs straightforward. Code Listing 34: Reading from a URL demonstrates this by converting a string value (**'http://www.example.com/'**) to a URL, then accessing the contents through the **text** property—just as we read a file's contents. The alternative use of the `withReader` method can be useful for larger web pages.

Code Listing 34: Reading from a URL

```
//The basic (easy) approach
print 'http://www.example.com/'.toURL().text

//Using a reader (handy for larger pages)
'http://www.example.com/'.toURL().withReader { reader ->
    print reader.text
}
```

We should note that the previous code accesses a page available at a website—it doesn't access a web service. If you'd like to access a web service, take a look at groovy-wslite or Jersey.

Databases

It's easy to feel as if there are as many libraries for working with databases as there are database products. Java's JDBC API has been around since JDK version 1.1 was released in 1997. It's a pretty direct approach to working with relational databases, and it generally requires the direct use of SQL. Over the years, a number of libraries have appeared to handle not only relational data but other data structures as well. This work has been annexed under the concept of "persistence" and, for Groovy developers, the various persistence libraries help bridge the gap between Groovy objects and persistence. The following libraries/APIs are worth exploring as your experience grows:

- Grails' Object Relational Mapping (GORM): for relational and nonrelational data.
- Java Persistence API (JPA): for mapping objects to relational data.
- Java Data Objects (JDO): for mapping objects to relational and nonrelational data.

Those technologies are useful in larger applications, but their level of abstraction can be cumbersome in small codebases and scripts. I'll keep it simple and lean towards working with direct SQL queries. Groovy's **`groovy.sql`** package provides enough of a layer over JDBC to make it easy to work with databases without getting quite as involved as working with JDBC.

I'll utilize an SQLite database for the first two examples because providing you with a sample database file (see *data/weather_data.db* in the source code) is easier that way and doesn't require you to set up a server. If you'd like to explore the sample database in a graphical tool, I suggest you grab a copy of the DB Browser for SQLite.

Working with a database requires the use of the appropriate JDBC driver. I've chosen to use the Xerial SQLite driver because it includes all of the components required to interact with an SQLite database. In order to utilize the driver in a Groovy script, we'll need to **`@Grab`** the coordinates **`org.xerial:sqlite-jdbc:3.8.11.2`**. We'll also need to use a configuration setting to tell Grape that it should attach the driver to our script. That last sentence is rather vague because the specifics aren't well-placed for this e-book. Just know that you should always include the following line of code when using JDBC drivers in your scripts: **`@GrabConfig(systemClassLoader = true)`**.

Once you have a JDBC driver loaded, you can connect to the database using a JDBC connection URL. These follow the general format **`jdbc:`*`driverName`*`:`*`databaseName`*`;`*`properties`***. For accessing an SQLite database, we use **`jdbc:sqlite:databaseFile`** where databaseFile is the location of the SQLite database. The example code for this chapter is in *fundamentals/database/,* and the database file is in the relative location of **`../../data/weather_sqlite.db`**, which results in a connection URL of **`jdbc:sqlite:../../data/weather_sqlite.db`**. Given this URL and the **`@GrabConfig`** setting, Groovy will locate the driver and we can start querying a database.

With that configuration overview complete, let's turn to Code Listing 35: Querying a Database and see how it comes together. The first two lines configure Grape and grab a copy of the JDBC driver[2]. Once we have the driver, we import the `groovy.sql.Sql` class and call the **`Sql`** class's static **`withInstance`** method. This method takes two arguments: the JDBC connection URL and a closure that can work with the database connection. Instead of simply using **`it`** within the closure, we declare the **`sql`** parameter to aid in readability.

I display a bit of text within the closure in order to explain what's being queried. The call to **`sql.eachRow`** runs a query (the first argument) and passes each row of the result set to a closure (the second argument). I've written the query within a Groovy String so that I can pass in a variable and rely on the **`Sql`** class's protection against SQL injection. The closure receives one row at a time, and accessing the individual column values is easy—we use the dot-point notation seen in maps (e.g., **`row.recordingDate`**).

Code Listing 35: Querying a Database

```
@GrabConfig(systemClassLoader = true)
@Grab('org.xerial:sqlite-jdbc:3.8.11.2')

import groovy.sql.Sql

Sql.withInstance('jdbc:sqlite:../../data/weather_sqlite.db') { sql ->
    def extremeTemp = 38

    println "Extreme temperatures (over $extremeTemp degrees):"
    sql.eachRow("""
      SELECT strftime('%Y-%m-%d', recordingDate) as recordingDate,
             maxTemp
      FROM WeatherData
      WHERE maxTemp > $extremeTemp""") { row ->
        println "  - ${row.recordingDate}: $row.maxTemp"
    }
}
```

Code Listing 36: Querying a Database (version 2) follows the same pattern as Code Listing 35, but the SQL query calculates the annual average rainfall. We don't need a Groovy String because we don't need to insert a variable, so we simply use the fixed-string form. The average may have any number of decimal places, so we should round the output using **`round(2)`**.

[2] Once Grape has downloaded a library, it will reuse that library. Don't worry about running the script multiple times—you'll only download once.

Code Listing 36: Querying a Database (version 2)

```
@GrabConfig(systemClassLoader = true)
@Grab('org.xerial:sqlite-jdbc:3.8.11.2')

import groovy.sql.Sql

Sql.withInstance('jdbc:sqlite:../../data/weather_sqlite.db') { sql ->
    println 'Annual rainfall averages:'
    sql.eachRow('''
      SELECT strftime('%Y', recordingDate) as year,
             avg(rainfall) as rainfallAverage
      FROM WeatherData
      GROUP BY strftime('%Y', recordingDate)''') { row ->
        println "  $row.year: ${row.rainfallAverage.round(2)}"
    }
}
```

Data formats

Comma-separated Files (CSV)

The comma-separated file is much derided, but it never seems to disappear (most likely due to its alignment with relational databases and its readability). Unfortunately, there's no real CSV standard, and you'll find that systems will differ on the separator they use, the role of quotes, and the use of line breaks. For these examples, we'll use the CSV model described in RFC 4180 and rely on the Apache Commons CSV library because it can work with RFC 4180 format as well as others commonly labeled "CSV."

As you look at Code Listing 37: Reading a CSV File, you might recognize the first few lines:

- I've imported the **`Paths.get`** method with the **`getFile`** alias.
- I use Grape to get a copy of the Apache Commons CSV library. From this, I import the **`RFC4180`** class that predefines the CSV format I want to use.

Next, the code gets a little more complex than the examples we've seen so far. I'll break it down into steps:

1. **`RFC4180.withHeader()`** returns a CSVFormat object that can read an RFC4180-compliant file in which the first row contains the column names.
2. In **`.parse(getFile('../../data/weather_data.csv').newReader())`**:
    a. The call to **`getFile`** loads a **`Path`** to the CSV file, then **`newReader`** opens a reader on the file.

b. The reader is used by the **CSVFormat**'s parse method to correctly parse the rows in the CSV file, which will return a CSVParser object.

3. Once the CSV has been parsed, I call the **iterator()** method, which will let me iterate through the rows using the **each** method.

We've seen the **each** method previously and, in this case, the closure is passed one row from the CSV at a time. As the column headers in a CSV may contain punctuation, Groovy's ability to access map keys using strings comes in handy when accessing the rainfall field: **record.'Rainfall (millimetres)'**.

I've also used the **with** method to make the code a bit more readable. Groovy enhances Java's **Object** class (the basis of all Java classes) by adding the **with** method, which accepts a closure argument. The code within the closure is invoked against the calling object in the first instance. This means we can reduce the verbosity down from **println "$record.Year-$record.Month-$record.Day: $rainfall"**.

Code Listing 37: Reading a CSV File

```
import static java.nio.file.Paths.get as getFile

@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.RFC4180

RFC4180.withHeader()
    .parse(getFile('../../data/weather_data.csv').newReader())
    .iterator().each { record ->
        def rainfall = record.'Rainfall (millimetres)'
        record.with {
            println "$Year-$Month-$Day: $rainfall"
        }
    }
```

This approach of using a series of method calls is referred to as method chaining and relies on each method to return an object on which another method can be called. I encourage you to use this approach because it avoids the need for temporary variables, which makes code more readable. To help you see how method chaining works, I'll use this style of programming for the rest of the e-book.

JavaScript Object Notation (JSON)

The JavaScript Object Notation (JSON) has become ubiquitous in the world of data exchange, configuration files, and in nonrelational data storage. Groovy makes reading JSON files easy through its JsonSlurper class. In Code Listing 38: Reading a JSON File, the contents of a JSON file are read into the script, then handled as we would treat any other Groovy list.

The code reads the JSON file (*weather_data.json*) using **withReader**, much as we've seen previously. Within the closure, an instance of **JsonSlurper** is used to parse the file's contents and place the resulting data structure (a list of objects) into the **weatherData** variable. At this point, we have a variable containing an easily navigable version of the JSON data.

The call to **weatherData.findAll**[3] takes a closure argument that provides a filter for each entry—in this case we want all records for August 2015. All entries in **weatherData** that match this filter are returned in a list, and against this we call the **each** method to display the rainfall details.

Code Listing 38: Reading a JSON File

```
import static java.nio.file.Paths.get as getFile

import groovy.json.JsonSlurper

def jsonSlurper = new JsonSlurper()
def weatherData

getFile('../../data/weather_data.json').withReader { jsonData ->
    weatherData = jsonSlurper.parse(jsonData)
}

weatherData.findAll { it.Year == 2015 && it.Month == '08' }
    .each { record ->
        def rainfall = record.'Rainfall (millimetres)'
        record.with {
            println "$Year-$Month-$Day: $rainfall"
        }
    }
```

Groovy makes it easy to read JSON, but how about creating JSON data? Yup, Groovy has you covered. In fact, it's so easy that I'm going to focus the next example on building on the **findAll** aspect of the previous example.

Code Listing 39: Preparing JSON Output demonstrates an approach to reading a CSV file and producing JSON. Much like we saw in the CVS chapter, the code reads in the CSV file, then gets an iterator to read through the rows. I've used the iterator's **findAll** method to get all entries for January 2010. The subsequent call to the **collect**[4] method is used to return a transformed set of records. In this example, I have used **collect** to create a map for each record that will simplify field access and create a convenient date string for the **day** field. This is a simplistic use of **collect** to remap the data—I could have performed calculations or other, more involved, processes.

Once we have the list of objects, we need only to call **JsonOutput.toJson** to convert the data into JSON. To make this a little more pleasant to read, let's next call **JsonOutput.prettyPrint** to display a nicely formatted version.

[3] See: http://docs.groovy-lang.org/latest/html/groovy-jdk/java/lang/Object.html#findAll(groovy.lang.Closure)

[4] See: http://docs.groovy-lang.org/latest/html/groovy-jdk/java/lang/Object.html#collect%28groovy.lang.Closure%29

Code Listing 39: Preparing JSON Output

```
import static java.nio.file.Paths.get as getFile

@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.RFC4180

import groovy.json.JsonOutput

def data = RFC4180.withHeader()
    .parse(getFile('../../data/weather_data.csv').newReader())
    .iterator()
    .findAll { record ->
        record.Year.toInteger() == 2010 &&
        record.Month.toInteger() == 1
    }.collect { record->
        [day:"$record.Year-$record.Month-$record.Day",
         station: record.Station,
         max: record.'Maximum temparature (celcius)',
         min: record.'Minimum temparature (celcius)',
         rainfall: record.'Rainfall (millimetres)']
    }

print JsonOutput.prettyPrint(JsonOutput.toJson(data))
```

The partial JSON output from the previous example is illustrated in Code Listing 40: JSON Snippet.

Code Listing 40: JSON Snippet

```
[
    {
        "day": "2010-01-01",
        "station": "AU_QLD_098",
        "max": "21",
        "min": "-1",
        "rainfall": "246.0"
    },
    {
        "day": "2010-01-02",
        "station": "AU_QLD_098",
        "max": "32",
        "min": "15",
        "rainfall": "68.0"
    },
```

Extensible Markup Language (XML)

Just as you can slurp up a JSON file, Groovy lets you slurp up XML. Code Listing 41: Reading an XML File demonstrates the use of the `XmlSlurper` class to parse an XML file. Just as with the JSON example, I get a reader for the XML file. Within the reader's closure, I create a new instance of **`XmlSlurper`** on the fly and call the parse method to read the XML data. The **`weatherData`** variable now holds a GPath object that we can use to interrogate the XML. It's handy to know what's in the XML, and I've included a snippet in Code Listing 42: XML Snippet.

Code Listing 41: Reading an XML File

```
import static java.nio.file.Paths.get as getFile

import java.time.Month
import java.time.YearMonth

def weatherData

getFile('../../data/weather_data.xml').withReader { xmlData ->
    weatherData = new XmlSlurper().parse(xmlData)
}

def dateCheck = YearMonth.of(2015, Month.AUGUST)

weatherData.reading.findAll { reading ->
    YearMonth.parse(reading.@day.text()[0..-4]) == dateCheck
}.each { reading ->
    println "${reading.@day.text()}: ${reading.@rainfall.text()}"
}
```

Code Listing 42: XML Snippet

```
<weather>
  <reading day='2006-01-01' station='AU_QLD_098' max='6' min='-4'
           rainfall='286.8' />
  <reading day='2006-01-02' station='AU_QLD_098' max='6' min='-2'
           rainfall='229.5' />
  <reading day='2006-01-03' station='AU_QLD_098' max='14' min='-1'
           rainfall='231.3' />
</weather>
```

Now that we've parsed the XML data, let's get a subset of records, in this case the weather readings for August 2015. In order to set up this filter, we first create a date variable (**`dateCheck`**) using the **`YearMonth`** class from the java.time package (a very useful one to know). This package also includes a **`Month`** enumeration, which we use in order to get a "constant" for **`AUGUST`**.

Once the **dateCheck** has been set up, we call **weatherData.reading.findAll** because **weatherData** holds the root node of the XML file, and we want to **findAll** instances of the **reading** element that relate to August 2015. The closure passed to **findAll** is supplied with each instance of the **reading** element, and the closure performs a check against the reading's date. This looks a little messy at first, but I'll break down the sequence for you:

1. **reading.@day** accesses the day property in the reading element.

2. **reading.@day.text()** returns the value of the property (e.g., **2006-01-01**).

3. **reading.@day.text()[0..-4]** returns a substring of the value, starting at index 0 through to the 4th-last character (e.g., **2006-01**).

4. **YearMonth.parse(reading.@day.text()[0..-4])** creates a type of date object representing a year and month combination.

This would look like **YearMonth.parse('2006-01')** for one of the readings in Code Listing 42: XML Snippet.

When the year-month date object has been determined, we can check it against **dateCheck** to determine if the reading date meets our needs. Once **findAll** has completed, we will have the subset of XML elements that meet our requirement (readings from August 2015), and we can call on the **each** method to display them. Remember that **each** is called against those XML elements, so we must use **@day.text()** and **@rainfall.text()** to access the element's property values.

In the section on Web resources, I demonstrated the use of the **toURL** method to create a URL object from a string and call the **text** property to download a resource. Once downloaded, this is just text (a string) that can be read using **XmlSlurper**'s **parseText** method. Code Listing 43: Reading XML from a URL demonstrates this process and displays the **title** and **updated** element values from the downloaded XML. We can use a similar approach to accessing JSON content from various web resources.

Code Listing 43: Reading XML from a URL

```
def slurper = new XmlSlurper()

def url = 'http://www.iana.org/assignments/media-types/media-
types.xml'.toURL()
def page = slurper.parseText(url.text)
println "${page.title.text()} last updated ${page.updated.text()}"
```

Tip: This works for XHTML pages, but check out http://jsoup.org/ if you want to parse HTML pages.

Now we're into creating XML and Groovy, which gives us the **MarkupBuilder** that makes this particularly easy. Groovy's builders are based on hierarchical object structures such as the ones we see in XML, HTML, JSON, graphical interfaces, etc., and Groovy makes full use of its dynamic nature in order to help the developer build these structures easily.

For XML work, using Groovy's **MarkupBuilder** lets us avoid swaths of method calls to create elements and properties while keeping the logic inside the code (rather than using templates).

Code Listing 44: MarkupBuilder for XML

```
import static java.nio.file.Paths.get as getFile

import groovy.xml.MarkupBuilder

def outputWriter = getFile('weather_data_demo.xml').newWriter('UTF-8')

def weatherData = new MarkupBuilder(outputWriter)

weatherData.rainfall {
    year (value: 1990) {
        reading month: '01', day: '21', 12.7
        reading month: '01', day: '22', 6.3
    }
    year (value: 1995) {
        reading month: '01', day: '21', 0
        reading month: '01', day: '22', 1.8
    }
}

outputWriter.close()
```

Code Listing 44: MarkupBuilder for XML contains this sequence of familiar code:

1. We get a writer for a new XML file with a minor addition—we want to make sure that the file is UTF-8 encoded.

2. We then create a new **MarkupBuilder** object and store it in the **weatherData** variable.

3. Calling **weatherData.rainfall**, we next create the structure we need.

4. To complete the script, we close the writer.

The output of the script is provided in Code Listing 45: MarkupBuilder for XML—Output, and it all looks pretty good to me. However, take another look at item 3 in the explanatory list above. It looks wrong. After all, how did the **MarkupBuilder** class know about our weather data use-case? Well, it didn't. But Groovy has meta-programming mechanisms that give developers a variety of facilities so that they can react to situations on the fly. As users of **MarkupBuilder**, we can feed it structures that meet our needs. **MarkupBuilder** will respond this way:

1. **weatherData.rainfall** starts off our XML structure and sets the root node to **<rainfall>**, so that everything within the closure argument is a child of that node.

2. The next node is a **year** element with a single property (**value**)—the associated closure sets the element's children.

3. Within a **year** element is a set of **reading** elements:

a. The **`reading`** has a number of properties that are set using map notation of *key*: *value*.

b. The content of each **`reading`** element is a rainfall measurement.

This makes for a very readable approach to creating the structure we need to represent in XML. For an XML node, we provide a name and (optionally) a set of map items for the element's properties. We can provide a value for the node once the properties have been provided. A closure is then started to add child elements.

Code Listing 45: MarkupBuilder for XML—Output

```
<rainfall>
  <year value='1990'>
    <reading month='01' day='21'>12.7</reading>
    <reading month='01' day='22'>6.3</reading>
  </year>
  <year value='1995'>
    <reading month='01' day='21'>0</reading>
    <reading month='01' day='22'>1.8</reading>
  </year>
</rainfall>
```

Because HTML is a close neighbor to XML, we can use MarkupBuilder to create (X)HTML. Code Listing 46: MarkupBuilder for HTML builds on the previous XML example to create an HTML file followed by the output provided in Code Listing 47: MarkupBuilder for HTML—Output.

The XML and HTML examples both use a rather static set of data, and we'll look into building the markup in a more dynamic manner shortly. As a brief aside, let me mention a number of items that might be worth further investigation, depending on your needs:

- If you want to generate HTML (or any other text format) using a template, Groovy's template engines are definitely worth a look. They're quite versatile, and they save using a third-party engine such as Velocity.
- Groovy provides a StreamingMarkupBuilder class that's useful for incrementally building up a structure.
- If you need to manipulate XML in place, check out the Groovy documentation.

Code Listing 46: MarkupBuilder for HTML

```
import static java.nio.file.Paths.get as getFile
import groovy.xml.MarkupBuilder

getFile('weather_data_demo.html').withWriter('UTF-8') {
    new MarkupBuilder(outputWriter).html {
        head {
            title 'Rainfall readings'
        }
        body {
            h1 'A selection of rainfall readings'
            h2 'Year: 1990'
            table {
                tr {
                    th 'Month'
                    th 'Day'
                    th 'Reading'
                }
                tr {
                    td '01'
                    td '21'
                    td '12.7'
                }
                tr {
                    td '01'
                    td '22'
                    td '6.3'
                }
            }
        }
    }
}
```

Tip: There's also a builder for JSON—the [JsonBuilder.]

Code Listing 47: MarkupBuilder for HTML—Output

```
<html>
  <head>
    <title>Rainfall readings</title>
  </head>
  <body>
    <h1>A selection of rainfall readings</h1>
    <h2>Year: 1990</h2>
    <table>
      <tr>
        <th>Month</th>
        <th>Day</th>
        <th>Reading</th>
      </tr>
      <tr>
        <td>01</td>
        <td>21</td>
        <td>12.7</td>
      </tr>
      <tr>
        <td>01</td>
        <td>22</td>
        <td>6.3</td>
      </tr>
    </table>
  </body>
</html>
```

Let's turn our MarkupBuilder skills toward the weather data. You'll see in Code Listing 49: Preparing XML Output that I've read in the data from the CSV and filtered the January 2010 readings. Calling **`new MarkupBuilder().weather`**, I'm preparing to convert those CSV-based records into XML (with **`<weather>`** as the root node). Yet again, I call on the **`each`** method to iterate through the data in order to create one **`<reading>`** element per CSV row. The format of the code may look a little odd at first, but remember that Groovy methods don't need parentheses, which means the body of the **`each`** closure is a single method call that the **`MarkupBuilder`** turns into an HTML element with properties. The top few elements can be seen in Code Listing 48: Sample Weather Output.

Code Listing 48: Sample Weather Output

```
<weather>
  <reading day='2010-01-01' station='AU_QLD_098' max='19' min='1'
rainfall='196.0' />
  <reading day='2010-01-02' station='AU_QLD_098' max='21' min='9'
rainfall='300.8' />
```

Code Listing 49: Preparing XML Output

```
import static java.nio.file.Paths.get as getFile

@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.RFC4180

import groovy.xml.MarkupBuilder

def data = RFC4180.withHeader()
    .parse(getFile('../../data/weather_data.csv').newReader())
    .iterator()
    .findAll { record ->
        record.Year.toInteger() == 2010 &&
        record.Month.toInteger() == 1
    }

new MarkupBuilder().weather {
    data.each { record ->
        reading day:"$record.Year-$record.Month-$record.Day",
                station: record.Station,
                max: record.'Maximum temparature (celcius)',
                min: record.'Minimum temparature (celcius)',
                rainfall: record.'Rainfall (millimetres)'
    }
}
```

A CSV-to-database example

Having worked through loading data from a CSV file and interacting with a database, we're now in a position to go a little deeper into a range of database tasks. In Code Listing 50: A Complete Database Usage Example, we'll construct a database from scratch, load it with CSV data, then run some queries. Instead of revisiting SQLite, we'll use an in-memory Apache Derby database.

Code Listing 50: A Complete Database Usage Example is a large script but can be broken down into the following sequential segments:

1. Load the required libraries, including the JDBC driver for Apache Derby (**`'org.apache.derby:derby:10.12.1.1'`**).

2. Parse the CSV file with **`RFC4180.withHeader().parse(getFile('../../data/weather_data.csv').newReader())`**.

3. Setup an SQL instance: **`Sql.newInstance('jdbc:derby:memory:myDB;create=true')`**. Note the JDBC connection string includes a flag (**`create=true`**) that allows us to create the database if one doesn't exist. This may not be crucial for an in-memory database, but it would be important if we are using a persistent Derby database.

4. In order to create the required **WeatherData** table, we issue an **sql.execute** call with the appropriate data definition language (DDL) statement (**CREATE TABLE**).

5. The insertion of the CSV-based records into the **WeatherData** table is performed within a transaction using the **withTransaction** method. This is passed a closure that, when successfully completed, causes the transaction to be committed. Within the transaction, we make a series of parameterized INSERTs by calling **sql.executeInsert**.

6. Next, the first query is run in order to provide a summary of the data. We use **sql.firstRow** because we expect only one row to be returned.

7. The second query is for the rainfall readings for February 2012. Because this will return multiple rows, we use **sql.eachRow**, then pass it the query and a closure that will handle each row in the result set.

8. Just to be tidy, let's then close the connection to the database with the call **sql.close()**.

You might have noticed that I've used the **Date** class from the **java.sql** package when formatting the data for the insert operation. For each insertion, I call **valueOf("$Year-$Month-$Day")** class method from **Date** so as to bridge how Groovy deals with dates and how SQL deals with them.

Code Listing 50: A Complete Database Usage Example

```
import static java.nio.file.Paths.get as getFile
import static java.sql.Date.valueOf
@GrabConfig(systemClassLoader = true)
@Grab('org.apache.derby:derby:10.12.1.1')

@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.RFC4180

import groovy.sql.Sql

//Read in the CSV
def parser = RFC4180.withHeader()
        .parse(getFile('../../data/weather_data.csv').newReader())

//Get a connection to the database
def sql = Sql.newInstance('jdbc:derby:memory:myDB;create=true')

sql.execute '''
    CREATE TABLE WeatherData (
        id INT NOT NUll GENERATED ALWAYS AS IDENTITY,
        station CHAR(10) NOT NULL,
        date DATE NOT NULL,
        maxTemp SMALLINT,
        minTemp SMALLINT,
        rainfall DECIMAL
    )
'''
```

```
//Insert the CSV records into the WeatherData table
def insertCounts = 0
sql.withTransaction {
    def insertSql = '''
        INSERT INTO WeatherData(station, date, maxTemp,
                                minTemp, rainfall)
        VALUES (?,?,?,?,?)'''


    parser.each { record ->
        record.with {
            sql.executeInsert(insertSql,
                    [ Station,
                      valueOf("$Year-$Month-$Day"),
                      record."Maximum temparature (celcius)",
                      record."Minimum temparature (celcius)",
                      record."Rainfall (millimetres)"
                    ])
        }
        insertCounts++
    }

}

println "Rows created: $insertCounts"

//Query some summary data
sql.firstRow('''
  SELECT min(date) as startDate,
         max(date) as endDate,
         max(maxTemp) as highestTemp,
         min(minTemp) as lowestTemp,
         avg(maxTemp) as averageMaxTemp,
         avg(minTemp) as averageMinTemp
  FROM WeatherData''').with {
    println """Temperature extremes for $startDate to $endDate
    Highest temperature: $highestTemp (average: $averageMaxTemp)
    Lowest temperature: $lowestTemp (average: $averageMinTemp)"""
}

//Query the rainfall for a specific month.
println '\nRainfall for February 2012:'
sql.eachRow("""
  SELECT date, rainfall
  FROM WeatherData
  WHERE YEAR(date) = 2012 AND MONTH(date) = 2""") { row ->
    println "${row.date.toLocalDate().dayOfMonth}: $row.rainfall"
}

//Last thing is to close the database connection
sql.close()
```

Summary

This chapter has built on the basic language overview by demonstrating some very common programming processes. On nearly a daily basis, we are called on to read and write data from various sources and in various formats. I've presented the main formats you'll see (CSV, XML, JSON and databases) and also how far you can go by using Groovy's built-in feature set and existing libraries. This makes Groovy a very strong candidate for preparing nontrivial scripts for running on servers and desktops.

Chapter 4 Data Streams

Introducing streams

In Chapter 3, we used the **`findAll`** and **`collect`** methods to filter and transform data. These methods are handy, but keep in mind that they return new data structures (such as a list or a map) from each call. As we chain them through a series of methods (i.e. a pipeline) each returned value uses an amount of memory that (eventually) gets wiped while the next method grabs another chunk of memory. It's all a bit "chunky."

When Java 8 was released, it included two new features—lambdas and streams. Groovy developers already had closures, so lambdas weren't particularly exciting. In fact, Groovy closures can simply be dropped into spots where lambdas are expected and the closure syntax is nicer (to my eye). At first glance, streams look to perform the same role as many of the methods that Groovy added to lists and maps. However, some very useful differences make streams worth investigating.

Tip: I tend to use the built-in Groovy methods such as `each` and `findAll` for smaller data or simple operations, but I use streams when it gets more complex.

Let's start by looking at how we work with streams. First of all, a source, which can be finite or infinite, is needed. For the examples we have been looking at, the source would be a finite list of items, but, because we're looking at weather recordings, the data could just as well be infinite. To keep things simple, we'll read in the finite amount of data from the previously utilized CSV file of weather data.

Streams will let us analyze data in a manner looks a lot like SQL. We'll be able to filter the data, perform aggregations (e.g., sums and averages), map the data to different structures, and group and sort. This is all performed by using a sequence of method calls that adhere to the following process:

1. Create the stream.
2. Perform zero or more intermediate operations.
3. Perform one terminal operation.

The elements in this process are referred to as a pipeline, and it's easy to see why. Picture the data starting in a pool or lake. The data then flows through a number of operations and terminates in a final output. I imagine something like the Nile or the Amazon starting from a source, flowing into different rivulets, and finally arriving at a river mouth. (I'm not overly poetic, so find an analogy of streams and pipes that best suits you!)

Our weather data is held in a list after being read from the CSV, which means creation of a stream is easy—you simply call the **`stream()`** method on the list.

Intermediate operations, such as **`filter`** and **`map`**, return a stream and allow for further intermediate operations. Think of these operations as applying a lens over the original data—subsequent operations will see the data through this lens. However, these operations are different from the built-in Groovy methods such as **`findAll`** because the intermediate operation doesn't return a new data value (e.g., a list of maps).

Intermediate operations are said to be lazy because they aren't actually processed until the stream reaches a terminal operation. Terminal operations mark the end of a stream and are generally used to reduce the data through some sort of calculation or to collect the data in some manner. Once a terminal operation has been called, you'll be unable to access that stream again, and you will need to create a new strream.

The original data source (e.g., the list) is not changed by the stream, and this is a desirable characteristic. While some of the stream methods open up the ability to modify the underlying data, doing this is not recommended—it will reduce your ability to achieve other efficiencies.

As we begin exploring streams, I'll describe the examples that follow by posing a question for the data, describing a method for answering it, and providing the output. I'll build up the use of streams over these examples so we can focus on a specific operation at each stage.

The first question is "Which records exist for February 2008?" Code Listing 51: Filtering a Stream treads some familiar ground in reading a CSV file, so the second half of the code listing is what interests us. In order to create a stream, we simply call **`weatherData.stream()`**, then we can provide the rest of the pipeline as a chain of methods calls.

In order to extract the required records, let’s use the **`filter`** method twice—once for the year and again for the month. The filtering can be performed in a single call, but I find that breaking each filter item into its own call is more readable for a set of filters that are performing a boolean AND (where all elements must be true). I suggest using a single filter call for performing boolean OR. The **`filter`** method is passed a closure that defines the predicate that an entry must match to meet the filter. This is much the same as we saw in the **`findAll`** method.

The terminal operation for the stream pipeline is **`collect()`**. This call causes the **`filter`** methods to be processed and the resulting subset to be coalesced. The **`mappedData`** variable then holds a list of (**`CSVRecord`**) items that match the filter predicates. If **`collect`** wasn't called, the **`mappedData`** variable would hold a stream that would be waiting for its terminal operation.

Code Listing 52: Partial Output from the Filtered Stream displays the first few JSON-formatted list items from the **`mappedData`** variable. These are generated as a list of lists by the **`JsonOutput`** library, but they also could have been easily handled in the approaches seen in the Chapter 3 Solution Fundamentals section. I'll improve on the output in the next example.

Code Listing 51: Filtering a Stream

```
import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static java.nio.file.Paths.get as getFile

@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*

def weatherData = RFC4180.withHeader()
    .parse(getFile('../data/weather_data.csv').newReader())
    .getRecords()

def mappedData = weatherData.stream()
    .filter { it.Year == '2008' }
    .filter { it.Month == '02' }
    .collect()

print prettyPrint(toJson(mappedData))
```

Code Listing 52: Partial Output from the Filtered Stream

```
[
    [
        "AU_QLD_098",
        "2008",
        "02",
        "01",
        "20",
        "5",
        "345.4"
    ],
    [
        "AU_QLD_098",
        "2008",
        "02",
        "02",
        "18",
        "8",
        "206.0"
    ],
```

The next question is "What was the daily rainfall for February 2008?" As you can see in Code Listing 53: Mapping a Stream, I create the stream and perform the filtering in the same manner as earlier listings. I then call the **map** intermediate operation and pass it a closure to perform the mapping. This closure is called for each item in the stream that meets the filter predicates. Within the closure I set up a map with two keys:

- **`date`** holds a formatted string using the ISO 8601 format of YYYY-MM-DD.
- **`rainfall`** holds the value held in the 'Rainfall (millimeters)' field in the CSV, cast as a **`BigDecimal`**.

The mapping simply prepares a data structure that suits my need, nothing particularly complex. Another mapping could perform calculations to transform the data if needed, perhaps changing millimeters to inches in order to meet another system's requirements.

The call to **`collect`** results in **`mappedData`** holding a list of the maps created by the **`map`** operation. This is converted into JSON, as seen in Code Listing 54: Partial Output from the Mapped Stream, and it is more useful that the previous JSON output.

Code Listing 53: Mapping a Stream

```
import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static java.nio.file.Paths.get as getFile

@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*

def weatherData = RFC4180.withHeader()
    .parse(getFile('../data/weather_data.csv').newReader())
    .getRecords()

def mappedData = weatherData.stream()
    .filter { it.Year == '2008' }
    .filter { it.Month == '02' }
    .map {
        [date: "${it.Year}-${it.Month}-${it.Day}",
         rainfall: it.'Rainfall (millimetres)'.toBigDecimal()]
    }.collect()

print prettyPrint(toJson(mappedData))
```

Code Listing 54: Partial Output from the Mapped Stream

```
[
    {
        "date": "2008-02-01",
        "rainfall": 345.4
    },
    {
        "date": "2008-02-02",
        "rainfall": 206.0
    },
```

Carrying over from the previous example, we can use the collect method to perform a mapping operation as the terminating operation. Taking a look at Code Listing 53: Mapping a Stream, you'll see that I've taken out the call to the **map** operation and called the **collect** method with an argument to the Collectors.toMap static method. Remember that Groovy allows us to skip use of parentheses when calling a method, and I use this option with the **collect** call's arguments but use parentheses for the **toMap** call. This groups the arguments cleanly and avoids excess syntax.

The **toMap** method accumulates each stream item into a map item. The call is passed two closures—the first generates the key for the map; the second returns the value for the map. As you'll see in Code Listing 56: Partial Output from the Collected Stream, we now have a very compact answer to our question.

Code Listing 55: Collecting a Stream

```
import static java.nio.file.Paths.get as getFile
@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*

import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static java.util.stream.Collectors.toMap

def weatherData = RFC4180.withHeader()
        .parse(getFile('../data/weather_data.csv').newReader())
        .getRecords()

def mappedData = weatherData.stream()
        .filter { it.Year == '2008' }
        .filter { it.Month == '02' }
        .collect toMap({"${it.Year}-${it.Month}-${it.Day}"},
                       { it.'Rainfall (millimetres)'.toBigDecimal() })

print prettyPrint(toJson(mappedData))
```

Code Listing 56: Partial Output from the Collected Stream

```
{
    "2008-02-21": 142.1,
    "2008-02-22": 53.5,
    "2008-02-23": 29.2,
    "2008-02-01": 345.4,
    "2008-02-24": 327.5,
```

Reducing

So far I have filtered a stream and mapped the data in a basic manner. I haven't performed any calculations to summarize the data, and this is where reduction comes in. You may have heard of the MapReduce model before—essentially, it consists of a process through which data is filtered, sorted, and transformed (the Map part), then summarized (the Reduce part). A summarizing operation might determine the average (mean), maximum, minimum, sum, or any other calculation that is performed across the data to return a value of interest.

Streams provide a variety of terminal operations that perform the Reduce component of MapReduce. What's more, a number of handy methods are provided by streams that save us from writing our own Reduce functions.

In addition to specific methods (such as **`average()`**), the **`collect`** terminal operation can be passed a `Collector` that performs some sort of reduction. In the previous section, I used the **`toMap`** method provided in the `Collectors` class, and this is a great place to check before writing your own **`Collector`**.

Let's prepare another question to answer: "What was the average rainfall for February 2008?"

Code Listing 58: Calculating an Average creates the stream then filters it in the same manner we saw in the previous examples. Once this has been laid out, we next call **`mapToDouble`** because this method lets us extract the values for the **`'Rainfall (millimetres)'`** field (converted to a **`Double`**). We do this as the **`mapToDouble`** method returns a specialized stream (`DoubleStream`) that provides us with an easy way to determine the average of the filtered items.

So, while the original stream was for the list of records from the CSV, we use **`mapToDouble`** to slice out a stream based on a specified field/column. We can then use this stream to perform an aggregate operation—in this case it's the **`average()`** calculation. At this point, it would be reasonable to think we'd have a single value, the average rainfall for February 2008. However, we actually have an **`OptionalDouble`**, and this needs some explaining.

In programming, the concept of `null` represents the absence of a value—this means that **`null`** itself isn't a value. Null causes a lot of frustration for developers because it either creates exceptions or has to be handled so as to avoid these exceptions. In the Java and Groovy world, an attempt to call a variable that is **`null`** (i.e. it holds no value) causes a **`NullPointerException`** and no small amount of annoyance.

In Code Listing 57: Damn You Null!, I have written some pretty stupid code that sets up a new map variable, assigns it to null, then tries to access an element that is no longer there. This is something you should never see in the wild, but imagine that it's a much larger application in which my **`details`** variable is assigned **`null`** by some other part of the code. Perhaps this assignment was an act of tidying up that I'm unaware of. Regardless, my code now throws a NullPointerException and comes to a screaming halt.

Code Listing 57: Damn You Null!

```
def details = [name: 'Fred']

details = null

println details.name
```

Groovy lets me handle this more elegantly through the use of the safe navigation operator (**?.**), and I could have used **println details?.name** to avoid the exception. Groovy also interprets null to be false, which means I can use the ternary operator: **println details ? details.name :'Null'**.

The Java 8 release included the Optional class and its siblings, including DoubleOptional. Optional objects may or may not contain a value. This is a handy abstraction because it provides a wrapping that offers another approach to avoiding **NullPointerException**s. We can check that an optional contains a value by calling its **isPresent** method. However, we can simply use the optional's **orElse** method and provide a default value as an argument.

In Code Listing 58: Calculating an Average, the call average() returns a **DoubleOptional,** so we then call its **orElse** method to access the value. If the call to average happens to result in **null**, the call to **orElse** will result in zero (0) being returned and a **NullPointerException** avoided.

Code Listing 58: Calculating an Average

```
import static java.nio.file.Paths.get as getFile
@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*

def weatherData = RFC4180.withHeader()
    .parse(getFile('../data/weather_data.csv').newReader())
    .getRecords()

println weatherData.stream()
    .filter { it.Year == '2008' }
    .filter { it.Month == '02' }
    .mapToDouble { record ->
        record.'Rainfall (millimetres)'.toDouble()
    }
    .average()
    .orElse(0)
```

If I was to ask the question "What was the total rainfall for February 2008?" you'd be excused for thinking that you could simply swap out the call to **average()** with one to **sum()**, but, annoyingly, this isn't quite right. Code Listing 59: Calculating a Sum, illustrates that the **sum()** method doesn't return an optional but instead gives us a direct value. It's always useful to check with the API documentation to see if you'll get a value or an optional returned.

Code Listing 59: Calculating a Sum

```
import static java.nio.file.Paths.get as getFile
@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*

def weatherData = RFC4180.withHeader()
    .parse(getFile('../data/weather_data.csv').newReader())
    .getRecords()

println weatherData.stream()
    .filter { it.Year == '2008' }
    .filter { it.Month == '02' }
    .mapToDouble { record ->
        record.'Rainfall (millimetres)'.toDouble()
    }
    .sum()
```

Tip: These examples are all based on "clean" data and assume that numeric fields in the CSV contain numbers. If your data is messy, it's worth using the `map` operation to tidy it up.

While we can use operations such as **`sum`**, **`average`**, **`max`**, **`min`**, and **`count`** to summarize a stream, we can also perform reduce operations in the **`collect`** call. This will let us answer the question "What were the rainfall stats for February 2008?"

Code Listing 60: Summarizing Data demonstrates just how easy streams make it to prepare summary statistics for a set of values. In this case, I have summarized the rainfall readings within the **`collect`** operation. In order to perform the aggregation, we pass the **`collect`** method a call the to `Collectors.summarizingDouble`. This latter method accepts a closure that returns a Double value for each item in the stream. In this case, we simply cast the rainfall reading to a Double value. The resulting output of the stream is then displayed in JSON format in Code Listing 61: Output of Summarizing Data.

Code Listing 60: Summarizing Data

```
import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static java.nio.file.Paths.get as getFile
import static java.util.stream.Collectors.summarizingDouble

@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*

def weatherData = RFC4180.withHeader()
        .parse(getFile('../data/weather_data.csv').newReader())
        .getRecords()

def summaryData = weatherData.stream()
    .filter { it.Year == '2008' }
    .filter { it.Month == '02' }
    .collect summarizingDouble{it.'Rainfall (millimetres)'.toDouble()}

print prettyPrint(toJson(summaryData))
```

Code Listing 61: Output of Summarizing Data

```
{
    "sum": 6283.5,
    "average": 216.67241379310346,
    "max": 380.6,
    "min": 29.2,
    "count": 29
}
```

Grouping

Breaking down the data into groups, especially for summary/aggregate operations, is often useful. We can achieve this in the **`collect`** operation through calls to the [`Collectors.groupingBy`] methods.

Let's try out grouping by posing the question "What were the monthly rainfall statistics for 2008?" The answer will look similar to Code Listing 60: Summarizing Data, but that only gave us a summary for a single month. Essentially, that solution summarized the filtered data, and the filter applied to a specific month and year combination. In order to solve this question, we'll have to filter for just the year, then calculate the summaries for each month. Luckily, we can do this with a stream.

Code Listing 63: Summarizing by Month sets up the stream and filters for items from the year 2008. We then use the **`map`** operation to set up a map for each item. This map contains two keys:

- **`date`**—constructed using a helpful closure (**`getDate`**) I set up.
- **`Rainfall`**—uses a BigDecimal representation of the reading.

We now have a "data set" for 2008 represented as a series of date and rainfall fields. In order to answer the question, we'll need to group this data by month and, for each month, we must prepare the summary data. This is performed in the **`collect`** operation by calling the form of **`Collectors.groupingBy`**, which takes two arguments:

- The classifier for the group provided as a closure. This is straightforward—we just grab the month component from the date field.
- A collector that is used to perform a reduction. For this, we run **`summarizingDouble`** over the rainfall data.

Code Listing 62: Partial Output for Summary by Month provides a portion of the JSON displayed by Code Listing 63: Summarizing by Month. The JSON is a series of map items, each one representing a month from 2008 and containing a map of summary data.

Code Listing 62: Partial Output for Summary by Month

```
{
    "APRIL": {
        "sum": 6389.8,
        "average": 212.99333333333334,
        "max": 393.7,
        "min": 5.3,
        "count": 30
    },
    "SEPTEMBER": {
        "sum": 6498.5,
        "average": 216.61666666666667,
        "max": 395.2,
        "min": 11.1,
        "count": 30
    },
```

Code Listing 63: Summary by Month

```
import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static java.nio.file.Paths.get as getFile
import static java.util.stream.Collectors.groupingBy
import static java.util.stream.Collectors.summarizingDouble

@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*

import java.time.LocalDate

def weatherData = RFC4180.withHeader()
        .parse(getFile('../data/weather_data.csv').newReader())
        .getRecords()

def getDate = { y, m, d ->
    LocalDate.of(y.toInteger(),
            m.toInteger(),
            d.toInteger())
}

def monthlyData = weatherData.stream()
    .filter { it.Year == '2008' }
    .map {
        [date: getDate(it.Year, it.Month, it.Day),
         rainfall: it.'Rainfall (millimetres)'.toBigDecimal()]
    }.collect groupingBy({ it.date.month },
                         summarizingDouble{it.rainfall.doubleValue()})

print prettyPrint(toJson(monthlyData))
```

In Code Listing 63: Summarizing by Month, I prepared a **LocalDate** object for the date field, but this wasn't strictly needed because I could have used the two-digit month value for the date. I went that extra bit further so that my resulting data (and JSON) would have a real date value that could be more easily utilized by other code.

The **groupingBy** method allows us to create subgroups in a cascading fashion. This allows us to group the data to the level we need, then perform aggregation.

Subgroups will help us answer the question "What was the daily rainfall record as grouped by year then month?" This isn't asking us for a calculation, just a data structure in which the daily rainfall recordings are grouped by month, which are themselves grouped by year.

Code Listing 64: Grouping by Year then Month is somewhat like our previous example in that the **groupingBy** method is called within the **collect** operation. The first call to **groupingBy** is based on the year (the first argument), then it calls another **groupingBy** (the second argument). This second **groupingBy** works within the context of the grouped year (e.g., 2006), so that there's no crossover from data belonging to the same month in another year. It's at the second grouping (month) that we then call **toMap** to create a map in a manner similar to the one seen in Code Listing 55: Collecting a Stream.

Code Listing 64: Grouping by Year then Month

```
import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static java.nio.file.Paths.get as getFile
import static java.util.stream.Collectors.groupingBy
import static java.util.stream.Collectors.toMap

@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*

import java.time.LocalDate

def weatherData = RFC4180.withHeader()
        .parse(getFile('../data/weather_data.csv').newReader())
        .getRecords()

def getDate = { y, m, d ->
    LocalDate.of(y.toInteger(),
            m.toInteger(),
            d.toInteger())
}

def monthlyData = weatherData.stream()
    .map {
        [date: getDate(it.Year, it.Month, it.Day),
         rainfall: it.'Rainfall (millimetres)'.toBigDecimal()]
    }.collect groupingBy({ it.date.year },
                groupingBy({ it.date.month },
                    toMap({ it.date.day },{ it.rainfall })))

print prettyPrint(toJson(monthlyData))
```

As you can see in Code Listing 65: Partial Output of Grouping by Year then Month, the resulting JSON is a series of year objects containing a series of month objects, each containing a map of rainfall readings for each day in the month.

Code Listing 65: Partial Output of Grouping by Year then Month

```
{
    "2006": {
        "DECEMBER": {
            "1": 237.2,
            "2": 188.2,
            "3": 359.6,
            "4": 231.1,
            "5": 93.2,
            "6": 337.0,
            "7": 234.2,
            "8": 246.8,
            "9": 184.0,
            "10": 332.6,
```

The intent of our previous question was to restructure the data held in the CSV file, but it's more likely we'd ask "What was the average rainfall by month, for each year?" The solution code is similar to the previous code, but we call on **averagingDouble** instead of **toMap** within the subgroup.

Code Listing 66: Average by Year-Month

```
import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static java.nio.file.Paths.get as getFile
import static java.util.stream.Collectors.averagingDouble
import static java.util.stream.Collectors.groupingBy

@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*

def weatherData = RFC4180.withHeader()
        .parse(getFile('../data/weather_data.csv').newReader())
        .getRecords()

def monthlyAverages = weatherData.stream()
    .collect groupingBy({ "$it.Year" },
                groupingBy({ "$it.Month" },
                    averagingDouble {
                        it.'Rainfall (millimetres)'.toDouble()
                    }))

print prettyPrint(toJson(monthlyAverages))
```

In Code Listing 66: Average by Year-Month, you'll see that I skipped my conversion to a **LocalDate** and instead used the raw **Year** and **Month** values from the CSV. I'd prefer to use the **LocalDate** approach, but I wanted to demonstrate an alternative. Within the inner grouping (by month), I call the **averagingDouble** method from the **Collectors** class. This accepts a closure that I use to provide a double value for each record, then calculates the average across the values in the group.

This will give me a useful output structure of year objects, each containing their constituent months and the average rainfall for each month. You can see the first part of the output in Code Listing 67: Partial Output for Average by Year-Month.

Code Listing 67: Partial Output for Average by Year-Month

```
{
    "2015": {
        "10": 198.66774193548386,
        "11": 183.61666666666667,
        "12": 240.41935483870967,
        "01": 161.56129032258065,
```

Sorting

While reading the previous code listing, you might have noticed that the months weren't in order. This may not be an issue if the recipient code doesn't need the data to be sorted. Furthermore, by not worrying about ordering, the stream operations have greater flexibility with processing. Different chunks of the pipeline can be sent to other threads or even to processors, and it won't matter when they're returned. You often may not care about sorting/order, but this isn't the case with everyone.

The **`sorted`** method in the **`Stream`** class provides an intermediate operation that, you guessed it, returns a stream consisting of sorted elements. The first variant of **`sorted`** uses the natural order of the stream elements. This is great if you have a numeric stream such as the **`DoubleStream`** we saw earlier. The second variant lets us provide our own comparator, which will be handy for our next question: "Could I have the daily rainfall figures for February 2008 presented in descending order?"

Code Listing 68: Sorting a Stream filters for the appropriate records, then calls on **`sorted`**. If I had only a stream of the rainfall recordings, I could have simply relied on **`sorted`** without a comparator, but I would lose the day on which the recording was made, and I want this in my answer. This means that I must provide my own comparator and that this is a closure that accepts two arguments: the first and second objects for comparison. As you can see, the closure I've provided calls another closure (**`sortDescending`**), which uses the spaceship operator (mentioned in Operators) to provide the comparison.

Instead of terminating the stream with a **`collect`** operation, I used **`forEach`** to display the results. Having seen Groovy's **`each`** method, you won't be overly surprised with the sample output seen in Code Listing 69: Partial Output for Sorting a Stream.

Code Listing 68: Sorting a Stream

```
import static java.nio.file.Paths.get as getFile
@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*

def weatherData = RFC4180.withHeader()
        .parse(getFile('../data/weather_data.csv').newReader())
        .getRecords()

def sortDescending = { n1, n2 ->
    n1.toBigDecimal() <=> n2.toBigDecimal()
}

weatherData.stream()
    .filter { it.Year == '2008' }
    .filter { it.Month == '02' }
    .sorted { day1, day2 ->
        sortDescending day2.'Rainfall (millimetres)',
                       day1.'Rainfall (millimetres)' }
    .forEach {
        println "$it.Day: ${it.'Rainfall (millimetres)'}"
    }
```

Naturally, we could use **forEach** to perform more serious actions—perhaps shooting the data out to a web service.

Code Listing 69: Partial Output for Sorting a Stream

```
14: 380.6
13: 373.0
06: 371.5
08: 370.8
07: 362.5
01: 345.4
24: 327.5
```

Conclusion

In this chapter we've filtered, group sorted, and reduced the weather data. Those of you with an SQL background are probably seeing aspects of the **select**, **where**, and **order by** statements with aspects of aggregate functions. I tend to see streams from this perspective—they give me a domain-specific language for working with a data series.

Although I must load the data from a file, I use an existing library (Apache Commons CSV) to do the heavy lifting. This is similar to SQL's **from** statement, but the CSV library is less abstract, so I still need to do a little prep. Because Java streams continue to be folded into general use, we may see libraries further simplify this aspect. For now, though, it's easy enough.

In this chapter's examples, I have performed the CSV read as one unit of work and the stream operations as another. This means that I have needed an interim variable (such as **`weatherData`** or **`monthlyData`**) to hold the parsed CSV data. My goal was to improve readability and comprehension for learning, but the whole process can be performed in a single chain of method calls, as seen in Code Listing 70: A Single-Method Chain for a CSV File and a Stream. Ultimately, you'll make a call on what's more readable, but the chain really captures the process of reading in data, filtering, mapping, and reducing (summarizing).

Code Listing 70: A Single-Method Chain for a CSV File and a Stream

```
import static java.nio.file.Paths.get as getFile
@Grab('org.apache.commons:commons-csv:1.2')
import static org.apache.commons.csv.CSVFormat.*
import static java.util.stream.Collectors.summarizingDouble

def displayJson = { data ->
    print prettyPrint(toJson(data))
}

displayJson RFC4180.withHeader()
        .parse(getFile('../data/weather_data.csv').newReader())
        .getRecords()
        .stream()
        .filter { it.Year == '2008' }
        .filter { it.Month == '02' }
        .collect(summarizingDouble { it.'Rainfall
(millimetres)'.toDouble() })
```

There's a lot more to Java streams than I can cover here, and the following resources will help build your understanding:

- Processing Data with Java SE 8 Streams is a two-part article by Raoul-Gabriel Urma from the Java Magazine that provides a good tutorial:
    - Part 1 from the March/April 2014 issue.
    - Part 2 from the May/June 2014 issue.
- The Java 8 API documentation for the Stream class and the java.util.stream package are very well documented.
- Tired of Null Pointer Exceptions? Consider Using Java SE 8's Optional! by Raoul-Gabriel Urma is also a worthwhile resource.

Chapter 5 Integrating Systems

Using the techniques garnered in Chapter 3 Solution Fundamentals and Chapter 4 Data Streams, we can easily leverage Groovy to read data from one source, transform it in some manner (e.g., reformat it or perform calculations), then send it to another source. Think of this as using Groovy to thread systems together. Groovy, Java, and third-party libraries provide a range of functionality that aids in integrating systems. For example, libraries for working with key technologies such as HTTP, message queues, RESTful web services, data stores (relational and nonrelational), and FTP all exist. However, most of these libraries leave a fair bit of work for us to do.

Luckily, Apache Camel gives us a large library of integration components and a very readable approach to describing how data is routed. Take a look at Code Listing 71: Example Camel Data Route and you'll see some of the key components for defining how data enters the route, is processed, and creates an output:

- The **`from`** method configures the source of the data for the route. The string passed to the **`from`** method describes how data is consumed—in this example, I use the `file` component to read files from the **`data_in`** directory. In fact, the code we'll see soon watches this directory for new files and processes them as they appear.

- Once a file has been read, I call **`unmarshal`** to set up an object that will hold the incoming data. In this case, I'm reading CSV data into a bean I've prepared.

- Next, I'm able to **`marshal`** the data into **`JSON`** format.

- And finally, I save the JSON data into a file in the **`data_out`** directory. I base the new file name on the original by using the that file's name (without its extension) and add the **`.json`** extension.

Some of these items above won't make a lot of sense outside of the full code listing, but hopefully you can see that Camel is dealing with all of the tricky aspects around handling the files, monitoring for new data, and converting formats.

Code Listing 71: Example Camel Data Route

```
from('file://data_in/')
  .unmarshal(new CSV(DataEntryBean))
  .marshal().json(true)
  .to('file://data_out/?fileName=${file:onlyname.noext}.json')
```

Notice that copying the previous code into Groovy Console won't yield a result. First, we need to put up some scaffolding to support the Camel route. So let's go through setting up Camel routes that work with files, message queues, and databases.

Working with files

Camel is based on Enterprise Integration Patterns (EIP), which is described well in the book Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf. Because many of our integration requirements have been around for some time, we don't need to write our own solution for this type of text and Camel's implementation—we need only to configure the framework to meet our local needs.

In the File Transfer pattern, one application exports data into a file that is imported into another application. Camel supports this pattern through a number of components:

- The FTP and FTPS components support sending and receiving files through the File Transfer Protocol.
- The Jetty component can consume and produce HTTP-based requests.
- The File component supports reading and writing files in local directories.

Let's work with the File component, as it's straightforward and easier to pick up. The File component will poll a directory and wait for new files to be added. Once a new file appears, it will be sent to the route and processed. The File component can also write files to a directory. This allows us to set up a route that polls a directory for a CSV file, process it in some manner, then store the result in a new directory.

Before we set up the route, we first need to prepare a class that can hold the incoming CSV data. We can imagine that such data might be produced by a data logger that measures the current temperature. Each measurement is recorded against the date and time (timestamp) of the recording and saved in a CSV file stored in a specific directory. Over time, this directory will contain a series of CSV files, each one containing a single line that looks something like: **`2016-02-21T09:40:28.922,916`**.

Code Listing 72: A Bean for Data Logging describes a class, **`DataEntryBean`**, that holds the two properties created by the data logger:

- The **`timestamp`** property is held as a String.
- The **`value`** property is held as an Integer.

Both properties have been marked as final so that, once we've created a **`DataEntryBean`** object, the properties can't be changed.

`DataEntryBean` provides a no-argument constructor, **`DataEntryBean()`**, that sets **`timestamp`** to the current date and time and generates a random number between 0-999 for the **`value`** property. Perhaps it's a very hot planet.

The **`toString()`** method returns a String representation of the object. You'll see that I separate the properties with a comma—handy for my CSV needs. Groovy methods and closures will return the result of their last expression, so I don't need to explicitly use the **`return`** keyword.

Hopefully, most of the syntax for creating a class in Groovy is self-evident. However, the various annotations may look a bit odd. First of all, the **@CsvRecord** and **@DataField** annotations are from Camel's Bindy component. Bindy helps us work with CSV data (among others), and the **@CsvRecord** annotation indicates that each **DataEntryBean** represents a line in a CSV file. I pass the annotation two arguments: the separator used in the CSV file and whether or not the first line in the CSV is to be skipped. This second argument lets us ignore a header row if need be.

The two properties (timestamp and value) carry a **@DataField** annotation that indicates that the property aligns with a field in the CSV. Each annotation indicates the position of the field, that we require it to be present, and that any leading/trailing whitespace should be trimmed.

Code Listing 72: A Bean for Data Logging

```
@Grab('org.apache.camel:camel-bindy:2.16.0')
import org.apache.camel.dataformat.bindy.annotation.CsvRecord
import org.apache.camel.dataformat.bindy.annotation.DataField

import javax.xml.bind.annotation.XmlAccessType
import javax.xml.bind.annotation.XmlAccessorType
import javax.xml.bind.annotation.XmlAttribute
import javax.xml.bind.annotation.XmlRootElement
import java.time.LocalDateTime

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
@CsvRecord(separator = ',', skipFirstLine = false)
class DataEntryBean {
    static random = new Random()

    @XmlAttribute
    @DataField(pos = 1, required = true, trim = true)
    final String timestamp

    @XmlAttribute
    @DataField(pos = 2, required = true, trim = true)
    final Integer value

    DataEntryBean() {
        timestamp = LocalDateTime.now().toString()
        value = random.nextInt(1000)
    }

    String toString() {
        "$timestamp,$value".toString()
    }
}
```

The **DataEntryBean** class also features a number of XML annotations, and these allow us to bind to an XML structure. We'll explore this in more depth shortly.

Now that we have a bean to hold CSV data, let's create a script that will fake a data logger for us. Code Listing 73: A CSV Producer will generate a new CSV file every second and place it into the **data_in** directory.

Code Listing 73: A CSV Producer

```
@GrabConfig(systemClassLoader = true)
@Grab('ch.qos.logback:logback-classic:1.1.3')
@Grab('org.apache.camel:camel-core:2.16.0')

import org.apache.camel.main.Main
import org.apache.camel.builder.RouteBuilder

new Main() {
    {
        enableHangupSupport()
        addRouteBuilder new RouteBuilder() {
            void configure() {
                from('timer://demoTimer?period=1s')
                .process {exchange ->
                    def data = new DataEntryBean()
                    exchange.out.body = data.toString()
                }
                .to('file:data_in?fileName=${date:now:yyyy-MM-
dd\'T\'HH:mm:ss}.csv')
            }
        }
    }

    void afterStart() {
        LOG.info 'Started Camel. Use ctrl + c to terminate the JVM.'
    }

    void beforeStop() {
        LOG.info 'Stopping Camel.'
    }
}.run()
```

First of all, **GrabConfig** is set up in the same manner we set it up in the section on Databases because it lets Camel find the component classes. Next, we Grab the **camel-core** library and import two classes:

- Main: provides an easy way for us to get a Camel-based script up and running.
- RouteBuilder: provides the basis for building a route that looks like Code Listing 71: Example Camel Data Route.

Note: You'll notice that I've imported the Logback library (logback-classic). This sets up logging for the script and is used by Camel. An associated file, `Logback.groovy`, can be found in the source and is used to configure the logging.

We next use a handy piece of syntax to create what's called an anonymous class: **`new Main() {..}`**. This essentially extends Camel's **`Main`** class on the fly. Within the curly braces (**`{}`**), we can configure the anonymous class and, once that's setup, call the **`run()`** method to start up Camel.

Here, I have performed a few tasks within the anonymous class. The easiest task to see is where I override the Main class's **`afterStart()`** and **`beforeStop()`** methods to log some useful messages.

Note: In order to avoid repetition, I won't restate the afterStart() and beforeStop() code in the remaining code listings. The full code is available in the source code accompanying this book.

Somewhat more perplexing is the block of code surrounded by another set of curly braces. This is an instance initializer block and is run whenever a new instance of the class is created. Because I'm using an anonymous class, this is run straight away. The block itself starts with a call to the **`enableHangupSupport()`** method because this allows the route to be gracefully stopped using Ctrl+C. Next is the call **`addRouteBuilder new RouteBuilder() {..}`**. This is a method call (**`addRouteBuilder`**) for setting up a Camel route. The argument to **`addRouteBuilder`** is another anonymous class, this time extending the **`RouteBuilder`** class.

Within the **`RouteBuilder`** class, we need only to override the **`configure()`** method, and this is where we set up the Camel route. I've broken out the route into Code Listing 74: The Fake Data Logger Route to help us see what's happening. I use the `timer` component to trigger every one second. In Camel, we are concerned with endpoints, and these sit at either end of a route and consume or produce messages/data. An endpoint is described using a URI that indicates the required component (e.g., **`timer:`** for the timer component) followed by the context path (e.g., the name of the timer—**`demoTimer`**) and, finally, the options for the endpoint (e.g., the timer period—**`period=1s`**). The timer endpoint is passed into the **`from`** method and fires off the route every second.

We use the **`process`** method call to define a processor that can manipulate a message. I won't get too deep into Camel's structure here, but Camel works on an exchange container mechanism in which there is an "in" and an "out" message within the container[5]. This is why the closure passed to the **`process`** method declares one parameter named **`exchange`**. As we have no real source message (just a timer event), the closure only creates a new **`DataEntryObject`** (whose constructor will create default values) and adds this object's string representation to the exchange's out message.

[5] We're only interested in the out message for these examples.

We use the **to** method to define another endpoint. In this case, I use the File component to save the message contents into a file in the **data_in** directory. I set the filename to the current date-time with a **.csv** extension. It is important to note that, while the URI contains the **${}** syntax, this isn't a Groovy String—it's using Camel's Simple Expression Language. Happily for us, Camel will create the destination directory if it doesn't already exist.

Code Listing 74: The Fake Data Logger Route

```
from('timer://demoTimer?period=1s')
.process {exchange ->
    def data = new DataEntryBean()
    exchange.out.setBody(data.toString())
}.to('file:data_in?fileName=${date:now:yyyy-MM-dd\'T\'HH:mm:ss}.csv')
```

Now that we've worked through the code, how do we run it? I'd suggest that we run it from the command line because stopping it is easier than with using the Groovy Console. Start by changing into the **camel** directory of the sample source code for the book. Then start up the script with **groovy ProducerFile.groovy** and you'll see Camel prepare its configuration and start producing output every second (e.g., **INFO camelApp - Exchange[Body: 2016-02-27T13:47:51.017,158]**). This means that Camel is now creating a series of CSV files in the **data_in** directory. This will keep going until you press **Ctrl+C**, which will shut down Camel

Note: If you check out the ProducerFile.groovy file, you'll see that it has a little extra code—specifically an extra endpoint that outputs log entries.

Once you shut down the script, check out the **data_in** directory and you'll see a fresh set of CSV files.

CSV to JSON

Now that we have created a fake data logger that generates CSV files, let's create the route for importing the CSV data. Code Listing 75: CSV to JSON Convertor sets up Camel in the same manner we've just seen but with a route configured for reading a CSV file and converting it into a JSON file. The **data_in** directory is monitored by the file component and forms the route's inbound endpoint.

The call to **unmarshal** allows us to take the incoming comma-separated record (e.g., **2016-02-27T13:38:15.891,317**) and convert it to a new data format. In this example, we call on the camel-bindy library (note the import alias for **BindyCsvDataFormat**) to set up a **DataEntryBean** object with the data from the CSV file. This is where those **@DataField** annotations used in **DataEntryBean** come into play. These guide Bindy in allocating CSV fields into Groovy object properties.

Once the data is within the bean, we can marshal it into JSON format through the calls **marshal().json(true)**. This uses the **camel-xstream** library to create a JSON representation of the data. The JSON is printed by passing **true** to the **json** method.

Finally, the File component is used to place the resulting JSON into a file stored in the **data_out** directory.

Code Listing 75: CSV to JSON Convertor

```
@GrabConfig(systemClassLoader = true)
@Grab('ch.qos.logback:logback-classic:1.1.3')
@Grab('org.apache.camel:camel-core:2.16.0')
@Grab('org.apache.camel:camel-bindy:2.16.0')
@Grab('org.apache.camel:camel-xstream:2.16.0')
import org.apache.camel.builder.RouteBuilder
import org.apache.camel.dataformat.bindy.csv.BindyCsvDataFormat as CSV
import org.apache.camel.main.Main

new Main() {
    {
      enableHangupSupport()

      addRouteBuilder new RouteBuilder() {
        void configure() {
          from('file://data_in/')
            .unmarshal(new CSV(DataEntryBean))
            .marshal().json(true)
            .to('file://data_out/?fileName=${file:onlyname.noext}.json')
          }
      }
    }
}.run()
```

Running **groovy ConsumerFileJson.groovy** will cause the CSV files in **data_in** to be read, and the equivalent JSON format will be saved into the **data_out** directory. As Camel processes the files from **data_in**, it will transfer those files into the **data_in/.camel** directory and won't be read again. Code Listing 76: Example JSON Output provides an example of a generated JSON file.

Code Listing 76: Example JSON Output

```
{"DataEntryBean": {
  "timestamp": "2016-02-21T09:40:28.922",
  "value": 916
}}
```

CSV to XML

Converting the CSV into XML doesn't require much more work than converting to JSON. First, the DataEntryBean[6] is decorated with a number of annotations:

- **`@XmlRootElement`** is used to map the **`DataEntryBean`** to an XML element.
- **`@XmlAccessorType(XmlAccessType.FIELD)`** indicates that all of the **`DataEntryBean`** fields (properties) will be mapped to XML.
- **`@XmlAttribute`** is used for each property in order to indicate that the **`DataEntryBean`** property will become an attribute rather than a subelement.

The Java Architecture for XML Binding (JAXB) provides these annotations.

We can now map a **`DataEntryBean`** to XML, and the **`camel-jaxb`** library will take care of the details. Take a look at Code Listing 77: CSV to XML Convertor—it looks much the same as the CSV to JSON version, but with the marshal call changed to **`marshal().jaxb(true)`** and the resulting file stored as XML.

Code Listing 77: CSV to XML Convertor

```
@GrabConfig(systemClassLoader = true)
@Grab('ch.qos.logback:logback-classic:1.1.3')
@Grab('org.apache.camel:camel-core:2.16.0')
@Grab('org.apache.camel:camel-bindy:2.16.0')
@Grab('org.apache.camel:camel-jaxb:2.16.0')
import org.apache.camel.builder.RouteBuilder
import org.apache.camel.dataformat.bindy.csv.BindyCsvDataFormat as CSV
import org.apache.camel.main.Main

new Main() {
    {
        enableHangupSupport()
        addRouteBuilder new RouteBuilder() {
          void configure() {
            from('file://data_in/')
            .unmarshal(new CSV(DataEntryBean))
            .marshal().jaxb(true)
            .to('file://data_out/?fileName=${file:onlyname.noext}.xml')
          }
        }
    }
}.run()
```

[6] See Code Listing 72: A Bean for Data Logging.

Tip: You might need to run `groovy ProducerFile.groovy` again if you don't have any CSV files waiting in the data_in directory.

Running **`groovy ConsumerFileXML.groovy`** will cause the CSV files in **`data_in`** to be read, and the equivalent XML format will be saved into the **`data_out`** directory. Code Listing 78: Example XML Output provides an example of a generated XML file.

Code Listing 78: Example XML Output

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dataEntryBean timestamp="2016-02-27T15:00:07.643" value="175"/>
```

Although the CSV to JSON/XML examples have been based on local files and directories, it's not a big leap to setting up routes to CSV files created in one application and made available via FTP or HTTP.

Note: For the adventurous, why not run the ProducerFile.groovy script in one console and one of the ConsumerFile scripts in another? You'll see the producer creating records and the consumer picking them up.

Working with queues

Moving on from the File Transfer pattern, let's look at how we can use Camel to place messages on a message queue and read the messages from the queue. This is the basis of the Message Channel EIP.

Let's take a moment to look at message queues. In Figure 2: High-Level View of a Message Queue, you'll see four main elements:

- The **Producer** adds **Messages** to the **Message Queue.**
- The **Message** is data—perhaps text (e.g., XML or JSON)—that we're sending from the **Producer** to the **Consumer**.
- A messaging system manages the **Message Queue**.
- A **Consumer** takes messages from the queue and, presumably, does something with the message.

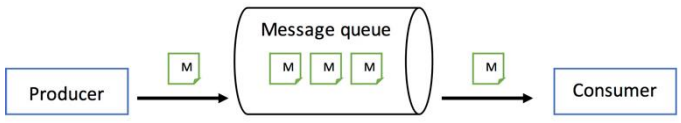


Figure 2: High-Level View of a Message Queue

There are many benefits to message queues, including asynchronous processing, which most developers prefer. Asynchronous processing allows the producer to add messages to the queue without relying on the consumer being active/ready. Each new message waits in the queue until the consumer is ready to handle them. If you've ever used email, then you've used queues—other people (producers) send you emails, and the messages wait on the mail server (message queue) until you (the consumer) are ready to read them.

Before we can use Camel with messaging, we'll need to set up an Apache ActiveMQ messaging server. ActiveMQ will manage the message queue and Camel will both produce and consume messages.

Installing Apache ActiveMQ

The ActiveMQ Getting Started guide is the best place to go for installation instructions, but the steps below should get you going quickly:

1. Open the Apache ActiveMQ download page: http://activemq.apache.org/download.html.
2. Download a release version (such as ActiveMQ 5.13.1 Release) for your operating system (Windows or Unix/Linux).
3. Extract the downloaded archive file.
4. Change into the new directory.
5. Start the server with:
    a. Windows: **`bin\activemq start`**
    b. Unix: **`bin/activemq console`**
6. Access the web-based administration interface http://0.0.0.0:8161/admin/ with:
    a. Username: **`admin`**
    b. Password: **`admin`**

If you have problems getting the server to start, check out **`data/activemq.log`** for log messages.

Once the server is running, try out the following commands:

- Start the consumer in one terminal: **`bin/activemq consumer --messageCount 100 --destination queue://TEST`**. This will consume new messages in the TEST queue and cease once 100 messages have been consumed.
- Send a message from a producer from another terminal: **`bin/activemq producer --message "Hello,\ world" --messageCount 100 --sleep 1000 --destination queue://TEST`**. This will generate a new "Hello, world" message every second and cease once 100 messages have been sent.

To clear the queues, run **`bin/activemq purge`** while the server is running or use the admin interface.

After you make sure that ActiveMQ is running, you're ready to start sending and receiving messages.

The message producer

Sending messages to the message queue isn't overly different from the code we wrote to create new CSV files[7]. Code Listing 79: Message Producer is based on a timer endpoint and the same process we'd seen previously—every second we create a new **`DataEntryBean`** object. In order to communicate with the message server, we use the ActiveMQ component, so that defining the endpoint is easy: **`'activemq:DEMO'`**. The **`DEMO`** context path designates the queue to which messages will be sent. Happily, ActiveMQ will create this queue for us automatically.

That's all we need to get messages to the server. Just run **`groovy ProducerQueue.groovy`** and you'll start to see the messages being delivered. To make sure that it's working, open the ActiveMQ web interface and browse to the DEMO queue—the URL will look like this: http://0.0.0.0:8161/admin/browse.jsp?JMSDestination=DEMO. You should see a list of messages, and you can click on an individual message to see its details.

At this point, stop the **`ProducerQueue.groovy`** script. The DEMO queue will have the messages waiting for us to consume them.

[7] See: Code Listing 73: A CSV Producer.

Code Listing 79: Message Producer

```
@GrabConfig(systemClassLoader = true)
@Grab('ch.qos.logback:logback-classic:1.1.3')
@Grab('org.apache.camel:camel-core:2.16.0')
@Grab('org.apache.activemq:activemq-camel:5.13.1')

import org.apache.camel.builder.RouteBuilder
import org.apache.camel.main.Main

new Main() {
    {
        enableHangupSupport()
        addRouteBuilder new RouteBuilder() {
            void configure() {
                from('timer://demoTimer?period=1s')
                .process { exchange ->
                    def data = new DataEntryBean()
                    exchange.out.body = data.toString()
                }
                .to('activemq:DEMO')
            }
        }
    }
}.run()
```

The message consumer

Consuming the DEMO queue messages is easy with Camel. As you can see with the route defined in Code Listing 80: A Basic Message Consumer, all we need is **`from('activemq:DEMO').to('stream:out')`**. The ActiveMQ endpoint picks up messages in the DEMO queue, and our route sends them to the console using the camel-stream component. This component allows us to write to the standard output stream—in this case it outputs to the console.

If you now run **`ConsumerQueueStream.groovy`**, you'll see the messages being read and then displayed to the screen (e.g., **`2016-02-27T15:55:28.978,655`**). The script will pick up all of the messages we created earlier and, if you go into the ActiveMQ web interface, you'll see that the queue is now empty.

Code Listing 80: A Basic Message Consumer

```
@GrabConfig(systemClassLoader = true)
@Grab('ch.qos.logback:logback-classic:1.1.3')
@Grab('org.apache.camel:camel-core:2.16.0')
@Grab('org.apache.camel:camel-stream:2.16.0')
@Grab('org.apache.activemq:activemq-camel:5.13.1')

import org.apache.camel.builder.RouteBuilder
import org.apache.camel.main.Main

new Main() {
    {
        enableHangupSupport()
        addRouteBuilder new RouteBuilder() {
            void configure() {
                from('activemq:DEMO').to('stream:out')
            }
        }
    }
}.run()
```

That consumer was quite basic so, for our final Camel script, let's consume messages and insert the data into a database. This route is one you're likely to have seen a number of times, and we can imagine a weather sensor feeding data into our system, so we'll send the data to a message queue for another system to collect and add to a database.

The script is a little more involved for defining a route that consumes messages and inserts the data into a database. Code Listing 81: Storing Message Data in a Database isn't too different from what we've already looked at, but there are a few noteworthy elements. First, in order to use the JDBC component (**`to('jdbc:demoDb')`**), we need to define the **`demoDB`** connection. This isn't as straightforward as simply adding the JDBC connection URL as an option in the endpoint definition. Instead, we need to register the connection in a registry accessed by Camel. While we could use JNDI or Spring, let's use the registry made available as a field in Camel's **`Main`** class. To define the data source (an SQLite database):

1. Create a BasicDataSource[8] object: **`def ds = new BasicDataSource()`**.
2. Set the database's JDBC URL: **`ds.url = 'jdbc:sqlite:logger_data.db'`**.
3. Add the data source to the registry field: **`registry.put('demoDb', ds)`**.

You'll note that the name I give to the connection (demoDB) matches the one used by the JDBC endpoint.

Before we declare the route, we'll use Groovy's SQL library in order to make sure the required table exists in the database.

[8] This is made available by the Apache Commons DBCP library.

The JDBC component expects that the message body will contain the SQL required to perform the action needed by the route. There are several effective ways to do this, but I've decided to demonstrate the use of a bean to prepare the SQL. In Groovy, a bean is essentially a Plain Old Groovy Object (POGO), and the **LoggerDbInsert** is a very basic class with one method. While I've declared this class within my script, you can use one from another library—just make sure it's available to Groovy (e.g., with **@Grab**).

In order to use my bean in the route, I simply include **.bean(LoggerDbInsert)** in my Camel route. You'll note that I pass the **bean** method the name of the class but not an instance/object. Camel has the smarts to locate the class, create an instance, and call the **insertRow** method with my **DataEntryBean** object created from the CSV data in the incoming message. This is easier because the **LoggerDbClass** has only one method, but note that I could have specified the method as a second argument to the **bean** call.

Tip: There is a bean component, and I could have used this in a `.to()` call, but `.bean()` is straightforward.

Code Listing 81: Storing Message Data in a Database

```
@GrabConfig(systemClassLoader = true)
@Grab('ch.qos.logback:logback-classic:1.1.3')
@Grab('org.xerial:sqlite-jdbc:3.8.11.2')

@Grab('org.apache.camel:camel-core:2.16.0')
@Grab('org.apache.camel:camel-bindy:2.16.0')
@Grab('org.apache.camel:camel-jdbc:2.16.0')
@Grab('org.apache.activemq:activemq-camel:5.13.1')
import org.apache.camel.builder.RouteBuilder
import org.apache.camel.dataformat.bindy.csv.BindyCsvDataFormat as CSV
import org.apache.camel.main.Main
@Grab('org.apache.commons:commons-dbcp2:2.1.1')
import org.apache.commons.dbcp2.BasicDataSource

import groovy.sql.Sql

class LoggerDbInsert {
    def insertRow(DataEntryBean dataEntryBean) {
        "INSERT INTO readings (timestamp, value) VALUES
('${dataEntryBean.timestamp}', ${dataEntryBean.value})"
    }
}

new Main() {
    {
        enableHangupSupport()

        def ds = new BasicDataSource()
        ds.url = 'jdbc:sqlite:logger_data.db'
        registry.put('demoDb', ds)

        Sql.withInstance(ds.url) { sql ->
```

```
            sql.execute '''
            CREATE TABLE IF NOT EXISTS readings (
                timestamp NOT NULL,
                value NOT NULL)'''
        }

        addRouteBuilder new RouteBuilder() {
            void configure() {
                from('activemq:DEMO')
                    .unmarshal(new CSV(DataEntryBean))
                    .bean(LoggerDbInsert)
                    .to('jdbc:demoDb')
            }
        }
    }
}.run()
```

So, with a little work to define the connection and add it to the registry, combined with providing a basic class for setting up the SQL Insert command, my Camel route is able to read data from the queue and insert a new row into the database. As I mentioned earlier, there are other approaches to this, and I've included two examples in the source code: **`ConsumerQueueDatabaseSetBody.groovy`** and **`ConsumerQueueDatabaseProcess.groovy`**.

Summary

This chapter has introduced you to using Apache Camel in Groovy scripts, which, as you've seen, is a powerful combination. Take some time to browse the list of Camel components and you'll find support for a wide range of systems. Camel can make it easy to try out new systems, and I hope that these examples give you a good starting point.

My example code is written as Groovy scripts and can be run as stand-alone elements. This approach can assist many solutions and may be enough for your project. However, you may wish to explore more substantial middleware solutions, and you'll be happy to learn that Camel can work with many of the existing platforms. Take a look at Apache ServiceMix, JBoss Fuse, or Camel's support for the Spring Framework if you're interested in a larger, enterprise approach.

Chapter 6 Larger Applications

Creating scripts with Groovy is both easy and beneficial, but don't limit your use of this versatile language to just scripts! The best way to build larger applications with Groovy (and Java) is the Gradle build tool. Gradle will help you manage your project's dependencies, compilation, testing, and deployment. In this chapter, we'll walk through a small Gradle project and get a feel for how it works.

You'll find the codebase described in this chapter in the **gradle** subdirectory of the sample code. We won't go through Gradle installation because the sample code for this chapter will sort that out for you. However, eventually you'll most likely want to install Gradle, so head over to the comprehensive documentation and you'll find the installation instructions.

Note: In the code listing for this chapter, you'll find some syntax not discussed in this e-book. I won't describe all of the syntax here, but it should be reasonably self-evident, and I'll recommend some resources for you to check out in the next chapter.

The build file

Take a look at Code Listing 82: The build.gradle File. What does it look like? That's right—Gradle uses Groovy to define the build file. This makes it quite readable (much nicer than XML) and also lets you use Groovy code to extend and customize your more complex builds.

Code Listing 82: The build.gradle File

```
apply plugin: 'groovy'
apply plugin: 'application'

repositories {
    jcenter()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.4.5'
    compile 'commons-cli:commons-cli:1.2'
    testCompile 'org.spockframework:spock-core:1.0-groovy-2.4'
}

mainClassName = 'demo.App'
```

Code Listing 82 performs the following sequence:

1. Two plugins are applied:
    a. The **groovy** plugin provides the functionality required to build Groovy projects.

    b. The **application** plugin packages a distribution that makes it easy to share the application we're building.

2. The **repositories** section tells Gradle to use Bintray's JCenter repository—as discussed in the section on Libraries.

3. The **dependencies** section lists all of the project's dependencies. Each dependency descriptor uses the same syntax we've been using with Grape and is assigned to a configuration:

    a. Dependencies in the **compile** configuration are required to compile our code. This project is using Groovy 2.4.5 as well as the Apache Commons CLI library.

    b. Those in the **testCompile** configuration are required to run the test code in order. In this project, we'll use the Spock Framework for creating tests.

4. The final item in the build file sets the **mainClassApp** property as the class with which the end user will run our application. This is used by the **application** plugin when preparing the distribution.

Take a look at the files for the project and you'll see the structure illustrated in Figure 3: Gradle Project Layout:

1. **README.md** is a markdown file used to provide a brief overview of the project.

2. The **build.settings** file performs a role in builds more complex than **build.gradle**, but, for this project, it simply holds the project's name.

3. The **gradle** directory contains files used by the Gradle Wrapper to download Gradle for use with the project. The Gradle Wrapper prepares two script files: **gradlew** for Unix systems and **gradlew.bat** for Windows systems.

The **src** directory contains the project's source code and test code. Using the standard configuration, Groovy files are located in **src/main/groovy** and the Spock tests are located in **src/test/groovy**. I've set up my source code into a **demo** package, hence you see **src/main/groovy/demo**. I've also got some resource files for my testing, so I've placed them into **src/test/resources**.

In order to build the project, just run **./gradlew build** (Unix) or **gradlew.bat build** (Windows). This may take a little while on your first run because Gradle will fetch a variety of dependencies.

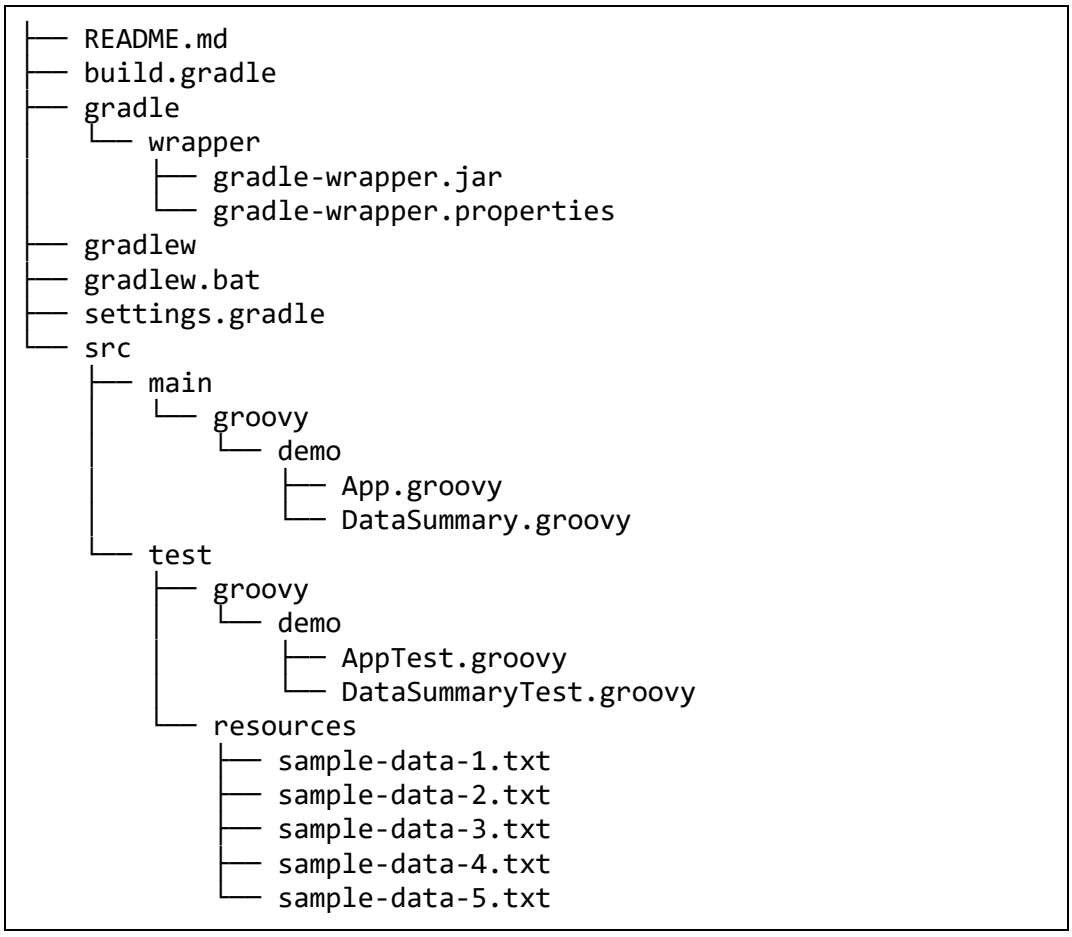


Figure 3: Gradle Project Layout

Once the build has completed, you'll find that a new **build** subdirectory has been created. Take a look in this directory and you'll see a **reports** subdirectory containing an **index.html** file. Open this file in your favorite browser and you'll see the results of the Spock tests—I trust that they all passed. You'll also find a **distributions** subdirectory—this contains two archive files (zip and tar). These archives contain the distribution items needed to run our application, including start scripts (in the **bin** directory) and the required libraries (in the **lib** directory).

Extract one of the archives into a directory and open a new terminal/command prompt at the base of this directory. Run **bin/gradle-demo -h** (Unix) or **bin\gradle-demo.bat -h** and you'll see the usage information for the application.

The sample application provided for this book will read in a file containing a list of numbers (one number per line) and will display a basic summary—for an example, see Figure 4: Sample Application Output. In order to read in a file, we use the **-f** flag followed by a file path. Copy the test files found in **src/test/resources** to the directory into which you extracted the distribution. You can then run the application with **bin/gradle-demo -f sample-data-1.txt** (Unix) or **bin/gradle-demo -f sample-data-1.txt** (Windows).

```
Count: 6
Max: 9
Min: 0
Sum: 23
Average: 3.8333333333
```

Figure 4: Sample Application Output

The code

The codebase for the project consists of two classes, **`DataSummary`** and **`App`**. Both of these are in the **`demo`** package.

> ***Tip: If you'd like to "decode" some of the code provided here, check out the Groovy documentation for object-oriented programming.***

Code Listing 83: The DataSummary Class is reasonably simple, and the constructor does the heavy lifting of determining the summary for the provided list of numbers. I'll draw your attention to one line of code, **`data = sourceData.asImmutable()`**, as it demonstrates the very handy **`asImmutable()`** method of creating a copy of the **`sourceData`** parameter and prevents the contents of the copied list from being changed. It's such as useful tool for defensive coding that I'd feel bad if I didn't point it out!

Code Listing 83: The DataSummary Class

```
package demo

class DataSummary {
    final List data

    final Integer count
    final BigDecimal sum, average, max, min

    def DataSummary(List sourceData) {
        if (!sourceData)
            throw new IllegalArgumentException
                    ('The source list cannot be empty')

        data = sourceData.asImmutable()

        count = data.size()
        max = data.max()
        min = data.min()
        sum = data.sum()
        average = sum / count
    }

    String toString() {
        """\
Count: $count
Max: $max
Min: $min
Sum: $sum
Average: $average"""
    }
}
```

Code Listing 85: The App Class provides the primary application entry point. The **`main`** class method is the standard entry point for Groovy (and Java) applications. In the case of the Groovy scripts in the previous chapter, the **`main`** method is created for us, but we need to create an entry point for classes. The **`App`** class provides a main method, and our Gradle configuration indicates that this is the entry point for our application through the **`mainClassName = 'demo.App'`** setting in the **`build.gradle`** file.

The **`App`** class's main method has a single parameter called **`args`** that will hold a list of all of the command-line arguments passed to the application. The choice of this parameter name stems from common Java usage and can be found in Groovy scripts as well. None of the scripts we've seen previously have used the command line, so I'll provide a basic demonstration in Code Listing 84: A Basic Script that Lists Args. If you were to call this script with something like **`groovy cli.groovy -f test.txt`**, you'd see the two command-line arguments (**`-f`** and **`test.txt`**) displayed each on a separate line. Essentially, the **`args`** variable is a built-in script variable.

Code Listing 84: A Basic Script that Lists Args

```
args.each {
    println it
}
```

We could prepare code to parse the command-line arguments, but Groovy provides us with the CliBuilder class, which is built on the Apache Commons CLI library. In order to set up a **CliBuilder** object, we pass it a string that provides the usage statement for the application—for the App class I've used **new CliBuilder(usage: 'App [options]')**. Next, let's use a static initializer block (**static {..}**) to prepare the **cli** class property. Within the initializer block, I've used that handy **with** method and prepared the list of acceptable command-line arguments. For each entry, we can provide the short form of the option (e.g., **f** for **-f**) and a number of additional items:

- The long form (**longOpt**) for more literate flags (e.g., **file** for **--file**).
- The number of expected arguments for the option. For the **-f** flag we expect one argument, the name of the input file.
- A display name for the option's argument (e.g., **inputFile**).
- A user-friendly description of option.

This sets up the command-line interface for our application. By calling **cli.usage()**, we can have our usage displayed in a common format, as demonstrated in Figure 5: Usage Display.

```
usage: App [options]
 -f,--file <inputFile>   The input file with one number per line
 -h,--help               Displays the usage information
```

Figure 5: Usage Display

Within the **main** method, we can have the command-line arguments parsed by simply calling **def options = cli.parse args**. When this finishes, we can access the arguments using **options.h** and **options.f**. You'll see in Code Listing 85: The App Class that I check if the application was passed the **-h** flag and, if so, displays the usage information. Otherwise, we expect that the application was called with **-f** and a file path. At various points marking the end of the application, **System.exit** is called with a number to indicate the exit status—usually 0 if there were no problems, or a number if there was a problem. I also use **System.err.println** to output error messages to standard error.

Tip: While Commons CLI library is available by default to Groovy scripts, you'll recall that I've added it as an explicit dependency for the Gradle project. This is required because the groovy-all library doesn't include commons-cli.

There are a few other syntax aspects in Code Listing 85: The App Class that I haven't previously covered. I've used the basic form of the **`if`** statement to check the command-line arguments. This consists of the **`if`** keyword, a conditional expression within parentheses—**`(options.h)`**—and a statement that is evaluated if the conditional expression resolves as true. Groovy evaluates **`options.h`** to be false if there is no value, so we don't have to check it against something. In fact, values such as **`0`** (zero), an empty string (**`''`**), and **`null`** are all seen as false to Groovy, which reduces the syntax needed in this type of check.

The last syntax element you'll notice is the throwing of exceptions—**`throw new FileNotFoundException("Path does not exist: $fileName")`**—and exception handling with the **`try-catch`** block. However, I'll leave you to pursue other resources to find out more about this.

Code Listing 85: The App Class

```
package demo

import java.nio.file.Files
import java.nio.file.Paths

class App {
    static final cli =
            new CliBuilder(usage: 'App [options]')

    static {
        cli.with {
            f longOpt: 'file',
                    args: 1,
                    argName: 'inputFile',
                    'The input file with one number per line'
            h longOpt: 'help',
                    'Displays the usage information'
        }
    }

    static loadNumberListFromFile(String fileName) {
        def path = Paths.get(fileName)

        if (!Files.isRegularFile(path))
            throw new FileNotFoundException
                    ("Path does not exist: $fileName")

        path.readLines().findAll {
            it.isNumber()
        }.collect {
```

```
            it.toBigDecimal()
        }
    }

    static void main(args) {

        def displayHelpAndExit = {
            cli.usage()
            System.exit 0
        }

        def displayErrorAndExit = { message, errorNumber = -1 ->
            System.err.println "Error: $message"
            System.exit errorNumber
        }

        def options = cli.parse args

        if (options.h)
            displayHelpAndExit()

        if (!options.f)
            displayErrorAndExit 'No input file provided', -2

        def inputData = null
        try {
            inputData = loadNumberListFromFile options.f
        } catch (any) {
            displayErrorAndExit "${any.message}", -3
        }

        if (!inputData)
            displayErrorAndExit "No numeric data in ${options.f}", -4

        println "${new DataSummary(inputData)}"

        System.exit 0
    }
}
```

Spock tests

The Spock Framework provides facilities for writing unit, behavior-driven, integration, and functional tests for Java and Groovy code. That's right, even if you have to write Java for your application code, you can use Spock to test it. And what are Spock tests written in? Yep, Groovy.

To get started with Spock in our Gradle project, we must declare the dependency for the **testCompile** configuration: **testCompile 'org.spockframework:spock-core:1.0-groovy-2.4'**. This means that the Spock library will be used to run tests, but it will not be included in the distribution. In order to set up our tests, we need only to add the test code to the **src/test/groovy** directory. The tests will be performed when you run **gradlew build**, but you can specifically run the tests by calling **gradlew test**.

Spock leverages Groovy to provide an easily readable approach to writing tests. To see how far this goes, check out Code Listing 86: Spock Test for the DataSummary Class. In order to declare the test class, we extend Spock's **Specification** class, then provide one or more tests, each defined as a method. There's a high level of human readability in the code:

- The test (method) name is a sentence that describes the test. That's right, **def "Trivial DataSummary Test with a basic data table"()** is an acceptable Groovy method signature.
- The test is divided into blocks, each containing an aspect of the test:
    - The **given** block provides initialization items for the test.
    - The **when** block provides the action for the test and is followed by a **then** block to check the result.
    - The **expect** block contains conditions that the test must meet in order to pass and represents a conflation of the **when** and **then** blocks.
    - The **where** block provides input and output data for the test.

Each block can be accompanied with a piece of text to explain what's going on. You'll see that the tests provided here use a mix of blocks, depending on what action needs to be performed.

Code Listing 86: Spock Test for the DataSummary Class

```
package demo

import spock.lang.Specification
import spock.lang.Unroll

class DataSummaryTest extends Specification {
    @Unroll("Number list: #numbers")
    def "Trivial DataSummary Test with a basic data table"() {
        given: "A new DataSummary object"
        def summary = new DataSummary(numbers)

        expect: "That the stats are correct"
        summary.count == count
        summary.max == max
        summary.min == min
        summary.sum == sum
        summary.average == average

        where: "The input data and expected summaries are"
        numbers        || count | min | max | sum | average
        [ 10 ]         || 1     | 10  | 10  | 10  | 10
        [ 10, 12 ]     || 2     | 10  | 12  | 22  | 11
        [ 10, 12, 14 ] || 3     | 10  | 14  | 36  | 12
    }

    def "Ensure that an IllegalArgumentException is thrown with an
empty list"() {
        when: "I try to create a new DataSummary with an empty list"
        new DataSummary([ ])
        then: "the correct exception should be thrown"
        thrown(IllegalArgumentException)
    }
}
```

To my mind, Spock's approach to data-driven testing is a key selling point for the framework. In both of the test classes, I've prepared a data table in the **`where`** block that allows me to easily provide a series of test inputs alongside the desired result. The format of the data table is elementary:

- The first row is the header and provides the data variable names. These names can be used within the test code and are replaced by the associated data value from the subsequent row(s).

- Subsequent rows provide input and output data. Input data is provided to the left of the double pipe (**||**) and output data to the right. Spock doesn't enforce this formatting, but I use it to aid readability. A single pipe delineates values within the input/output section.

Each (nonheader) row represents an iteration of the test. Therefore, a table with three (nonheader) rows will cause the test method to be run three times. Take a look at the data table in Code Listing 86: Spock Test for the DataSummary Class and you'll notice that we can even declare lists (and other objects) within the data table. The data variables are then accessed as variables within the test.

The **@Unroll** annotation is used to report each test iteration independently. This is useful in determining a problem, especially because we can use data variable names in the test's descriptor. For example, **@Unroll("Number list: #numbers")** results in a single test iteration being named **Number list: [10, 12, 14]** in the test report. By prefixing the hash/pound symbol (**#**) to the data variable name, we've advised Spock to perform a substitution. In fact, we don't need to provide this within the **@Unroll** annotation and could have used the data variable names in the test's method declaration (e.g., **def "Trivial DataSummary Test with number list: #numbers"()**).

In both of the example test classes, I've also demonstrated our ability to check that an exception was thrown when running the test. For example, in Code Listing 87: Spock Test for the App Class, I make sure that the call to **App.loadNumberListFromFile('')** causes an exception by checking for **thrown(FileNotFoundException)**.

Code Listing 87: Spock Test for the App Class

```
package demo

import spock.lang.Specification
import spock.lang.Unroll

class AppTest extends Specification {
    @Unroll("Input file: #file")
    def "Testing loadNumberListFromFile"() {
        expect:
        App.loadNumberListFromFile("src/test/resources/$file") == result

        where: "The input file has an associated result"
        file                || result
        'sample-data-1.txt' || [ 5, 6, 2, 9, 0, 1 ]
        'sample-data-2.txt' || [ 5, 6, 2, 9, 0, 1 ]
        'sample-data-3.txt' || [ ]
        'sample-data-4.txt' || [ 5, 6, 2, 9, 0, 1 ]
        'sample-data-5.txt' || [ -5, 6, 2, -9.3, 0.2, 1 ]
    }

    def "Ensure that a FileNotFoundException is thrown for absent
files"() {
        when: "I try to load a file that doesn't exist"
        App.loadNumberListFromFile('')
        then: "the correct exception should be thrown"
        thrown(FileNotFoundException)
    }
}
```

Summary

Gradle is a great step forward for build systems. It's more readable than XML-based approaches and easily customized and extended with Groovy. Investing some time in exploring Gradle and the range of community-submitted plugins will be worthwhile.

This chapter also introduced you to the Spock Framework. Spock provides an extremely versatile approach to testing that blends technical and narrative aspects. I've only scratched the surface here of what's possible with Spock, and I encourage you to look into it further.

Both Gradle and Spock employ the strengths of Groovy and can be utilized whether or not you use Groovy in your application code.

Chapter 7 Next Steps

By now, an experienced programmer will have noticed that I did not introduce a range of syntax elements, including loops, switch statements, and the gamut of object-oriented programming. Don't panic—these features are all available in Groovy, and you can definitely create comprehensive software with this powerful and friendly language. However, I hope that you come away with a model for approaching Groovy development that is succinct and takes advantage of Groovy's built-in strengths and the knowledge instilled into existing Groovy and Java libraries.

Before I leave you, I'll outline some key resources and Groovy-based projects.

Resources

For those wishing to learn more about the Groovy syntax, the online documentation is a great place to start. My own book, The Groovy 2 Tutorial, seeks to provide you with a solid basis in Groovy's key language elements (and it's free to read online). No Groovy developer should be without the canonical Groovy in Action 2nd edition[9], as it reveals the language and its wonderful level of flexibility in an engaging manner. I'd round out a Groovy bookshelf with Fergal Dearle's Groovy for Domain Specific Languages 2nd edition, which explores how you can create mini-languages to help solve real problems.

Groovy excels at making light work of connecting systems. As I mentioned in Chapter 5 Integrating Systems, Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf is a key text in the field. The Apache Camel website provides a good basis of documentation, but it can be a little tricky to piece everything together if you're starting from scratch. Luckily, Camel in Action 2nd edition by Claus Ibsen and Jonathan Anstey is currently being written, and the first edition is a very readable introduction into this useful integration framework.

For those considering Gradle as their new build tool, the online documentation is both comprehensive and readable and will take you a long way into Gradle before you need to look for other resources. The CodeNarc analysis tool is well worth a look when you're working with Groovy and Gradle because it guides you on good practice and possible issues in your code.

The Spock Framework has online documentation to help get you started. Java Testing with Spock by Konstantinos Kapelonis provides solid coverage of testing and Spock's feature set.

Going further

There are many Groovy-based projects out there, and you're likely to find one that helps you get the job done quickly.

[9] Also referred to as GINA and authored by key Groovy luminaries Dierk König, Paul King, Guillaume Laforge, Hamlet D'Arcy, Cédric Champeau, Erik Pragt, and Jon Skeet.

It's hard to go past Grails if you are looking to build dynamic websites and web services. The recently released Grails 3 saw the framework built on Spring Boot and utilize Gradle. Grails provides a range of services that can make it easier to get a project up and running. Take a read through the documentation and you'll see that you can easily configure data storage/persistence, internationalization, security, and a range of plugins.

Some other Groovy projects for your perusal include:

- Griffon: for writing desktop applications.
- GPars: a library for concurrent programming.
- Gaiden: for creating markdown-based documentation.

I have to admit that I haven't done any work in the mobile-application domain, but you'll find that a number of people are using Groovy in Android development. If you're interested in this area, it is worth checking out the Gradle plugin for working with Groovy & Android.

Of course, should you find a gap in the Groovy landscape, there's likely to be a Java-based project to meet your needs. Because Groovy plays so nicely with Java, you'll likely find that you can pick up a Java library or framework and extend it using Groovy.

Appendix: Code Listings

Language essentials

The various code listings in Chapter 2 Language Essentials can be copied and pasted into the Groovy Console. The **`essentials`** directory provides a few code files for general perusal/interest.

Solution fundamentals

The **`fundamentals`** directory contains the example code from the Chapter 3 Solution Fundamentals chapter.

Code listing	Code file
Code Listing 31: Reading a File	files/ReadBasic.groovy
Code Listing 32: Writing a File	files/WriteBasic.groovy
Code Listing 33: Appending a File	files/AppendBasic.groovy
Code Listing 34: Reading from a URL	web/ReadUrl.groovy
Code Listing 35: Querying a Database	database/SQLiteDatabaseQuery.groovy
Code Listing 36: Querying a Database (version 2)	database/SQLiteDatabaseAverageRainfall.groovy
Code Listing 37: Reading a CSV File	csv/ReadWeatherData.groovy
Code Listing 38: Reading a JSON File	json/ReadWeatherDataJson.groovy
Code Listing 39: Preparing JSON Output	json/RainfallFormatJson.groovy
Code Listing 41: Reading an XML File	xml/ReadWeatherDataXml.groovy
Code Listing 43: Reading XML from a URL	xml/ReadXmlUrl.groovy
Code Listing 44: MarkupBuilder for XML	xml/XmlBuilderDemo.groovy
Code Listing 46: MarkupBuilder for HTML	xml/HtmlBuilderDemo.groovy
Code Listing 49: Preparing XML Output	xml/RainfallFormatXml.groovy

Code listing	Code file
Code Listing 50: A Complete Database Usage Example	database/DerbyDatabase.groovy

Data streams

The example code for the Chapter 4 Data Streams chapter is located in the **`streams`** directory.

Code listing	Code file
Code Listing 51: Filtering a Stream	RainfallDataFilter.groovy
Code Listing 53: Mapping a Stream	RainfallDataMap.groovy
Code Listing 55: Collecting a Stream	RainfallDataCollect.groovy
Code Listing 58: Calculating an Average	RainfallDataAverage.groovy
Code Listing 59: Calculating a Sum	RainfallDataSum.groovy
Code Listing 60: Summarizing Data	RainfallDataSummarizing.groovy
Code Listing 63: Summary by Month	RainfallDataGroupingBasic.groovy
Code Listing 64: Grouping by Year then Month	RainfallDataGrouping.groovy
Code Listing 66: Average by Year-Month	RainfallDataMonthlyAverage.groovy
Code Listing 68: Sorting a Stream	RainfallDataSort.groovy
Code Listing 70: A Single-Method Chain for a CSV File and a Stream	RainfallDataSummarizingSingleCall.groovy

Integrating systems

The example code for the Chapter 5 Integrating Systems chapter is located in the **`camel`** directory.

For the file conversion examples:

Code listing	Code file
Code Listing 72: A Bean for Data Logging	DataEntryBean.groovy
Code Listing 73: A CSV Producer	ProducerFile.groovy

Code listing	Code file
Code Listing 75: CSV to JSON Convertor	ConsumerFileJson.groovy
Code Listing 77: CSV to XML Convertor	ConsumerFileXml.groovy

For the message queue examples:

Code listing	Code file
Code Listing 79: Message Producer	ProducerQueue.groovy
Code Listing 80: A Basic Message Consumer	ConsumerQueueStream.groovy
Code Listing 81: Storing Message Data in a Database	ConsumerQueueDatabase.groovy

I've provided two variations on ConsumerQueueDatabase.groovy that weren't described in the text:

- ConsumerQueueDatabaseProcess.groovy uses a processor to create the SQL Insert statement.
- ConsumerQueueDatabaseSetBody.groovy sets the message body directly using Camel's Simple language.

The logback.groovy file provides configuration for the Logback library.

Larger applications

The example Gradle project is located in the **`gradle`** directory, and the various files are described in the Chapter 6 Larger Applications chapter.