The Groovy 2 Tutorial

An introductory text for budding Groovy developers

Duncan Dickinson

The Groovy 2 Tutorial

An introductory text for budding Groovy developers

Duncan Dickinson

This book is for sale at http://leanpub.com/groovytutorial

This version was published on 2015-07-04

Tweet This Book!

Please help Duncan Dickinson by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought The Groovy 2 Tutorial - get started with #groovytutorial

The suggested hashtag for this book is #groovytutorial.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#groovytutorial

I hope that this tutorial doesn't disappoint the many contributors to the Groovy codebase

I would like to respectfully acknowledge the Yagara people - the Traditional Owners of the land encompassing Ipswich - and Elders both past, present and emerging.

Contents

Introduction i

Legal Notices vi

I Getting started 1

1. Introduction 2
2. Installing Groovy 3
3. Your first lines of Groovy 7
4. Running a script 10
5. Compiling Groovy 12
6. Comments 13
7. Statements 16
8. The `assert` statement 18
9. Reserved Words 20
10. Packages 21

II Variables 27

11. Introduction 28
12. Declaring Variables 29
13. Objects 33
14. Booleans 41
15. Numbers 42

16. Strings 46

17. Collections 53

18. Arrays 62

19. Ranges 66

20. Regular Expressions 73

21. Data types 80

III Operators 86

22. Introduction 87

23. Operator Overloading 89

24. Numeric Operators 91

25. Simple Assignment Operator 94

26. The Complement Operator 95

27. Equality Operators 96

28. Arithmetic operators 100

29. Relational Operators 104

30. Increment and Decrement Operators 107

31. Conditional Operators 110

32. Bitwise Operators 117

33. Compound Assignment Operators 123

34. String Operators 125

35. Regular Expression Operators 131

36. Collection operators 133

37. Object Operators 143

IV Control Flow Statements 150

38. Introduction 151

39. Blocks 152

40. The `if` Statement 156

41. The `switch` Statement 160

42. The `for` Loop 165

43. The `while` loop 172

44. Branching statements 174

V Exceptions 179

45. Introduction 180

46. Common Exceptions 181

47. Anatomy of an Exception 186

48. Try-Catch-Finally 188

49. How Much Should We try? 194

50. Causing an Exception 196

51. Catching Errors 200

52. Methods and Exceptions 202

VI Methods 204

53. Introduction 205

54. The Basics 206

55. Parameters 208

56. Default Values for Parameters 211

57. Named Arguments 212

58. Variable Arguments (Varargs) 214

59. Return value . . . 216

60. Throwing an exception . . . 220

61. Documenting a method . . . 222

62. Techniques . . . 224

VII Closures . . . 231

63. Introduction . . . 232

64. Introducing Closures . . . 233

65. Parameters . . . 235

66. Returning Values . . . 237

67. Closures as Method Parameters . . . 238

68. Loops and closures . . . 241

69. Useful Methods That Use Closures . . . 244

70. Closure scope . . . 247

VIII Object-oriented programming . . . 250

71. Introduction . . . 251

72. Expando . . . 252

73. The basics of OO . . . 257

74. Creating a Class . . . 266

75. Constructors . . . 269

76. Packages . . . 270

77. Access Modifiers . . . 271

78. Useful Annotations . . . 274

79. Let Go of That Instance! . . . 278

80. Static Methods and Variables . . . 280

81. Interfaces 281

82. Traits 282

83. Inheritance 283

84. Enums 284

85. Other OO Topics 285

IX Going further 287

86. The great beyond 288

Colophon 290

Introduction

I like Groovy.

I like the way it lets me use the Java skills I've built up over the years but it makes it easier to code solutions to problems. I like the way it doesn't, well, get in the way. With Groovy I can:

- Easily code small scripts to perform command-line tasks
- Tie together existing systems and libraries - leveraging the breadth and depth of existing Java-based projects
- Write entire applications that can be deployed onto any system running the Java Virtual Machine (JVM) - without having to tell people the code isn't in Java.

Groovy programs run on the Java Virtual Machine (JVM) and the JVM is installed across a huge spectrum of systems: desktops, servers, mobiles and the Internet of Things. Importantly, the Java world has been going through a great renewal phase. Where once the JVM and the Java programming language were almost synonyms, a range of JVM-based languages have appeared: Groovy, Scala[1], Clojure[2], Jython[3]. These aren't languages that run within a Java program (though many can do just that), these are languages that compile down to JVM bytecode and run in a way that means you don't even need to tell your SysAdmin that you didn't write it in Java. What's more, we're not at the "cutting edge" of this approach - it's been going for long enough that you can expect a level of stability that supports the use of these languages in real application development.

In the following tutorials I aim to guide you through the basics of Groovy. I haven't really focussed on writing a "How to program" guide nor do I expend a lot of words comparing Groovy with Java. You may also notice that I haven't included the output of most of the code examples - this is to save some clutter in the text and encourage you pop open a groovyConsole and try the code for yourself. Go on, you know you'll love it.

I really hope that this tutorial gives you a basis in understanding the Groovy language and I hope that you start to see why I like Groovy so much.

Did you know?

As of March 2015 Groovy entered the incubation phase at Apache? This is the first step towards having Groovy become a full-blown Apache project.

[1]http://www.scala-lang.org/
[2]http://clojure.org/
[3]http://www.jython.org/

Something Wrong?

If you find something wrong I'd really appreciate you letting me know via either:

- A Leanpub email message[4]; or
- Through the Feedback tool[5]

Please remember that I'm not a big corporation or an automaton - I can't respond immediately to queries and I am an assemblage of emotions that respond well to positive encouragement a lot better than negativity.

Bookmarks

Between Groovy and Java there's a lot of useful websites - this is a few key ones you'll want to have in your browser.

Core resources to have at-hand:

- The Groovy homepage[6] is a great starting point.
- The latest Groovy API documents are available at:
  - The Groovy API[7]
    - aka the GAPI
  - The Groovy extensions to the JDK[8]
    - aka the GDK
- The Java API[9] always comes in handy
  - So do The Java Tutorials[10]
- Whilst not a perfect fit for Groovy, I tend to use the Google Java Style[11] as my stern mentor.

Great blogs and sites that provide Groovy coding examples:

- Mr Haki[12] - A really useful site containing heaps of Groovy code examples.
- DZone[13] - Features a number of Groovy articles.

[4] https://leanpub.com/groovytutorial/email_author/new
[5] https://leanpub.com/groovytutorial/feedback
[6] http://www.groovy-lang.org/
[7] http://groovy-lang.org/api.html
[8] http://groovy-lang.org/gdk.html
[9] http://docs.oracle.com/javase/7/docs/api/
[10] http://docs.oracle.com/javase/tutorial/
[11] http://google-styleguide.googlecode.com/svn/trunk/javaguide.html
[12] http://mrhaki.blogspot.com.au/search/label/Groovy
[13] http://www.dzone.com/links/tag/groovy.html

- InfoQ[14] - Another useful site for Groovy articles.
- My own Workbench[15] blog is very young but has a few Groovy samples.

The book Groovy in Action (2nd Edition)[16] will help you go further with your Groovy programming.

If you find yourself stuck and needing some help, the following channels are worth tuning into:

- StackOverflow's Groovy tag[17] is really useful
  - search for an answer before posting a question
  - Check out their article "How do I ask a good question?[18]" - it's an excellent outline for asking in a way people may answer
- The Groovy Mailing lists[19] are also worth joining and searching

I suggest to anyone within the orbit of Java that "Effective Java (2nd edition)" by Joshua Bloch should not only be read (repeatedly) but always at-hand.

Conventions Used in This Book

I've tried to present this book in a manner that will suit both the reader that likes a linear path of front-to-back and those who like to dip in on specific items.

Code

Code is displayed using a `monospaced` font.

Code presented within regular language looks something like:

> Use of `println` as your primary testing framework is discouraged.

Blocks of code are presented as follows:

[14]http://www.infoq.com/groovy
[15]http://workbench.duncan.dickinson.name
[16]http://www.manning.com/koenig2/
[17]http://stackoverflow.com/questions/tagged/groovy
[18]http://stackoverflow.com/help/how-to-ask
[19]http://groovy-lang.org/mailing-lists.html

A block of code

```
def name = "Billy"
println "Hi there $name"
```

Some code doesn't feature a title and is presented so as to be more aligned with the flow of the text:

```
def name = "Billy"
println "Hi there $name"
```

I've opted not to display line numbers with code as it makes copy and paste difficult.

Asides

I use a variety of asides to note information. These appear with an icon and some text and, on most occasions, feature a title.

An informative message

These provide some extra context

A tip

These are handy tips

A warning message

These point out things to worry about

Your Rights and This Book

I'm making the "Groovy Tutorial" freely available because I feel that open source projects such as Groovy deserve to have a variety of documentation that helps people use open source software. This body of work is one that took a significant amount of unpaid time but I have benefitted from many people's work in developing open source software and the associated, freely available text, that they make available.

This work is licensed under a Creative Commons Attribution License - this means that you have the right to share and adapt the text as you see fit *but* you must give me *"appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use"*. If you decide to use the whole text or parts thereof in a manner that derives you an income I think it'd be civil of you to consider contributing to my retirement fund.

All code samples are licensed under the Apache License, Version 2.0[20]

The "source code" for this book is written in Markdown, based on the LeanPub Manual[21]. You can access the source from my Git repository[22].

I don't provide any warranties, guarantees or certainties[23]. You should use this text to help you in your Groovy coding but you are responsible for your journey :)

[20]https://www.apache.org/licenses/LICENSE-2.0.html

[21]https://leanpub.com/help/manual

[22]https://bitbucket.org/duncan_dickinson/groovy-tutorial

[23]... or pekignese

Legal Notices

The Groovy logo that features on the cover page was sourced from the groovy-website GitHub project[24]

If you believe that any part of this text infringes on your intellectual property, copyright, trademark(s) or any other legal structure then please contact me[25] - I'm sure we can sort it out.

[24]https://github.com/groovy/groovy-website/blob/master/site/src/site/assets/img/groovy-logo-colored.svg

[25]mailto:groovy@duncan.dickinson.name

I Getting started

1. Introduction

Getting to know a new programming language can be both fun and, sometimes, frustrating. In this section we'll get Groovy installed, write the mandatory "Hello, World" program and look at some of the basic aspects of Groovy.

2. Installing Groovy

Before we start anything let's get Groovy installed and running.

There are a few methods for installing Groovy on your system and this section will describe the basics for getting started. Before you get started make sure that any installations are for the versions listed in the table below:

System	Version
Java JDK	8 (7 is fine too)
Groovy	2.4.0 (or higher)

The vagaries of installations

It is notoriously difficult to provide this type of information in a stable manner as versions are upgraded and URLs broken. The Groovy Download[1] page is the primary resource to return to if the links below appear to be broken.

Don't Install Groovy

I know this chapter is about installing Groovy but perhaps it's worth leaving this until later. The Groovy web console[2] is an online Groovy console that should let you run most of the sample code provided in this tutorial. Provided you have web access you can skip through to the next chapter and get straight into writing some code!

Install a Java Virtual Machine

All methods for installing Groovy require that you have a Java Virtual Machine (JVM) installed. For the purposes of this book we'll use the Java 8 SE JDK (Java 8 Standard Edition Development Kit). You'll notice that Java 8 may also be represented as Java 1.8 - the former is the Java Platform version number (8) and the latter is the version string (1.7).

[1]http://groovy-lang.org/download.html
[2]http://groovyconsole.appspot.com

JDK not JRE

If you look around for Java downloads you'll likely come across Java Runtime Environment (JRE) downloads. The JRE provides enough functionality to run a compiled application but not to compile your Groovy code. You'll need the JDK to create Groovy programs.

To install the Java JDK, head to the Oracle site and locate the download appropriate to your platform: Oracle JDK Downloads[3]. For the most part these installs are straight-forward and have enough associated documentation so we won't go through this step-by-step.

Before moving onto the Groovy installation I'd like to make you aware that the Oracle JDK is not the only version of the JDK available. As you become more comfortable with working in a Java-based environment you might like to take a look at the Open JDK[4].

Install Groovy

Once you have the JDK installed it's time to get Groovy. We'll be using Groovy 2.4 throughout this book. Other (newer) versions may work fine - it's just that the examples used throughout the book have been tested against Groovy 2.4.

The following subsections will guide you through an installation based on which platform you're using.

Windows

The Groovy Download[5] page includes links to a Windows installer - download this and run the installer using the provided defaults (it's as easy as that).

Just make sure you're downloading a 2.4.x version!

Checking for Groovy in the Start Menu

Once you have the JDK and Groovy installed you should see the GroovyConsole application in your Windows 7 Start menu. Start up the GroovyConsole application and you should be good to go.

Mac OSX and Linux

The Groovy enVironment Manager (GVM) is the best tool for getting Groovy running on your system. The GVM homepage is http://gvmtool.net/ but you don't need to visit it to run an installation.

[3]http://www.oracle.com/technetwork/java/javase/downloads/index.html

[4]http://openjdk.java.net/

[5]http://groovy-lang.org/download.html

If you are comfortable with using the terminal then you just need to run the following command as a regular user[6]:

```
curl -s get.gvmtool.net | bash
```

Once the `gvm` has been installed, run the following command to determine which versions of Groovy are available:

```
gvm list groovy
```

You'll see a large table of version numbers but are most interested in those marked with 2.4.x - you'll want the version with the highest value of x (e.g. 2.4). To install Groovy you now just call `gvm` as below:

```
gvm install groovy 2.4.0
```

Checking Groovy (all platforms)

Once you have the JDK and Groovy installed, run the following command to double-check your Groovy version:

```
groovy -v
```

You should see something like the following output:

```
Groovy Version: 2.4.0 JVM: 1.8.0_31 Vendor: Oracle Corporation OS: Mac OS X
```

This tells me that I am running:

- Groovy 2.4.0
- The Oracle Java 8 JVM
- The Apple Mac OS X operating system

[6]The `curl` command line tool is used for transferring data. It's very likely that your Linux distribution's package manager (`yum`, `apt-get` etc) includes a copy of cURL but if it doesn't, visit http://curl.haxx.se/ to download it.

Alternatives

The Groovy Download[7] page provides binary and source releases for Groovy. These are perfectly fine to use but you'll need to setup your system path to get up and running.

For those on Mac OS X you can also explore one of the following package manager options:

- MacPorts[8]
- Homebrew[9]

Linux users may find Groovy packages in their distribution's package repository but check the version being installed.

[7]http://groovy-lang.org/download.html
[8]https://www.macports.org/
[9]http://brew.sh/

3. Your first lines of Groovy

It has become a tradition (maybe an unwritten law) that the first lines of code that you write in a new language are to output the line `hello, world`. So let's start up the Groovy Console and get going.

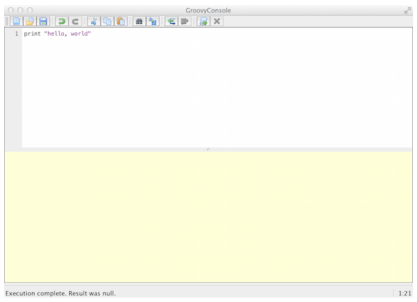
The Groovy Console provides a handy environment for preparing and testing basic Groovy scripts. In order to open the console you need to start a command line (or terminal) session and enter the following command:

Start the console

```
groovyConsole &
```

On Windows… Windows users may also find a link to `Groovy Console` in their Start menu.

The Groovy Console should look something like the following screen grab:



```
print "hello, world"
```

```
Execution complete. Result was null.                1:21
```

Screen shot of the Groovy Console application window

The main parts of the console are:

1. The top half is the editor area for adding your Groovy script
2. The bottom half is the output area that displays the results of your script
3. The menu provides the standard tools for opening and saving files (`File`) and cut/copy/paste (`Edit`)
4. The `Script` menu gives you a number of functions you'll use as you work through this book:

    1. `Run` will run the script
    2. `Run Selection` allows you to select (highlight) part of your script and run only that section
5. The `View` menu lets you reset the output area (`Clear Output`)
    1. I'd suggest that you select `Auto Clear Output on Run` as this helps reduce confusion

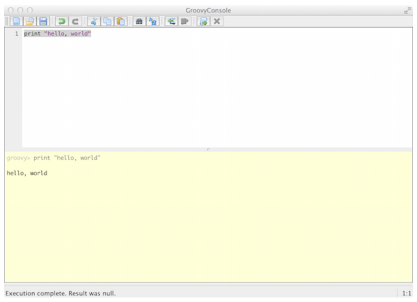
Once you have the Groovy Console open, enter the following line in the editor area:

Let's get groovy

```
print 'hello, world'
```

Groovy web console Don't forget that you can also try out the Groovy web console[1] - it'll run this code just fine.

Once that's ready, go to the menu and select `Script -> Run` and you should see your output in the bottom half of the window something like the image below:



```
print "hello, world"
```

```
groovy> print "hello, world"

hello, world
```

```
Execution complete. Result was null.    1:1
```

Screen shot of the Groovy Console application window with the hello, world script

If you see the output `hello, world` then congratulations - you've taken your first step into a larger world.

Examining the script

Our first Groovy script is very simple: it uses the `print` method (function) to output the string `hello world` to the console.

For those that have come from languages such as C++ and Java the script `print "hello, world"` probably appears to be missing items such as imported libraries for output and "container" or "boilerplate" code that sets up the context of the code. In fact, if we were to write this code in Java it would look something like:

[1] http://groovyconsole.appspot.com

Hello,world - Java style

```
class Hello {
    public static void main(String[] args) {
       System.out.print("hello, world");
    }
}

Hello.main();
```

When I look at the code above I see why Groovy is so appealing to me:

1. Groovy lets me focus on solving the problem and not working through so much decoration code.
    - Groovy doesn't need semi-colons at the end of each statement
    - Groovy essentially builds the `Hello` class around the script
2. The Groovy code is much more readable and this *should* help reduce bugs (or at least make finding them easier)
3. Most Java code is valid Groovy code - you can copy that Java code into the Groovy Console and run it - it will work
4. Groovy lets you use the comprehensive standard Java libraries and the extensive third-party libraries written by the Java developer community.
    1. But also extends these standard libraries with some great timesavers.

Groovy gives us the brevity and flexibility of a scripting language (such as Python, Ruby and Perl) whilst letting us tap into the galaxy of existing Java libraries.

4. Running a script

Now that we can output something to the screen, let's try and make our example a little more personal. Clear the Groovy Console editor and enter the following:

Using command-line arguments

```
println "hello, ${args[0]}"
```

Before we try to run this, let's look at what's in the code:

1. `print` has become `println` - this does the same thing as our previous use of `print` but adds a new line at the end of the output.
    - This makes the output easier to read when we're running on the command line
2. Instead of the text `world` we're now using `${args[0]}`:
    - `args` is a variable (an array[1]) that holds any command-line arguments we pass to the script
        - You may have noticed `String[] args` in the Java version of `hello, world` - essentially Groovy is writing that segment of code for you.
        - `args[0]` is the first element in the `args` array - this is the first parameter (command-line argument) passed to the script
    - The `${...}` notation tells Groovy that the contents need to the resolved into a value - in this case Groovy needs to determine the value of `args[0]` before displaying the output

Don't worry if this appears to be a big jump from our `hello, world` - there's a range of concepts being introduced and we'll work through them in this tutorial section. For now, put the code into your Groovy Console and know that, when run, your script will say hello to a specified person.

You now need to save your script so go to the `File` menu and select `Save`. When prompted, name the file `Hello.groovy` and save it into a directory you can access.

Unfortunately we can't run this script in the Groovy Console as it doesn't provide an option for passing in a command-line parameter. Follow this process to run the script:

1. Open a command prompt (terminal) and change to the directory (`cd`) into which you saved `Hello.groovy`.
2. Type the command `groovy Hello.groovy Newman` and press the `return` key

You should see the following output:

[1] More about arrays in a little bit

```
hello, Newman
```

Of course you can change “Newman” to be any name so feel free to try out your name, the dog’s name etc. However, make sure you add a name - your script needs that parameter or you’ll see a disconcerting error.

5. Compiling Groovy

You can compile a Groovy script into a `class` file - the same type of file that Java developers compile their code into. The resulting `class` file is in bytecode format that can be read by the Java Virtual Machine (JVM). Once compiled to bytecode, Groovy code can work on the same JVM that runs existing Java systems - this is extremely handy if you work in a Java-centric organisation but want to use Groovy.

In order to compile `Hello.groovy` we will use the `groovyc` command in the command-prompt as follows:

```
groovyc hello.groovy
```

When you look at the directory contents you should now see a file named `Hello.class`. Don't try to read the contents of the file - it's now in bytecode.

In order to run your new application you enter the following command:

```
groovy --classpath=. Hello Jerry
```

You should see the output `hello, Jerry`.

This uses the `groovy` command to run your compiled script but how?

1. The `--classpath=.` parameter tells `groovy` to look for `.class` files in the current directory (denoted as `.`)
2. 'Hello' is a little confusing until we remember that we named our script file `Hello.groovy`
    - When `groovyc` compiled our script it generated a class named `Hello` with a `main` method around our single line so that, behind the scenes, the code turned into something like the Java version discussed earlier
3. And `Jerry` is the name of the person we're greeting

The call to `groovy` basically says "Run the main method in the Hello class and pass it Jerry as the parameter".

6. Comments

Comments are used to clarify sections of Groovy code. Groovy supports two types of comments - single line and multi-line.

Comments are read by the Groovy compiler - they're purely used to help humans follow your code. They're really important once your code becomes more complex and your programs larger. Key places you'll see/use comments are:

1. When a complex algorithm is being used
2. When specific business logic is being implemented
3. For documenting interfaces that other coders will use
4. To remind you why you chose one approach over another - really handy when you revisit the code in 6-weeks and say "why did I do it that way?".

Single-line comments

A single-line comment is introduced with two forward slash characters `//`:

Single-line comment

```
//This is a single-line comment
println "hello, world"
```

Single-line comments can be appended to a Groovy statement:

Inline comments

```
def radius = 10
def pi = 3.14 //This is not very precise
def area = pi * (radius * radius)
```

Multi-line comments

A multi-line comment is introduced by the characters `/*` and terminated with the characters `*/`. Generally, the `/*` and `*/` appear on their own line:

Multi-line comments

```
/*
This is a multi-line comment
and here is the second line
*/
```

Multi-line comments are most commonly formatted with an asterisk (*) on each line, aligned with the introductory asterisk as follows:

Formatting multi-line comments

```
/*
 * This is a multiline comment
 * and here is the second line
 */
```

Multi-line comments can be introduced and terminated on a single line:

Multi-line one liners

```
/* This is a multiline comment on a single line */
```

Nesting within a multi-line comment is not possible, rendering the following code invalid:

Don't nest comments

```
/*
 * Multi-line comments cannot
 * /* be nested */
 */
```

Usage to avoid

In a similar vein to single-line comments, multi-line comments can be appended to a statement. However, the single-line comment is generally more readable than the following example:

```
def radius = 10
def pi = 3.14 /* This is not very precise */
def area = pi * (radius * radius)
```

Even less expected is a multi-line comment appended to a statement in the following manner:

```
def radius = 10
def pi = 3.14 /* This is not very precise
                and should really use java.lang.Math.PI */
def area = pi * (radius * radius)
```

In such a case the multi-line comment should appear above the statement being discussed:

```
def radius = 10
/*
 * This is not very precise
 * and should really use java.lang.Math.PI
*/
def pi = 3.14
def area = pi * (radius * radius)
```

Use of a comment within a statement should never be contemplated as it results in code that is hard to read:

```
def radius = 10
def pi = 3.14
def area = pi * /* I should find out how to square */ (radius * radius)
```

Groovydoc

Java provides a very handy tool for documenting the outward-facing aspects of your code - i.e. those items that others may reuse - it's called javadoc[1]. Groovy has its own version called `groovydoc`. Essentially this is the same tool as `javadoc` but is run over groovy code. I won't go into `groovydoc` now - it'll come up when we explore object-oriented programming.

[1]See the Javadoc guide

7. Statements

A Groovy program is made up from lots of statements, each telling the computer to do something.

A Groovy statement is generally completed by an end-of-line (EOL) character such as a carriage return:

```
def num = 21
println num
```

A semicolon (`;`) can be used to explicitly mark the end of a statement however this is deemed to be redundant in most cases and spoils readability:

```
def num = 21;
println num;
```

The backslash (`\`) is used indicates that a statement continues on the next line. The example below uses continuation to break up a long statement:

```
def solution =  1 * 2 * 3 \
                * 4 * 5 * 6
```

Without the backslash the code above would cause an error but a minor rewrite will work:

```
def solution =  1 * 2 * 3 *
            4 * 5 * 6
```

I would suggest the first version is easier to read and explicitly indicates that you intend to carry into the next line. However, statements can span multiple lines without a backslash provided Groovy can determine that the lines make up a single statement. This feature should be utilised if it aids in improved readability - this is often referred to as *line-wrapping*. For example, an array declaration that provides a number of entries may be written as:

Okay formatting

```
def myArray = ['Tasmania', 'Victoria', 'New South Wales', 'Queensland', 'Western\
 Australia', 'South Australia']
```

The judicious use of line-wrapping may improve readability:

Better formatting

```
def myArray = ['Tasmania',
               'Victoria',
               'New South Wales',
               'Queensland',
               'Western Australia',
               'South Australia']
```

It is difficult to provide specific metrics regarding readability in these cases and the programmer is left to determine the best use of white space and placement.

Usage to avoid

A semicolon can be used to separate two statements appearing on the same line:

One statement per line please

```
def num = 1 + 1; println num
```

The presentation of multiple statements in a single line should be avoided - it's not easy to read and is likely to trip you up at some point.

Groovy is very forgiving of statements spread over more than one line but usage such as the one below should be avoided as it reduces readability. For example, the following code will actually work but it looks odd and isn't worth the hassle:

Keep things together

```
def num = 1 +
1
println num
```

8. The `assert` statement

The “assert” statement is handy for checking if we have the correct result or if there was a problem in our code.

The `assert` statement is perhaps out of order being described here but it will be relied on in many code examples.

Equality and inequality

Two operators are used in the examples below: `==` (two equals signs) is the equality operator and `!=` is the inequality operator. Both are discussed later.

The `assert` statement evaluates a boolean expression (one that is `true` or `false`). If the result is `false` then the assertion has failed, the program is halted and an error is reported. The following example provides an obviously incorrect statement:

Basic assert

```
assert 1 == 2
```

An expression can be appended to the `assert` statement after a semicolon (`:`):

Assert with expression

```
assert true == false : 'true cannot be false'
```

The second expression can be anything Groovy can evaluate and the result is used in the error message. The following example will (unhelpfully) place the number “8.0” in the error message:

Assert with expression

```
assert true == false : Math.sqrt(64)
```

Handling failed assertions

For the purposes of our tutorial scripts, using asserts is a handy way to demonstrate a result for a problem. However, it's not good practice to have a program suddenly just quit when an assertion fails. When you start writing large programs, your code should aim to "fail gracefully" unless it's really in a position where bailing out is the only option.

Groovy (unlike Java) does not provide a mechanism for turning off assertions so be careful about where you use the `assert` statement in larger systems. Remember that a failed `assert` raises an `Error` (which signals a critical problem) rather than an `Exception` (from which a program is more likely to recover). Arguably, in running (production) systems, assertions are best suited to dark places in code that should never be reached - they flag when the extremely unlikely condition has happened.

The error raised by a failed assertion can be caught within a `try-catch` and handled but this isn't how errors are usually treated (we normally just let them happen). The following example illustrates a class handling a failed assertion by logging the problem - don't be concerned if you don't follow the code as it utilises a number of concepts not yet visited:

Handling failed assertions

```
import groovy.util.logging.*

@Log
class AssertionTest {
    static void runTest() {
        try {
            assert true == false : 'true cannot be false'
        } catch(AssertionError err) {
            log.severe "An assertion failed ${err}"
        }
    }
}

AssertionTest.runTest()
```

Annotations

You may have noticed `@Log` in that last code snippet. This is called an annotation and is marked using an "at" (`@`) sign followed by a class name. These are used to markup code and will be discussed in a later chapter.

Although it's Java-focussed, check out the Programming with Assertions guide[1] for more information.

[1] http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html

9. Reserved Words

Groovy reserves a set of keywords that cannot be used as identifiers.

Groovy's reserved words

abstract	as	assert	boolean	break
byte	case	catch	char	class
const	continue	def	default	do
double	else	enum	extends	false
final	finally	float	for	goto
if	implements	import	in	instanceof
int	interface	long	native	new
null	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	threadsafe	throw
throws	trait	transient	true	try
void	volatile	while		

Groovy relies of a number of key words that it reserves for its own use. This means that you shouldn't use these words for the names of variables and other items you declare. For example, the code below won't run as `package` is a reserved word:

```
def package = 'my package'
```

10. Packages

Groovy lets you take advantage of the core Java packages (the JDK) so let's take a quick look at how this works.

Except for very small programs, most Groovy and Java-based programs are made up of packages of code:

- The `package` keyword is used to designate that a class is part of a package and we'll cover this more fully in the tutorial on object-oriented Groovy.
- The `import` keyword is used to import classes from other packages into a program.

Consider the sample code below:

Using imports

```
package test

import java.time.Year

println Year.now()
```

I've indicated that this code:

- Is part of a `package` named `test`
- Needs to use the `Year` class defined in the `java.time` package

This notion of packaging allows for thousands of developers to create classes and packages without clashing. If another developer creates a `Year` class but put it into a package with a name other than `java.time` then all will be well. Oh, and you'd never start your own package name with `java.` - that really won't work out well for you[1].

Before you write any new code you should always check out these resources in the order I've given below:

[1]There's actually a package naming convention that is very easy to follow.

1. The Groovy API (GAPI)[2]
2. The Groovy extensions to the JDK (GDK)[3]
3. The standard Java classes (JDK)[4]

Using the order I've provided above lets you look at the libraries providing the Groovy approach first (the GAPI and GDK) then looking at the Java standard library (JDK).

For the rest of this chapter I'll focus on `import` as that will help us in the early set of tutorials.

Using `import`

You can `import` other classes in a variety of manners - let's take a look.

Basic Imports

The basic form of imports are the most commonly seen and you should get accustomed to them pretty quickly.

`import java.time.Year`

This will import the `Year` class from the `java.time` package

`import java.time.*`

This is a star (wildcard) import

This will import all classes in the `java.time` package

Static imports

Static imports can help your code look a little cleaner as they give you an easy way to refer to useful constants and functions (methods) declared in other code packages.

`import static java.lang.Math.PI`

This is a *static* import

This lets you import static items from another class

In this example I've imported the `PI` constant from the `java.lang.Math` class and can now use it as if it was just part of my code: `println PI`

[2] http://docs.groovy-lang.org/docs/groovy-2.4.0/html/gapi/
[3] http://docs.groovy-lang.org/docs/groovy-2.4.0/html/groovy-jdk/
[4] http://docs.oracle.com/javase/8/docs/api/index.html

`import static java.lang.Math.PI as pi`

This is a *static* import with *aliasing*

This is the same as the previous `import` but I can use the `as` keyword to rename the item being imported - I've decided to use `PI` but refer to it using the lowercase form (`pi`)

`import static java.util.UUID.randomUUID as generateId`

This is also a *static* import with *aliasing* but I've imported the `randomUUID` static method and given in the alias `generateId`

I can now call `println generateId()` in my program

`import static java.lang.Math.*`

This is a *static star* import and will import all static elements described in `Math` and let me refer to them directly in my program.

I've thrown the term `static` around a lot here - don't worry too much about this for now as we'll really only need basic imports for now. The notion of `static` will be covered when we get to object-oriented Groovy.

Built in Libraries

The following libraries are imported into Groovy by default - you don't need to do anything to start using them:

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

Groovy is able to make use of classes within these packages without explicitly `import`ing them. You can still declare them with `import` and you'll notice various development environments (IDEs) will do this regardless of Groovy's default - either way it'll be groovy.

Useful third-party libraries

There is an extensive body of existing Java libraries available to the Groovy developer and it's best to do some investigating before you write your own code - re-using well-supported libraries is a real time saver - here's a couple to take a look at:

1. Apache Commons[5]
2. Google Guava[6]

In the olden days (in Java-time) you'd often have to download the third-party library you wanted, download any other libraries it depended on, store them in the correct place (called a Classpath) and then you could start using it. Time went by and systems such as Apache Maven[7] came along to make it easier to grab a copy of your dependencies. This then lead to The (Maven) Central Repository[8] and made it even easier to grab the libraries you needed.

Other repositories

There are other repositories you can use too - you'll likely come across these as you write more code and seek out new dependencies.

Grape

Whilst you can use Maven or (even better) Gradle[9] to grab these dependencies, Groovy includes a dependency manager called Grape[10] that you can start using straight away.

Say I wanted to grab a copy of my favourite web page and had worked out that Apache's HTTP Components[11] would really help me. I can search the Maven Central Repository and find what I need[12]. In fact, that web page even tells me how to use the library with Grape:

[5]http://commons.apache.org/
[6]https://code.google.com/p/guava-libraries/
[7]http://maven.apache.org
[8]http://search.maven.org
[9]http://gradle.org
[10]http://groovy-lang.org/grape.html
[11]https://hc.apache.org/index.html
[12]http://search.maven.org/#artifactdetails%7Corg.apache.httpcomponents%7Chttpclient%7C4.3.6%7Cjar

Grape example

```
@Grapes(
@Grab(group='org.apache.httpcomponents', module='httpcomponents-client', version\
='4.4')
)
```

Searching the repository can be a little frustrating at first but you'll get the hang of it over time.

Grape uses annotations - essentially the "at" (`@`) sign followed by a name - to do its thing. In the example above:

- `@Grapes` starts of the grape listing
  - You need this if you're grabbing several libraries in the same segment (node) of your code - we can actually ignore this in smaller examples.
- Each grape is declared using `@Grab` and providing the following:
  - The `group` that holds the module
  - The name of the `module`
  - The required version of the `module`

In the code below I use the Apache HTTP Components library to report on the HTTP status line from my request to "http://www.example.org". I've trimmed off the `@Grapes` as I just need to `Grab` one module:

Using Grape

```
@Grab(group='org.apache.httpcomponents', module='httpclient', version='4.3.6')
import org.apache.http.impl.client.HttpClients
import org.apache.http.client.methods.HttpGet

def httpclient = HttpClients.createDefault()
def httpGet = new HttpGet('http://www.example.org')
def response = httpclient.execute(httpGet)

println response.getStatusLine()
```

Where to grab

I like to put my `@Grabs` on the line above the `import` that include the dependency - it helps other see where my libraries are coming from.

You can use a short-form version of `@Grab` using the format `<group>:<module>:<version>` - this would let us use the following:

Short-form grape

```
@Grab('org.apache.httpcomponents:httpclient:4.3.6')
```

Once you start building more complex programs you will probably turn to Gradle but Grape works just fine for these tutorials.

II Variables

11. Introduction

Variables are (perhaps unsurprisingly), items that can change. Essentially a variable is a “box” that can hold a value. Groovy is a “dynamic language” in that it allows you to easily store and manipulate variables regardless of their value. This places it in similar company to Python and Ruby but, as a child of Java, Groovy can also operate as a “typed language”. In typed languages we can specify the data type (e.g. a number or piece of text) of the variable.

Groovy lets us work in both language modes - dynamic and typed - and this flexibility makes it that much easier to use.

12. Declaring Variables

Let’s look at how you declare a variable and what it is you’re actually declaring.

Groovy provides a few ways to create a variable but the best one to start with is to use the `def` keyword. In the example below I define (`def`) a new variable named `score` that can be used to hold a value later in my program:

Defining a variable

```
def score
```

In the next example I assign `score` a value of 10 and ask Groovy to display the value of `score` using `println`:

Using a variable

```
def score
score = 10
println score
```

Instead of declaring `score` and then assigning it the value `10` I can do this on a single line using `def score = 10`. I do just this in the example below and then change the value of `score` (it is a variable after all) - try this in your Groovy Console and the `println`s will show you the value of `score` after it’s been set.

Changing the value

```
def score = 10
println score
score = 11
println score
```

You’ll note that the second time I use `score` I don’t need the `def` prefix as I’ve already declared `score` and don’t need to redeclare it.

If we’re declaring a number of variables we could provide a `def` on each line:

```
def myNumber
def myName
```

Alternatively, the previous example could be represented on a single line in which each variable is separated by a comma (`,`):

```
def myNumber, myName
```

You can assign values to variables defined on a single line:

```
def number1 = 10, number2 = 20
```

A set of variables can be assigned values from a list (multiple assignment):

Multiple assignment

```
def number1, number2
(number1, number2) = [10, 20]

assert number1 == 10
assert number2 == 20
```

In the next example a third variable is introduced but the assignment list only provides two elements. This will result in `number1` and `number2` being set but `number3` remaining without a value (`null`):

Multiple assignment

```
def number1, number2, number3
(number1, number2, number3) = [10, 20]

assert number1 == 10
assert number2 == 20
assert number3 == null
```

Finally, we can perform multiple assignment at the point of declaring the variables:

Multiple assignment

```
def (number1, number2, number3) = [10, 20, 30]

assert number1 == 10
assert number2 == 20
assert number3 == 30
```

Variable names

Variable names must meet the following criteria:

- Must start with a letter (upper-case [A-Z] or lower-case [a-z]) - The underscore (_) is also allowed but this is very strongly discouraged
- Must only contain letters, digits (0-9) or an underscore (_)
  - The dollar-sign ($) is also allowed but very strongly discouraged
- Must not match a keyword (reserved word)

The use of literate variable names that comply to the criteria is encouraged. For example, a variable named `x` provides little information as to its role whereas `accountNumber` is likely to be clear within the context of a broader system.

Data Types

Data types define the sort of data a variable can hold. Most programming language feature the following data types:

- Booleans
  - A logical value of `true` or `false`
- Characters and strings
  - A character is a single letter, number or symbol (e.g. `#`)
  - A piece of text is referred to as a “string”
- Numbers
  - Integers (whole numbers) both positive and negative
  - Decimals (fractional numbers) both positive and negative
- Dates and times
  - You know, like dates and times
- Lists and sets
  - A variable that holds a number of values (list)

    - A variable that holds unique values (set)
- Maps
    - A variable that holds a number of values, each referred to by a key
- Ranges
    - A numeric sequence between a start and an end value - e.g. `1` to `10`

Being an object-oriented programming language, Groovy lets you also define your own types of objects (called classes).

Groovy allows you to create and use variables without declaring a data type - often called *dynamic typing*. Java, on the other hand, uses *static typing* and you need to tell Java the data type you want to use when declaring a variable. Once again, Groovy is flexible and lets you use dynamic or static typing (or both) in your programs.

13. Objects

Groovy is an object-oriented programming language and it's essential to understand what this means if you're to really get to grips with coding in Groovy.

But what is an object? Well, an object is an encapsulation of properties and methods:

- **Properties** are variables that hold data about the object
  - For example, a person object may have properties such as:
    - `name`
    - `email`
  - Properties are sometimes referred to as *fields*.
- **Methods** are a means for accessing and manipulating the object's properties
  - For example a person object may have methods such as:
    - `getName()`
    - `setName(name)`
  - Methods can take parameters and/or return values. For example:
    - `getName()` would return the person's name
    - `setName(name)` takes 1 parameter (`name`) and sets the person's name to that value
  - Methods are sometimes called *functions*

We use the `class` structure to define this assembly of properties and methods.

This is just an overview

We'll cover a lot more on Objects in a later tutorial as they're so central to Groovy. For now, this overview should help you get an understanding of how objects are used.

Declaring and using a class

Let's look at a Groovy script that declares a new class:

Declaring a new class

```
class Person {
    def name
    def email

    def getName() {
        return this.name
    }

    def setName(name) {
        this.name = name
    }

    def getEmail() {
        return this.email
    }

    def setEmail(email) {
        this.email = email
    }
}

// Create a new variable to hold an instance of the Person class
def david = new Person(name: 'David', email: 'david@example.com')

// Change David's email address:
david.setEmail('dave@example.com')

// Print out David's information
println david.getName()
println david.getEmail()
```

A class is defined using the `class` keyword and it's best practice to using an uppercase letter for the first character: `class Person {`

Curly brackets

You'll note the `{` in the code. This is called a curly bracket (or brace) and is used to denote a block of code. Each `{` has a partner `}` and are called the opening and closing brackets respectively. The opening bracket after `class Person` tells Groovy that all of the code up to the closing bracket relates to our definition of the Person class.

We declare the two properties in much the same way as we do for any variable:

Properties

```
def name
def email
```

A number of methods are declared to let us set and retrieve (get) the values of the object's properties:

Methods

```
def getName() {
    return this.name
}

def setName(name) {
    this.name = name
}

def getEmail() {
    return this.email
}

def setEmail(email) {
    this.email = email
}
```

More curly brackets

Again you'll note the opening and closing brackets for each method, telling Groovy where the method definition opens and closes. As these are *nested* within the brackets for the `class` we know the methods belong to the `class`

After we've declared the `Person` class we can now create instances of the class and assign values to the properties:

Creating an instance

```
def david = new Person(name: 'David', email: 'david@example.com')
```

We use `def david` as we would for other variables and then use `new Person` to indicated that `david` will hold an instance of the `Person` class. Lastly we call a special method called a *constructor* that

Groovy provides us for our objects: `(name: 'David', email: 'david@example.com')`. This sets up `david` with starting values for the properties.

What is `this`?

You may have noticed in my class declaration that I've used the `this` keyword in methods (e.g. `this.email = email`) - this denotes properties and methods related to the instance of the class. Using `this` in our class's methods lets us denote that the variable (or method) being called is a property or method of the current instance. It helps us not get mixed up with method parameters with the same name.

Constructors

Constructors are basically methods but use the class name for the method name. They're also only called when you create a new instance.

At some point David changes his email address so we call the `setEmail` method:

```
david.setEmail('dave@example.com')`.
```

You can see that the method call uses dot-point notation of `<variable name>.<method name>` - the dot (`.`) separates the variable name (`david`) from the method (`setEmail`).

Lastly, we use the two `get` methods to display `david`'s information:

Calling methods

```
println david.getName()
println david.getEmail()
```

The example `Person` class has demonstrated a number of Groovy's object-oriented programming syntax:

1. Creating a new class with properties and methods
2. Creating a new instance of the class and calling its constructor
3. Changing (setting) and retrieving (getting) the instance's properties

You can create lots of `Person` instances and each will exist in their own context. This means that `david` and `sarah` don't get mixed up:

Creating instances

```
def david = new Person(name: 'David', email: 'david@example.com')
def sarah = new Person(name: 'Sarah', email: 'sarah@example.com')
```

Useful Methods

In the Groovy/Java family tree, `java.lang.Object` is the grand-daddy of all classes. Using a system called "inheritance", each new class inherits attributes such as methods and properties from their forebears. Even the `Person` class I described above inherits from `java.lang.Object` and the Groovy developers have enhanced that class further! This means that all classes have built-in features that we can access. Let's look at a few of them.

`class`

The `class` property is used to access the Class that defines the object. This can be really useful when we want to check what sort of object we're dealing with.

The class property

```
class Person {
    def name
    def email
}

def david = new Person(name: 'David', email: 'david@example.com')

println david.class.name
```

`dump()`

This will return a String that describes the object instance's internals. Try out the following code to see what gets dumped:

The dump method

```
class Person {
    def name
    def email
}

def david = new Person(name: 'David', email: 'david@example.com')

println david.dump()
```

`with()`

This method works with closures (we'll cover them later) to give us an easy format for accessing a object's properties in methods. In the example below I wrap some code using `with` and don't have to use `david.name` and `david.email` to access those properties:

The with method

```
david.with {
    println name
    println email
}
```

Existing classes

The great strength/benefit/bonus of an object-oriented programming is the vast array of existing libraries of objects that you can reuse in your code. In Groovy and Java the listing of these available objects are referred to as the Application Programming Interface (API).

If we were going to create a variable to hold a string (a piece of text) we would do something like:

Creating a new String

```
def quote = 'Well may we say "God save the Queen", because nothing will save the\
 Governor-General!'
```

We could also use the following code to do exactly the same thing as the code above:

Also creating a new String

```
def quote = new String('Well may we say "God save the Queen", because nothing wi\
ll save the Governor-General!')
```

This looks similar to the way we created an instance of the `Person` class - we create a new instance of `String` and pass the text into the constructor.

Usage

I prefer the first version as it's a lot easier to read but the example provided here lets you see that under Groovy's hood there's some helpful stuff going on to make your life easier.

Now that we have our `quote` string we actually also get a number of methods that help us handle our variable:

Handy String methods

```
//Display the quote in upper case letters
println quote.toUpperCase()

//Display the quote backwards
println quote.reverse()

//Display the number of characters in the quote
println quote.size()
```

The example above demonstrates how we can call methods on an object instance and you'll see this used in the rest of the tutorials. Be sure to try out the code above to see what it does!

Classes and literal values

Literal values are best thought of the value you would write down:

- Boolean:
  - true
  - false
- Numbers:
  - 42
  - 3.14

- Strings (text):
  - ‘hi there’

We use single- or double-quotes for string literals otherwise Groovy thinks the text is actually code and tries to interpret it.

We can call methods directly on literal values as Groovy will create an appropriate object instance for us:

Calling a method from a literal

```
assert 1.plus(1) == 2
```

This definitely looks a bit odd but think of it this way:

1. Groovy sees the literal value `1` followed by a method call
2. Groovy creates a number object instance for `1`
3. Groovy then calls the `plus` method against the new number instance

`1.plus(1)` also looks odd because we’re used to seeing `1 + 1`. Both formats are supported but the latter is easier to read.

This can start to be very useful when you look at lists and ranges - something we’ll get to soon.

Lastly, as the literal is put into an object we can access methods and properties for the object. In the example below I can see what data type Groovy is actually using when I use 3.14:

Accessing properties from a literal

```
println 3.14.class.name
```

14. Booleans

Booleans help you get to the truth of the matter.

Boolean variables are perhaps the simplest and can hold a value of either `true` or `false`.

Booleans

```
def truth = true
def lies = false
```

Useful Methods

Booleans have a small number of methods that you generally won't find yourself using as they (mostly) have equivalent operators that are more "natural" to read and write.

The `and(right)` method performs a logical 'and'

The and method

```
def truth = true
def lies = false
assert truth.and(lies) == false
```

The conditional And operator (`&&`) is equivalent to the `and` method and the assertion above could also be written as `assert truth && lies == false`

The `or(right)` method performs a logical 'or'

The or method

```
def truth = true
def lies = false
assert truth.or(lies) == true
```

The conditional Or operator (`||`) is equivalent to the `or` method and the assertion above could also be written as `assert truth || lies == true`

15. Numbers

Groovy supports the various types of numbers that you'd expect - and makes it easy to use them. We'll take a quick look at those now but the chapter on data types will really get into the depths of messing around with numbers.

There are two main types of numbers you're likely to need:

- Integers (whole numbers)
- Decimals

Groovy also gives us scientific notation and other number systems and we'll take a look at how you use them.

Integers

Integers are whole numbers and can be negative or positive:

Using Integers

```
def age = 27
def coldDay = -8
```

Groovy will also handle very large numbers:

Large numbers

```
// 1 astronomical unit (au)
def distanceEarthToSun = 149597870700
def distanceNeptuneToSun = distanceEarthToSun * 30
```

Decimals

Decimal numbers provide a fraction and can be negative or positive:

Using decimals

```
def pi = 3.14159

// Measured in celsius
def absoluteZero = -273.15
```

Scientific notation

Base-10 (decimal) scientific notation ($a * 10^b$) can also be used by placing an `e` or `E` before the exponent:

Using SN

```
def atomicMass = 1.67e-27
```

The next example sets the `au` variable to 1.49597870700 * 10^11 and then checks to make sure I haven't messed up the exponent:

Just a check

```
def au = 1.49597870700e11
assert au == 149597870700
```

In the previous two examples you can see a signed (positive or negative) integer as the exponent:

- `e-27` is negatively signed
- `e11` can also be written as `e+11` and is positively signed

This tutorial will give you an overview that will cover most types of numbers but if you expect to be handling very large or very small numbers and calculations with such numbers then you really need to do some research to make sure you don't become a victim of truncation (where parts of the number are chopped off) and other issues with precision.

Number Systems

Most of the time we deal with decimal (base-10) numbers but there are other number systems out there. If we want to use the number 15 in base-10 we just type `15` but we can also use:

- Binary (base-2) by prefixing `0b`
  - That's a zero followed by lower-case "b"
- Octal (base-8) by prefixing `0`
  - That's just zero
- Hexadecimal (base-16) by prefixing `0x`
  - That's a zero followed by lower-case "x"

The code below illustrates the many faces of the number 15 (base-10):

Different number systems

```
println 0b1111    //Binary
println 15         //Decimal
println 017       //Octal
println 0xf       //Hexadecimal
```

To help you deal with long numbers Groovy lets you use underscores (_) to visually break up the number without changing its value:

Formatting large numbers

```
assert 1_000_000 == 1000000
assert 0b0001_0110_1101 == 365
```

Let's close with a joke:

Lolz

```
def value = 0b10

println "There are only $value types of people in the world - those who know bin\
ary and those who don't"
```

Useful Methods and Properties

Groovy (Java) numbers trace their lineage (inherit) back to `java.lang.Number`. The `Number` class provides methods to covert between different types of numbers (integer, decimal etc) - we'll cover this in the chapter on Data Types.

Most numerical classes (e.g. `Integer`) provide the handy `max` and `min` methods that let you compare two numbers of the same numerical type:

max and min

```
assert Integer.max(10, 2) == 10
assert Integer.min(10, 2) == 2
```

16. Strings

ℹ Strings are pieces of text.

There are two main ways in which you can declare a string in Groovy: single or double quotes

The String section

Method	Usage
Single quotes (`'...'`)	These are fixed strings and tell Groovy that the string is as we've written it (e.g. `def pet = 'dog'`).
Double quotes (`"..."`)	These are called GStrings and let us interpolate (insert) variables into our string. (e.g. `def petDescription = "My pet is a $pet"`)
Three single quotes (`'''...'''`)	A multi-line fixed string
Three double quotes (`"""..."""`)	A multi-line GString

Here's a quick example of a fixed string and a GString in action:

Fixed strings and GStrings

```
def pet = 'dog'
def petDescription = "My pet is a $pet"
println petDescription
```

Escape sequences

Strings can contain escape sequences, allowing you to use non-printable characters in your text.

Escape sequences

Sequence	Character
\n	line feed
\f	form feed
\r	carriage return
\t	horizontal tab
\’	single quote
\”	double quote
\\	backslash

The line feed (`\n`) is often used to move to a new line:

The line feed

```
print 'Hi \n there\n'
```

You’ll notice the use of `print` in the example above - the final `\\n` performs the same as `println` and moves to a new line.

The form feed (`\\f`) and carriage return (`\\r`) aren’t often used. Form feed indicates a new page and carriage return goes back to the start of the line.

The horizontal tab (`\\t`) is essentially the same as the `tab` key on your keyboard. It’s useful for formatting things like tables of information:

Sequences

```
println 'name\tage\tcolour'
println 'Sam\t12\tblue'
println 'Alice\t8\tgreen'
```

If you wish to use a quote within your string that matches the quote type you’re using to surround your string then you need to escape the internal quote using the \ character. In the code below you can see the quotes being escaped (`\'` and `\"`):

Escape, escape!

```
println 'That\'s mine'
println "I said \"NO!\""
```

As the backslash (\) is used to escape characters, it needs an escape of its own. In order to use a backslash in a string you need to double it up (\\) as in the example below:

Backslashing

```
println 'c:\\documents\\report.doc'
```

GStrings

In order to have Groovy interpolate the value of a variable we use the `$` symbol in front of the variable name - as you can see with `$pet` below:

GStrings

```
def pet = 'dog'
println "I own a $pet"
```

This can be handy if you have a number of variables that you'd like to use in a string:

Interpolating strings

```
def name = 'Sally'
def hobby = 'surfing'
println "Did you know that $name likes $hobby?"
```

Avoid + for String concatenation

This helps escape the use of the addition operator (+) to concatenate (join) strings: `println 'hello ' + 'world'` You'll see this in a lot of Java code and I, for one, am pleased to see that Groovy lets us pretend this never happened.

GStrings also let you interpolate more complicated expressions into a string by using `${...}`. In the following example we perform a calculation within the GString:

Operation in a GString

```
println "10 to the power of 6 is ${10**6}"
```

We can also access information about a variable in the same manner:

Operation in a GString

```
def word = 'Supercalifragilisticexpialidocious'
println "$word has ${word.length()} letters"
```

The code `word.length()` calls the `length` method available on a string - we'll cover what this means shortly.

Multiline Strings

The examples given so far use short strings but longer strings would be cumbersome to type using lots of new lines. Instead, Groovy provides multiline strings - the code below declares a multiline fixed string:

A Multiline string

```
def poem = '''But the man from Snowy River let the pony have his head,
And he swung his stockwhip round and gave a cheer,
And he raced him down the mountain like a torrent down its bed,
While the others stood and watched in very fear.'''

print poem
```

If you run the code above you'll see that new lines are used at the correct points in the display but the first line is not quite right. You can modify this slightly and place a backslash (\) at the start of the string - using statement continuation for readability:

Fixing the first line

```
def poem = '''\
But the man from Snowy River let the pony have his head,
And he swung his stockwhip round and gave a cheer,
And he raced him down the mountain like a torrent down its bed,
While the others stood and watched in very fear.'''

print poem
```

Use the backslash

Without the backslash the code above would cause a blank newline to be printed.

GStrings can also be defined using the multiline format:

A multiline GString

```
def animal = 'velociraptor'

println """But the man from Snowy River let the ${animal} have his head,
And he swung his stockwhip round and gave a cheer,
And he raced him down the mountain like a torrent down its bed,
While the others stood and watched in very fear."""
```

Building Strings

Working with basic strings is fine but if you need to build up a large piece of text throughout a program they can become very inefficient. We'll look into this in the tutorial on Operators.

Useful Methods

Strings (text) are important aspects to human-based systems so most programming languages provide a number of methods for modifying, search, slicing and dicing strings. Groovy provides a number of helpful methods you can use with strings and we'll look at just a few of them here:

- `length()` : returns the number of characters in a string
- `reverse()`: returns the mirrored version of the string
- `toUpperCase()` and `toLowerCase()`: returns the string with all of the characters converted to upper or lower case.

Some String methods

```
def str = 'Hello, World'
println str.length()
println str.reverse()
println str.toUpperCase()
println str.toLowerCase()
```

The `trim()` method returns the string with any leading and trailing whitespace removed:

Trimming a String

```
def str = '   Hello, World   '
println str.trim()
```

The `substring` method returns a subsection of a string and can be used in two possible ways:

- Provide a start index (e.g. `substring(7)`) to get the subsection that includes that index (i.e. the 7th character in the string) through to the end of the string
- Provide a start and an end index (e.g. `substring(7, 9)`) to get the subsection that includes that start index through to the end index of the string

Substrings

```
def str = 'Hello, World'
println str.substring(7)
println str.substring(7,9)
```

A number of methods are provided to help you with basic searching:

- The `indexOf` and `lastIndexOf` methods return the index (location) of the specified character in the string
- `contains`, `startsWith`, and `endsWith` return `true` or `false` if the supplied parameter is located within the string

Basic searching

```
def str = 'Hello, World'

//These methods return the index of the requested character
println str.indexOf(',')
println str.lastIndexOf('o')

//These methods check if the string contains another string
println str.contains('World')
println str.startsWith('He')
println str.endsWith('rld')
```

Lastly, `replace` lets us provide a string that we want to change to a new value:

Replacing text

```
def str = 'Hello, World'

println str.replace('World', 'Fred')
```

Regular expressions provide a comprehensive approach to searching and manipulating strings and are covered in an up-coming chapter. Additionally, the tutorial on Operators will look into this in more depth.

17. Collections

Having a single number or string is useful but collections help you keep them together.

Collections group a number of values in a single container. The Java Collections Framework[1] provides a really extensible and unified approach to handling collections. Groovy makes these even easier to use and focusses on two key collection types:

- Lists: provide a container for several values
- Maps: use keys as a method for indexing a set of values

Lists

List variables contain several items and are declared using square brackets (`[...]`).

The example below declares a variable (`temperatures`) as an empty list:

Declaring an empty list

```
def temperatures = []
```

The next examples declares the `temperatures` list with some initial values:

Declaring a list with values

```
def temperatures = [10, 5, 8, 3, 6]
```

In the temperatures example the list contains just numbers but Groovy lists can contain a mix of data types:

[1] http://docs.oracle.com/javase/tutorial/collections/intro/index.html

Lists can contain mixed types

```
def mixed = [1, true, 'rabbit', 3.14]
println mixed[2]
println mixed[-3]
println mixed.get(3)
```

The square brackets `[]` are used to create a list but are also used to refer to indexes in the list (e.g. `mixed[2]`) - this is often referred to as *subscript notation*. In the example above you'll notice I've printed out `mixed[2]` - the list item with index (subscript) `2`. Somewhat confusingly this causes `rabbit` to be displayed. This is because lists are zero-based and the first item is at index `0`, not index `1`. Where we use `mixed[2]` we're asking for the third item in the list.

It may surprise some programmers that `println mixed[-3]` is valid - it's a very handy approach to accessing list items from the end of the list. Item `-1` is the last in the list so `mixed[-3]` will be the value `true`.

The `get()` method can also be used to access a list element by its index - e.g. `mixed.get(3)` gives us `3.14`.

I can provide multiple indexes in the subscript notation and grab the specified elements from the list. In the example below I grab elements `0` and `2` (`temperatures[0, 2]`) and then elements `1`, `3` and `4` (`temperatures[1, 3, 4]`):

Using indexes with lists

```
def temperatures = [10, 5, 8, 3, 6]
assert temperatures[0, 2] == [10, 8]
assert temperatures[1, 3, 4] == [5, 3, 6]
assert temperatures[1..3] == [5, 8, 3]
```

Ranges can also be used in the subscript notation and, as demonstrated in the example below, return a list containing the items who's indexes are included in the range:

Using ranges with lists

```
def temperatures = [10, 5, 8, 3, 6]
assert temperatures[1..3] == [5, 8, 3]
```

We can also use a mix of individual indexes and ranges as we see fit:

Indexes and ranges with lists

```
def temperatures = [10, 5, 8, 3, 6]
assert temperatures[0..1, 3] == [10, 5, 3]
assert temperatures[0..1, 1..3] == [10, 5, 5, 8, 3]
assert temperatures[0..1, 1..3, 4] == [10, 5, 5, 8, 3, 6]
```

What? Let's take a look:

- `temperatures[0..1, 3]` returns a list containing the elements of `temperatures` with the indexes 0, 1 and 3
- `temperatures[0..1, 1..3]` returns a list using two ranges to select the indexes. As index item `1` is requested twice, the returned list features the item (5) twice.
- `temperatures[0..1, 1..3, 4]` does the same as the previous statement but adds in the item at index `4`

Adding elements

To add an element to a list we use the `add()` method or the `<<` operator:

Adding elements

```
def mixed = [1, true, 'rabbit', 3.14]
mixed << 'biscuit'
mixed.add(101)
println mixed
```

Sets

Sets are much like lists but each element in a set is unique:

Declaring a Set

```
def names = ['sally', 'bob', 'sally', 'jane'] as Set
println names
```

If you try the code above you'll get `[sally, bob, jane]` - the set just ignores the repeated element.

The as keyword

The `as` keyword is an operator used to cast a variable to another type. In the example above we're casting to the `Set` class but you can also cast to other collection types - something to look forward to when you get into more advanced coding.

Useful List Methods

The `size()` method returns the number of elements in the list:

List size

```
def periodic = ['hydrogen', 'helium', 'lithium']
println periodic.size()
```

The `first()` and `last()` methods return the first and last elements in a list. The `head()` method is synonymous with `first()`.

First and last methods

```
def periodic = ['hydrogen', 'helium', 'lithium']
println periodic.first()
println periodic.head()
println periodic.last()
```

The `tail()` method returns the list minus the first (head) element and the `init()` method returns the list minus the last element:

Tail method

```
def periodic = ['hydrogen', 'helium', 'lithium']
assert periodic.tail() == ['helium', 'lithium']
assert periodic.init() == ['hydrogen', 'helium']
```

The `contains()` method returns `true` if the requested element is contained in the list:

Contains method

```
def periodic = ['hydrogen', 'helium', 'lithium']
assert periodic.contains('helium') == true
```

The `reverse()` method returns the mirror of the list:

Reverse method

```
def periodic = ['hydrogen', 'helium', 'lithium']
println periodic.reverse()
```

The `sort()` will sort the elements in a "natural" order. Basically, this relies on the list elements being comparable in some manner. The `sort` method is best used when the list contents are all of the same type (e.g. strings or numbers):

Sort method

```
def periodic = ['hydrogen', 'helium', 'lithium']
periodic.sort()
```

The `asImmutable()` method is a handy way to set the list contents in stone - "Immutable" essentially means "unchanging".

Don't go changing

```
def friends = ['fred', 'sally', 'akbar'].asImmutable()

//This next line will cause an exception:
friends << 'jake'
```

Maps

Maps allow us to build up a type of look-up table using keys and values. Other languages call these dictionaries or associated arrays.

An empty map is declared using `[:]` and the example below shows this in use when declaring the `periodic` variable.

Declaring an empty map

```
def periodic = [:]
```

Each key in a map is unique and points to a value in the map. In the example below we see the start of a basic periodic table by declaring a variable (`periodic`) with a set of key-value pairs (`key: value`) each separated by a comma (`,`) and held within square brackets (`[...]`):

Declaring a map with elements

```
def periodic = ['h': 'hydrogen',
                'he': 'helium',
                'li': 'lithium']

println periodic['li']
println periodic.li
println periodic.get('li')
```

You should also note that we can access map items using:

1. The key in square brackets (`[]`)
   1. Much as we did with lists: `println periodic['li']`.
   2. This is often referred to as *subscript notation*.
2. We can also use the period (`.`) followed by the key:
   1. As in `println periodic.li`.
   2. This is often referred to as *dot-point notation*
3. Lastly, the `get()` method is passed a key and returns the associated value

Dot point or key access?

I prefer the dot-point notation but sometimes you need to use square brackets if you're using a key that's a reserved word or not a valid name. The `get()` method is OK but is a little less clean in terms of aesthetics.

The keys in a map can be names provided they adhere to the same rules we follow for variable names. That means that the keys in `periodic` don't have to be written as strings:

Keys as names

```
def periodic = [h: 'hydrogen',
        he: 'helium',
        li: 'lithium']
```

Adding elements

To add an element to a map we can use the square bracket, dot-point notation, `<<` operator, or `put()` method to add on a new key/value pair:

Adding elements

```
def periodic = ['h': 'hydrogen',
        'he': 'helium',
        'li': 'lithium']

periodic['be'] = 'Beryllium'
periodic.b = 'Boron'
periodic << ['c': 'Carbon']
periodic.put('n', 'Nitrogen')

println periodic
```

Keys

Map keys don't have to be strings - they can be a mix of strings, numbers or other objects. Let's look at an example then go through the various bits of code:

Different types of keys

```
class Chicken {
        def name
            String toString() {
                       return "I am $name".toString()
            }
}

def cluckers = new Chicken(name: 'Cluckers')

def mixedMap = [
    12:         'Eggs in a carton',
    'chicken':  'Egg producer',
    (cluckers):   'Head chicken'
]

println mixedMap[12]
println mixedMap.get(12)

println mixedMap.chicken
println mixedMap['chicken']
println mixedMap.get('chicken')

println mixedMap[(cluckers)]
println mixedMap.get(cluckers)

println mixedMap
```

In the example above:

1. I create a new class (`Chicken`)
    1. … and store a new instance of `Chicken` in the variable `cluckers`
2. I then create a map variable called `mixedMap` with different types of keys:
    1. `12` is a number
    2. `'chicken'` is a string
    3. `(cluckers)` indicates that the key is a variable value
3. I use the square-bracket notation and `get` method to access the value aligned to the key `12`
    1. `mixedMap.12` won't work
4. I use the square-bracket, dot-point and `get` method to access the value aligned to the key `'chicken'`

5. I use the square-bracket notation and `get` method to access the value aligned to the key `(cluckers)`
   1. `mixedMap.cluckers`
6. `println mixedMap` is called to display the map contents

I'd suggest you stick with strings as keys for now. I'd also suggest that using one type for your keys will usually make life a lot easier.

For those interested in such things, the `(cluckers)` key isn't affected if I change the value of `cluckers` later on. If you append the code below to the chicken example you'll see that `mixedMap.get(cluckers)` will now return `null` as the match fails. You'll also notice that `println mixedMap` is the same output you get before changing `cluckers`:

Changing objects used as keys

```
cluckers = new Chicken(name: 'Bill')
println mixedMap.get(cluckers)
println mixedMap
```

Useful Map Methods

As with lists, the `size()` methods returns the number of elements in a map.

The `get` method can be used to get the value for the requested key. A second optional parameter can be provided and is returned if the map does not contain the requested key:

Get method

```
def periodic = ['h': 'hydrogen',
        'he': 'helium',
        'li': 'lithium']

println periodic.get('he')
println periodic.get('he', 'Unknown element')
println periodic.get('x', 'Unknown element')
```

The `keySet()` method returns a list containing all of the keys in a map and `values()` returns a list of the values in a map:

keySet method

```
def periodic = ['h': 'hydrogen',
        'he': 'helium',
        'li': 'lithium']

println periodic.keySet()
println periodic.values()
```

The `containsKey()` and `containsValue()` methods are useful for checking on map contents:

Checking for keys and values

```
def periodic = ['h': 'hydrogen',
        'he': 'helium',
        'li': 'lithium']

println periodic.containsKey('li')
println periodic.containsValue('carbon')
```

The `asImmutable()` method works for maps in the same manner as it does for lists:

Don't go changing

```
def periodic = ['h': 'hydrogen',
        'he': 'helium',
        'li': 'lithium'].asImmutable()

//This will cause an exception:
periodic.x = 'new element'
```

18. Arrays

I'd use collections rather than arrays but you should probably know about arrays.

For my money, the collections we've just looked at (lists, sets, maps) are more versatile than arrays and collections are my preferred approach. However, there's a lot of code out there using arrays so let's take a quick look.

Arrays contain a fixed number of elements of a specified data type. Let's look at an example of array declaration and usage:

Declaring an array

```
Number[] point = new Number[2]

point[0] = 27
point[1] = -153

assert point.length == 2
```

So let's dissect that chunk of code:

- The `point` variable is declared using `Number[] point = new Number[2]`
  - `Number[]` indicates that we want an array of Numbers
    * `[]` indicates that the variable is an array, not just a single Number value
    * We don't use `def` as we're specifying the data type
  - `new Number[2]` sets `point` to be an empty array that can contain two (2) elements of the `Number` class (or a subtype thereof).
- Arrays are zero-based, meaning that the first element is at index `0`
  - `point[0]` is the first element
  - `point[1]` is the second
- `point.length` returns the number of elements in the array
  - Note that the range of indexes for an array is `0..(point.length - 1)`

`point.size()` would also work and provides the same result as `point.length`

If I'd tried something like `point[2] = 99` I would get a `java.lang.ArrayIndexOutOfBoundsException` as the array can only hold 2 elements.

It's important to note that the `size` of an array is fixed at declaration. If you decide that you need to expand the array then you'll slap your forehead and ask "Why didn't I use collections?". If you dig your heels in and stay with arrays you might check out the `java.lang.System.arraycopy` method and learn the gentle art of copying and resizing arrays. Then, you'll start using collections.

We can be more direct in creating the array and provide the values up-front. In the example below I create an array that can hold two elements and I load the values into the array:

Setting elements at creation

```
Number[] point = [27, -153]
```

So, why did I pick `Number`? Well, I want an array of numerical values but perhaps wasn't sure which *type* of numbers. Provided the values I put into the array are subtypes of `Number`, all will be well. That means the following will work fine and nothing will be truncated:

```
Number[] point = [27.9, -153]
```

If I really wanted to be specific about the type of number I could have declared `point` as an array of `Integer` values:

```
Integer[] point = [27, -153]
```

Arrays can also be declared to be of a primitive type such as `int`[1]:

```
int[] point = [27, -153]
```

Going further with subtypes etc, arrays can be of any type and the `Object` class provides a flexible type when your array needs to hold a mixture of values (e.g. numbers, strings, various types):

[1]Primitive types are discussed in the Data Types chapter.

A mixed bag array

```
Object[] bag = new Object[4]
bag[0] = true
bag[1] = 'Rabbit'
bag[2] = 3.14
bag[3] = null
```

Without wanting to be repetitive, the example above would probably be easier to work with if we used a collection such as a list.

Manipulating arrays

We've seen the `size()` method and `length` property - both indicating how many elements the array can hold.

Sorting an array is easy with the `sort()` method:

Sorting an array

```
Number[] nums = [99, 10.2, -7, 99.1]
nums.sort()
println nums
```

Sort changes the array

Note that the `sort()` modifies the `nums` array.

Of course `sort()` works well if the element types have a meaningful sorting order but try out the following code and you'll see that the `sort()` perhaps isn't overly useful on mixed values:

Can this be sorted?

```
Object[] bag = new Object[4]

bag[0] = true
bag[1] = 'Rabbit'
bag[2] = 3.14
bag[3] = null

println bag.sort()
```

Use the `Arrays.asList()` static method to get a copy of an array into a list (collection):

Arrays to lists with `asList`

```
Number[] nums = [99, 10.2, -7, 99.1]
def list = Arrays.asList(nums)
```

Alternatively, you can use the `as` operator to cast the array to a List.

Arrays to lists with `as`

```
Number[] nums = [99, 10.2, -7, 99.1]
def list = nums as List
```

Check out the `java.util.Arrays` class for more array methods.

19. Ranges

ℹ Ranges describe a range of numbers.

Ranges define a starting point and an end point. Let's look at a well-known type of range:

Declaring a range

```
def countdown = 10..0

println countdown.getFrom()
println countdown.getTo()
```

The `countdown` range starts at `10` and goes down to `0`. The notation should be easy to decipher: `<start>..<end>`.

Printing out a range variable will display that a range is rather like a list of values - in the case of `countdown` they're numbers:

Ready for launch

```
def countdown = 10..0
println countdown
```

Whilst my examples so far all go down, you can just as easily have a range that goes up:

Going up

```
def floors = 1..10
println floors
```

You can also use decimals but note that it is only the integer (whole-number) component that is stepped through:

Decimals in ranges

```
def countdown = 10.1..1.1
println countdown
```

Half-Open Ranges

Ranges aren't just limited to inclusive ranges such as `1..10`. You can also declare a *half-open range* using `..<` - that's two periods and a less-than. This denotes that the range ends prior to the number to the right. In the example below I setup a grading criteria that avoids an overlap between the grades:

Half-open range declarations

```
def gradeA = 90..100
def gradeB = 80..<90
def gradeC = 65..<80
def gradeD = 50..<65
def gradeF = 0..<50
```

I could tweak the above code if I want to get fancy:

A fancier approach

```
def gradeA = 90..100
def gradeB = 80..<gradeA.getFrom()
def gradeC = 65..<gradeB.getFrom()
def gradeD = 50..<gradeC.getFrom()
def gradeF = 0..<gradeD.getFrom()
```

Ranges of Objects

Ranges are primarily used with numbers but they can be of any object type that can be iterated through. This basically means that Groovy needs to know what object comes next in the range - these objects provide a `next` and `previous` method to determine this sequence. Over time you'll discover various options for use in ranges but numbers really are the main type.

Apart from numbers, individual characters (letters) can be used in ranges. In the example below I create a range of lower-case letters:

A range of characters

```
def alphabet = 'a'..'z'
println alphabet
```

Ranges and Enums

We'll look into Enums when we start looking at creating objects in a later tutorial

Ranges can be handy when dealing with `enums` as they give us the ability to set a subset of enum values. In the example below I create a handy helpdesk tool:

1. Setup an `enum` listing the possible ticket priorities
2. Create a new `class` to describe helpdesk tickets
3. Setup a `helpdeskQueue` containing a list of tickets
4. Set the `focus` variable as a range of `Priority` values
5. Go through the list of tickets and pick up any that are set to the `priority` I care about.

Using a helpdesk ticket `enum`

```
enum Priority {
    LOW,MEDIUM,HIGH,URGENT
}

class Ticket {
    def priority
    def title
}

def helpdeskQueue = [
    new Ticket(priority: Priority.HIGH, title: 'My laptop is on fire'),
    new Ticket(priority: Priority.LOW, title: 'Where is the any key'),
    new Ticket(priority: Priority.URGENT, title: 'I am the CEO and I need a coff\
ee'),
    new Ticket(priority: Priority.MEDIUM, title: 'I forgot my password')
]

def focus = Priority.HIGH..Priority.URGENT
```

```
for (ticket in helpdeskQueue) {
    if (ticket.priority in focus) {
        println "You need to see to: ${ticket.title}"
    }
}
```

Try the example above out with various settings for the `focus` variable:

- `def focus = Priority.MEDIUM..Priority.URGENT`
  - Gives us more tickets to see to :(
- `def focus = Priority.HIGH..Priority.LOW`
  - Is actually similar to `4..1` and leaves out the tickets marked `URGENT`

Ranges and List Indexes

You can access a subset of a list using a range subscript. In the example below I use the subscript `[1..3]` to grab a new list containing elements 1 through 3 of the `temperatures` list.

Zero-based lists

Remember that lists are zero-based so `5` is element number 1

Accessing range elements

```
def temperatures = [10, 5, 8, 3, 6]
def subTemp = temperatures[1..3]
assert subTemp == [5, 8, 3]
```

Ranges and Loops

Ranges are most often see when we're using loops - we'll get to them in a later tutorial but here's an example of a launch sequence:

Looping with ranges

```
def countdown = 10..0

for (i in countdown) {
    println "T minus $i and counting"
}
```

In the above example I store the range in the `countdown` variable in case I need it again later. If I don't really need to re-use the range I can put the range's literal value directly into the loop:

Looping with ranges refined

```
for (i in 10..1) {
    println "T minus $i and counting"
}
```

Useful Methods

We can use the `size()` method to find out how many elements are in the range:

The size method

```
def dalmations = 1..101
println dalmations.size()
```

As seen earlier, the `getFrom()` and `getTo()` methods return the start and final values respectively:

The range's start and end values

```
def intRange = 1..10
println intRange.getFrom()
println intRange.getTo()
```

The `isReverse()` method returns `true` if a range iterates downwards (backwards):

Checking for reverse

```
def countdown = 10..0
assert countdown.isReverse() == true
```

You can can use the `reverse()` method to flip the range:

Reversing the range

```
def floors = 1..10
println floors.reverse()
```

In order to check if a value is contained within a range we use the `containsWithinBounds` method and pass it the value we're checking on:

Checking bounds

```
def countdown = 10..0
assert countdown.containsWithinBounds(5) == true
```

The `step` method returns a list based going through the range via the specified increment (step). In the example below I step through the range one at a time (`step(1)`) and then two at a time (`step(2)`):

Stepping

```
def countdown = 5..1
assert countdown.step(1) == [5, 4, 3, 2, 1]
assert countdown.step(2) == [5, 3, 1]
```

As `step` returns a list I can use it to populate a list variable that has too many numbers for me to be bothered typing out:

```
def dalmations = (1..101).step(1)
println dalmations
```

As we're about to see the `step` method is very effective when used with closures.

Ranges and Closures

Closures are a method (function) that can be handled in a manner similar to variables. A closure is described within curly brackets `{..}` and can be passed as method parameters. Closure have a default variable named `it` and this holds a value passed to the closure by its caller.

We'll look into closures much more thoroughly in a later tutorial but, for now, take in the following examples and refer back to them when you get to know closures a little better.

The `step` method will call a closure for each item in a range. In the example below I step through `countdown` one number at a time and, for each number, I display a message:

Stepping through a range with closures

```
def countdown = 10..1
countdown.step(1) {
    println "T minus $it and counting"
}
```

Closure syntax

The syntax `countdown.step(1) {..}` probably looks a bit odd at this point - essentially, the closure is being passed as a parameter to `step`. There's a tutorial covering closures coming up so don't feel too annoyed if these examples don't look right to you.

I can use the range literal but need to place it within `(..)`:

Using the range literal

```
(10..1).step(1) {
    println "T minus $it and counting"
}
```

You can change the size of each step - in the case below I step down by `2` each time. Run the code and notice that launch never happens!

Changing the step

```
(10..1).step(2) {
    println "T minus $it and counting"
}
```

20. Regular Expressions

Regular expressions give us a powerful (and confusing) way of sifting through text.

Regular expressions (RegEx's) get entire books devoted to them and you'll find some developers are RegEx ninjas and others (like myself) are RegEx numpties. This chapter will introduce the basics but the Java Tutorial's Regular Expression trail[1] is a useful reference as is Wikipedia[2] for those seeking RegEx glory. There are also a number of online tools such as RegExr[3] that come in very handy when trying to debug that elusive RegEx pattern.

To define the regular expression pattern we use the `~/ /` syntax:

Declaring a regex

```
def regex = ~/\n/
```

Once stored as a variable, this regular expression can be used in a variety of ways. The example below sets up three string variables and tests them against the `regex` pattern by using the `matches` method - which returns `true` if the string matches the pattern:

Matching against a regex

```
def regex = ~/https?:\/\/.*/

def httpUrl = 'http://www.example.com/'
def httpsUrl = 'https://secure.example.com/'
def ftpUrl = 'ftp://ftp.example.com/'

assert httpUrl.matches(regex)
assert httpsUrl.matches(regex)
assert ! ftpUrl.matches(regex)
```

In the code above, `~/https?:\/\/.*/` is the regular expression pattern that's essentially looking for any string starting with `http` or `https`. The `s?` will match 0 or 1 occurrence of `s` in the pattern.

[1]http://docs.oracle.com/javase/tutorial/essential/regex/
[2]https://en.wikipedia.org/wiki/Regular_expression
[3]http://www.regexr.com

You'll notice the odd-looking `\/\/` - I need to escape the forward slashes in `http://` so that Groovy doesn't confuse them with the slashes used to define the regular expression pattern (∼`/../`).

We'll also look at the special operators for regular expressions in the next tutorial: Operators.

Underpinning Groovy's regular expression functionality is the Java class `java.util.regex.Pattern`[4]. Groovy handles the compiling of the pattern and this helps you focus on the struggle of getting the regular expression correct :)

Regular Expression Syntax

Regular expressions use a number of syntactic elements to define a pattern of text. We'll take a brief look at them here.

Characters

These elements are used to match specific literal characters.

Literal characters

Element	Matches
`g`	The character `g`
`\\`	The backslash character
`\t`	Tab character
`\n`	Newline character
`\f`	Formfeed character
`\r`	Carriage-return character

In the example below I take a section of a poem and use the `split` method to get a list whose elements contain a single line from the poem.

Splitting a poem

```
// The Ballad of the Drover by Henry Lawson
def poem = '''\
 Across the stony ridges,
  Across the rolling plain,
 Young Harry Dale, the drover,
  Comes riding home again.
 And well his stock-horse bears him,
  And light of heart is he,
 And stoutly his old pack-horse
```

[4] http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

```
  Is trotting by his knee.'''

def regex = ~/\n/

def lines = regex.split(poem)

def i = 1
for (line in lines) {
    println "Line $i: $line"
    i++
}
```

Character Classes

Character classes are used to define character sets and sequences.

Character classes

Element	Matches
`[xyz]`	`x`, `y` or `z`
`[^xyz]`	Not `x`, `y` or `z`
`[a-zA-Z]`	Range of characters (all letters)
`[0-9]`	Range of characters (all numbers)
`[a-zA-Z_0-9]`	Range of characters

Predefined Character Classes

The predefined character classes save you from having to define the class specifically and are handy for seeking out words and whitespace.

Predefined character classes

Element	Matches
`.`	Any character
`\d`	Digits `[0-9]`
`\D`	Non-digits
`\s`	Whitespace
`\S`	Not whitespace
`\w`	Word character `[a-zA-Z_0-9]`
`\W`	Not a word character

Boundaries

Boundaries, to state the obvious, mark the edge of something - specifically a line or a word.

Boundaries

Element	Matches
`^`	Start of a line
`$`	End of a line
`\b`	Word boundary
`\B`	Non-word boundary

Quantifiers

These determine how many matches are acceptable. For example `s?` matches the character `s` zero or one time - meaning that I expect that character to be an `s` or, if it's not, move to the next part of the pattern. `s+` means that I really want at least one `s` at that point.

Quantifiers

Element	Matches
`?`	Single match
`*`	Zero or more matches
`+`	One or more matches
`{n}?`	Exactly *n* matches
`{n, }?`	At least *n* matches
`{n,m}?`	At least *n* but not more that *m* matches

Useful Methods

A number of String methods can accept a regular expression and these are my preferred approach to checking text against regular expressions. Most of them take the pattern as the first parameter.

We saw the `matches()` method at the beginning of the chapter:

Matching

```
def regex = ~/https?:\/\/.*/
def httpUrl = 'http://www.example.com/'

assert httpUrl.matches(regex)
```

The `find()` method returns the first match against the pattern within the string. In the example below the `find()` will return the match against the port number in the URL:

Finding

```
def regex = ~/:[0-9]+/
def httpUrl = 'http://www.example.com:8080/'

println httpUrl.find(regex)
```

The `findAll()` method returns a list of matches for the pattern. In the example below I am returned all words in `speech` that start with `like`:

findAll

```
def speech = '''This like guy like I know but like don\'t really like
 was like so mean but likely to be nice when you know him better.'''

println speech.findAll(~/\blike\w*\b/)
```

> Like, wow!

The example below provides a very basic word counter by seeking out the `\b\w+\b` pattern and displaying the size of the list returned by `findAll`:

A word counter

```
def poem = '''\
 Across the stony ridges,
  Across the rolling plain,
 Young Harry Dale, the drover,
  Comes riding home again.'''

def regex = ~/\b\w+\b/

println poem.findAll(regex).size()
```

The `replaceFirst()` and `replaceAll()` methods seek out matches and replace them in a manner that their names implies:

Replacing

```
def speech = '''This like guy like I know but like don\'t really like
 was like so mean but likely to be a nice guy when you know him better.'''

println speech.replaceAll(~/\blike\b/, 'um')
println speech.replaceFirst(~/\bguy\b/, 'marmoset')
```

The `splitEachLine()` method is very handy when handling structured files such as comma-separated files. You can see in the example below that the first parameter is the pattern that will match commas (`~/,/`) and the second parameter is a closure that will do something for each line. Within the closure, the `it` variable is a list with each element being the delimited segment of the text with the line:

Splitting

```
def csv = '''\
Bill,555-1234,cats
Jane,555-7485,dogs
Indira,555-0021,birds'''

csv.splitEachLine(~/,/) {
    println "Name: ${it[0]}"
}
```

Pattern Methods

The `java.util.regex.Pattern` class provides a number of useful methods. I prefer to use the String methods but maybe I'm just lazy.

The static `matches` method is called against `Pattern` to evaluate a pattern against a piece of text. You'll note that the first parameter is the pattern but represented as a string so you drop the `~/../` notation:

Using `Pattern`

```
//Note the import
import java.util.regex.Pattern
assert Pattern.matches('https?://.*/', 'http://www.example.com/') == true
```

The `matcher()` method is called against a regular expression pattern and is passed the text that is to be checked. A `Matcher`[5] variable is returned and these give you a whole heap of regular expression functionality. In my example I just check for the match by calling `matches()`:

[5]http://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html

Using `Matcher`

```
def regex = ~/https?:\/\/.*/
def httpUrl = 'http://www.example.com/'
def matcher = regex.matcher(httpUrl)
assert matcher.matches() == true
```

The `split()` method uses a pattern as a delimiter and returns the elements of the parameter broken up by the delimiter. In my example below I split the domain up based on the period (`.`) delimiter:

Another split‘

```
def regex = ~/\./
def domain = 'www.example.com'

println regex.split(domain)
```

That last example is simple but you can use some pretty funky patterns to split up a string.

21. Data types

Groovy does a good job of working out what sort of variable you're using (numbers, strings, booleans etc) but let's look at what's going on under the hood.

Groovy does not preclude the programmer from explicitly declaring a data type, particularly when it would be pertinent to constrain the values being managed. Furthermore, knowledge of data types is very useful for a number of reasons:

1. Use of JVM-compatible libraries may require knowledge of the data types required by method calls.
    - Important if you want to mine the rich collection of existing Java libraries
2. Conversion between different data types (such as decimal numbers to whole numbers) can cause truncation and other (perhaps unexpected) results.
    - Essential knowledge if your program relies on calculations

Most of Java's "core" classes (types) are defined in the `java.lang` package. Groovy enhances some of these in the GDK to give you extra flexibility.

Groovy's use of types

The table below illustrates Groovy's selection of a data type based on a literal value:

Groovy's use of types

Value	Assigned Type
`true`	java.lang.Boolean
`'a'`	java.lang.String
`"This is a String"`	java.lang.String
`"Hello ${Larry}"`	org.codehaus.groovy.runtime.GStringImpl
`127`	java.lang.Integer
`32767`	java.lang.Integer
`2147483647`	java.lang.Integer
`9223372036854775807`	java.lang.Long
`92233720368547758070`	java.math.BigInteger
`3.14`	java.math.BigDecimal
`3.4028235E+38`	java.math.BigDecimal
`1.7976931348623157E+308`	java.math.BigDecimal

It is important to note that the type is selected at each assignment - a variable that is assigned a string such as `"Hello"` is typed as `java.lang.String` but changes to `java.lang.Integer` when later assigned the value `101`.

Using a specific type

A variable can be declared as being of a specific data type. When using a type, drop the `def` keyword:

Declaring a variable using a specific type

```
Integer myNum = 1
String myName = "Fred nurk"
```

Suffixes can also be used if you want to be really specific about the data type Groovy is to use for a number. When using suffixes you use the def keyword to define the variable: `def dozen = 12i`

Type suffixes supported by Groovy

Suffix	Type	Example
——	——	——
`I` or `i`	Integer	`12i`
`L` or `l`	Long	`23423l`
`G` or `g`	BigInteger	`1_000_000g`
`F` or `f`	Float	`3.1415f`
`D` or `d`	Double	`3.1415d`
`G` or `g`	BigDecimal	`3.1415g`

You may have noticed that BigInteger and BigDecimal have the same suffix - this isn't a typo - Groovy works out which one you need simply by determining if the number is a whole number (BigInteger) or a decimal (BigDecimal).

If you're going to use explicit types then you need to know limitations of that type. For example, the following code will fail:

```
assert 3.14159265335f == 3.14159265335d
```

This failure occurs because Float will shorten (narrow) the value to `3.1415927` - not a mistake you'd want to make when measuring optics for your space telescope! You can see which type Groovy will use automatically by running this snippet of code:

```
println 3.14159226535.class.name
```

The `null` Value

Variables that are not assigned a value at declaration are provided a `null` value by default. This is a special reference that indicates the variable is devoid of a value.

Variables can be explicitly assigned `null`:

```
def id = null
```

Available data types

As Groovy imports the `java.lang` package as well as the `java.math.BigDecimal` and `java.math.BigInteger` classes by default, a range of data types are available for immediate use:

- `Boolean`: to store a logical value of `true` or `false`
- Numbers (based on `java.lang.Number`):
  - `Byte`
  - `Short`
  - `Integer`
  - `Long`
  - `Float`
  - `Double`
  - `BigDecimal`
  - `BigInteger`
- `Character`: A single character such as a letter or non-printing character
- `String`: A regular Java-esque piece of text
- `GString`: A Groovy string that allows for interpolation
- `Object`: This is the base class for all other classes
- `Closure`: The class that holds closure values

The types listed above are often referred to as *reference types*, indicating that they relate to a class definition. Groovy also provides a set of *primitive types* that are more closely aligned to the C programming language than an object-oriented language such as Java and Groovy. In most cases, use of a reference type should be preferred and Groovy's dynamic typing uses *reference types*.

Primitive types

The table below maps the types defined in `java.lang` against their equivalent primitive types:

The primitive types

Type	Primitive type	Value range	Size (bits)
Boolean	`boolean`	`true` or `false`	-
Byte	`byte`	-128 to 127, inclusive	8
Short	`short`	-32768 to 32767, inclusive	16
Character	`char`	'\u0000' to '\uffff' inclusive	16
Integer	`int`	-2147483648 to 2147483647, inclusive	32
Long	`long`	-9223372036854775808 to 9223372036854775807, inclusive	64
Float	`float`	32-bit IEEE 754 floating-point numbers	32
Double	`double`	64-bit IEEE 754 floating-point numbers	64

You can check those value ranges by using the `MIN_VALUE` and `MAX_VALUE` constants available on the various classes representing numbers:

Determining value range

```
println Integer.MIN_VALUE
println Integer.MAX_VALUE
println Float.MIN_VALUE
println Float.MAX_VALUE
```

As an object-oriented language Groovy also provides a mechanism for declaring new data types (objects) that extend and encapsulate information to meet a range of requirements. These implicitly extend the `java.lag.Object` class.

Autoboxing

Autoboxing refers to the automatic conversion of a primitive type to a reference type. *Unboxing* is the reverse of *Autoboxing*.

Type Conversions

Groovy will convert values assigned to variables into the variable's declared data type. For example, the code below declares a variable of type "String" and then assigns it `3.14` (a number). The assertion that the variable remains of type "String" will succeed, indicating that `3.14` was converted to a String value by Groovy before being assigned to the `myName` variable.

Checking the type

```
String myName = "Fred nurk"
myName = 3.14
assert myName.class == java.lang.String
```

Care must be taken to not rely totally on this automatic conversion. In the example below the assertion will fail as the `myPi` variable is declared as an `Integer` and the assignment drops the fractional component of `3.14`:

```
def pi = 3.14
Integer myPi = pi
assert myPi == pi
```

Casting

The `as` operator can be used to cast (change) a value to another class.

Casting

```
def pi = 3.1415926535 as Integer
assert 3 == pi
```

You've seen this before... `def myGroceries = ['milk', 'honey'] as Set` - this is actually just casting the list to the `Set` data type.

This will be discussed further in the Operators tutorial.

Converting Numbers

`java.lang.Number` provides a number of methods for converting numbers between the various numerical data types:

- `byteValue()`
- `doubleValue()`
  - also `toDouble()`
- `floatValue()`
  - also `toFloat()`
- `intValue()`

    - also `toInteger()`
- `longValue()`
    - also `toLong()`
- `shortValue()`
- `toBigInteger()`
- `toBigDecimal()`

Here's a small example of grabbing the whole (integer) component from a number:

Getting the integer

```
def pi = 3.1415926535
assert 3 == pi.intValue()
assert 3 == pi.toInteger()
```

III Operators

22. Introduction

Groovy supports a range of operators - those you know from primary school (e.g. + and =), through to more specialised operators.

Operators are used with values and variables and the context in which they're used will vary the resulting output. This introduction lists the range of operators available to you and the following chapters describe each in more detail.

Arithmetic and Conditional Operators

Arithmetic and Conditional Operators

Operator(s)	Type
`=`	Simple Assignment Operator
`!`	Logical Complement Operator
`== !=`	Equality Operators
`+ - * / %, **`	Arithmetic Operators
`> < <= >=`	Relational Operators
`++ --`	Increment and Decrement Operators
`&& || ?:`	Conditional Operators
`<< >> >>> ~ & | ^`	Bitwise Operators
`+= -= *= /= &= |= ^= %= <<= >>= >>>=`	The Compound Assignment Operators

String Operators

String Operators

Operator(s)	Type
`+, <<`	Concatenate Operator
`<<= +=`	Append Operator
`-`	Remove Operator
`-=`	Remove In-place Operator
`*`	Repeat Operator
`*=`	Repeat In-place Operator
`++ --`	Increment and Decrement Operators

Regular Expression Operators

Regular Expression Operators

Operator(s)	Type
`=~`	Find
`==~`	Match

Collection Operators

Collection Operators

Operator(s)	Type
`in`	Membership Operator
`*`	Spread Operator
`*.`	Spread-Dot Operator
`..`	Range Operator
`[]`	Subscript Operator

Object Operators

Object Operators

Operator(s)	Type
`?.`	Safe Navigation Operator
`.@`	Field Operator
`.&`	Method Reference
`as`	Casting Operator
`is`	Identity Operator
`instanceof`	Type Comparison

23. Operator Overloading

Groovy supports something called “operator overloading” and it’s possible for classes to determine how they want operators to behave. Throughout this tutorial I’ll provide some examples of overloading but before we go too far, let’s take a look at what “operator overloading” actually means.

The C++ language provides a mechanism for programmers to customise how operators such as `+` (plus) and `-` (minus) work. This functionality isn’t provided in Java but is available to Groovy programmers. Essentially, a class can include certain methods that replace (overload) the default implementation - as such methods are tied to specific operators.

Consider the `+` operator, as seen in many great additions. You can use the operator in a statement such as `10 + 2` but you can also use the `plus` method instead: `10.plus(2)`. I’d argue (strongly) that using the `plus` method in your code will be far less readable. However, you should be able to see that using the `+` operator actually results in the `plus` method being called.

This this means that you can use operator overloading for evil - say, by creating a numerical class that ignores all mathematical sense. Aside from developer practical jokes you’ll probably only use operator overloading every now and then. Where it does become extremely useful is in the core Groovy classes and the Groovy extensions to the JDK.

In the coming chapters you’ll see a range of operator usage that isn’t available to the Java developer but made available through Groovy’s extensions to the JDK and through the GAPI.

To highlight all of this, operator overloading can be used in classes via the methods associated in the following table:

Methods for overloading operators

Method	Operator
plus	`+`
minus	`-`
div	`/`
mod	`%`
multiply	`*`
power	`**`
equals	`==`
compareTo	`<=>`, `>`, `<`
rightShift	`>>`
leftShift	`<<`
next	`++`
previous	`--`

This list doesn't include all possible operators, just the main ones

Throughout this tutorial on operators I'll provide information as to how certain functionality is obtained through operator overloading. Feel free to glaze past these sections - they're mainly there to explain why/how stuff is happening.

24. Numeric Operators

There are many numeric operators available in Groovy - you're likely to know many of them from your school days.

The following chapters will describe each of the numerical operators in further detail. In this tutorial I just lay the list of operators out for you to quickly see and then describe operator precedence - the order in which the operators are evaluated.

Groovy's numerical operators

Operator(s)	Type
`=`	Simple Assignment Operator
`!`	Logical Complement Operator
`== !=`	Equality Operators
`+ - * / %, **`	Arithmetic Operators
`> < <= >=`	Relational Operators
`++ --`	Increment and Decrement Operators
`&& || ?:`	Conditional Operators
`<< >> >>> ~ & | ^`	Bitwise Operators
`+= -= *= /= &= |= ^= %= <<= >>= >>>=`	The Compound Assignment Operators

Operator Precedence

Operator precedence describes the order in which operators are evaluated. For example, most people know that the multiplication operator is evaluated before the addition, resulting in the following code displaying `20` (and not `60`):

```
println 10 + 2 * 5
```

Parentheses can be used to denote the need for an operator to be evaluated first, allowing the following code to give us `60`:

```
def result = (10 + 2) * 5
println result
```

Note that you can't write the code above as `println (10 + 2) * 5` - Groovy can't evaluate it correctly. You'd need to write it as `println ((10 + 2) * 5)`

Operators with the same (equal) precedence (e.g. `+` and `-`) are evaluated depending on their *associativity*. There are three types of associativity:

- Left-associative: where the operators are grouped left to right
- Associative: where operators are grouped arbitrarily
  - Not seen in Groovy
- Right-associative: where the operators are grouped right to left

For example, the additive operators (+ and `-`) are left associative, meaning that they are evaluated left to right. The expression `6 + 2 - 4` is evaluated as the result of `6 + 2` minus `4`.

In the `6 + 2 - 4` example the result would be the same regardless (`4`) but in other expressions the associativity does matter - consider `6 / 2 * 4`

The simple assignment operator (=) is right associative, resulting in the following code displaying a result of `2`:

Right associativity in assignments

```
def a = 10
def b = 5
def c = 2

a = b = c

println a
```

This isn't typical

You don't tend to see code such as that given above out "in the wild"

Order of Precedence

The order of precedence (highest to lowest) for the arithmetic operators is as follows:

Numeric operator precedence

Operator		**Example**
Postfix increment and decrement		`n++, n--`
Unary operators		
	Positive and negative	`-10`
	Prefix increment and decrement	`++2, --1`
	Logical complement	`!true`
	Bitwise complement	`~0x64`
Power		`10**2`
Multiplicative		`10 * 2, 6 / 3`
Additive		`5 + 5, 10 - 2`
Shift		`>>`
Relational		`10 > 4`
Equality		`1 == 1`
Bitwise AND		`&`
Bitwise XOR		`^`
Bitwise OR		`|`
Logical AND		`true && false`
Logical OR		`true || false`
Ternary		`10 > 2? true: false`
Assignment (simple and compound)		`10 += 2, var = 9`

25. Simple Assignment Operator

Assigns a value to a variable

Simple assignment operator

Operator
=

The equals (=) sign is used to assign a value to a variable:

```
def age = 101
def name = "Fred"
```

Assignment and equality Be aware that the = and == operators perform different roles. In Groovy the equality operator (==) is used to determine if two things are the same.

In the following code the variable `count` is assigned the numeric value `10`:

```
def count = 10
```

If we then wanted compare `count` with another value (`11`) we need to use the == operator:

```
if (count == 11) println "Count is 11"
```

Use of = in the comparison will cause a compilation error:

```
if (count = 11) println "Count is 11"
```

Rest assured that if you accidentally use the simple assignment operator (=) instead of the equality operator (==) you'll not be the first in making that mistake.

26. The Complement Operator

The complement operator negates everything!

The exclamation (`!`) sign is used switch a value to its opposite boolean value. In boolean algebra[1] this is referred to as a Not (or negation) operator.

Complement operator

Operator
`!`

The following example makes sure that "not true" is the same as "false":

```
assert !true == false
```

The complement operator results in the following:

The complement truth table

Value	Complement
`true`	`false`
`false`	`true`

[1]see http://en.wikipedia.org/wiki/Boolean_algebra

27. Equality Operators

Used to determine if two things match

The equality operators return a boolean (`true` or `false`) result from a comparison.

The equality operators

Operator	Name
==	Equal to
!=	Not equal to

All of the following comparisons evaluate as being `true`:

Some equality checks

```
assert -99 == -99
assert 'koala' == 'koala'
assert 'cat' != 'dog'
assert 6 != 7

def domesticAnimal = 'dog'
def wildAnimal = 'lion'
assert domesticAnimal != wildAnimal

def str1 = 'Hello'
def str2 = 'Hello'
assert str1 == str2
```

What Is Equality?

Equality can be a little tricky - both for Groovy and humanity. Think about the statement "Hey cool, we have the same car!". This could mean that we have the same make and model but different instances of a car *or* it could mean that we share a car.

Groovy != Java

For those coming from a Java background, == is used to determine if the two variables reference the same instance and the `equals` method is used to determine if two variables are equivalent. Groovy conflates == and `equals` to perform the same comparison. The `is` method performs the Groovy equivalent of Java's == and returns `true` if two variables refer to the same instance of an object

Using `is` and ==

```
def obj1 = new Object()
def obj2 = new Object()
def obj3 = obj1

assert obj1.is(obj3)
assert ! obj1.is(obj2)

assert obj1 != obj2
assert obj1 == obj3

assert ! obj1.equals(obj2)
assert obj1.equals(obj3)
```

Precedence

In the following example, the equality operator (`!=`) is evaluated before the assignment operator (=), resulting in the value of `truth` being the boolean value `true`:

```
def truth = 'cats' != 'dogs'
assert truth == true
```

Overloading Equality

It is possible to define a custom implementation of == by overriding the `equals(Object obj)` method. This can be handy if your object has a simple method for determining equality, such as comparing staff members by their ID:

Overloading ==

```
class StaffMember {
    def id

    @Override
    boolean equals(obj) {
        if (this.id == obj.id) {
            return true
        } else {
            return false
        }
    }
}

def fred = new StaffMember(id: 12)
def jan = new StaffMember(id: 47)
def janet = new StaffMember(id: 47)

assert fred != jan
assert jan == janet
```

The Groovy package `groovy.transform` provides a handy annotation that generates an `equals` implementation which compares the object's properties. This reduces the previous `StaffMember` class to even fewer lines of code:

Using the built-n `EqualsAndHashCode`

```
@groovy.transform.EqualsAndHashCode
class StaffMember {
    def id
}

def fred = new StaffMember(id: 12)
def jan = new StaffMember(id: 47)
def janet = new StaffMember(id: 47)

assert fred != jan
assert jan == janet
```

Hash codes

The `HashCode` aspect to the annotation indicates that the `hashCode` method is overridden. This method generates a hash code[1] that aids in identifying an instance of the class.

[1]See http://en.wikipedia.org/wiki/Java_hashCode()

28. Arithmetic operators

These are one of the 3 R's (Reading, wRiting and aRithmetic) - that's it at the end.

The five arithmetic operators (`+`, `-`, `*`, `/`, `%`) are familiar to most people from their early school days.

Additive Operators

The additive operators

Operator	Name
+	Plus
-	Minus

The additive operators provide for basic addition and subtraction.

```
assert 1 + 1 == 2
assert 10 - 9 == 1
```

Additive operators are left-associative - they are assessed from left to right:

```
assert 1 + 4 - 3 == 2
```

+ and strings

The + operator can also be used to concatenate (join) two strings:

```
def mySentence = 'This is a game ' + 'of two halves'
```

I'd suggest that you avoid using this as Groovy has better options that we'll look at shortly.

Multiplicative Operators

The multiplicative operators

Operator	Name
*	Multiply
/	Divide
%	Remainder

The remainder operator (`%`) is also commonly referred to as the *modulus* operator and returns the remainder of a division:

```
assert 13 % 2 == 1
```

Multiplicative operators are left-associative:

```
assert 10 * 6 / 2 == 30
```

The Power operator

The power operator

Operator	Name
**	Power

The power operator (`**`) is used to raise a number to the power of the second number:

```
assert 5**3 == 125
```

Use the power

This is a handy shortcut to using `java.lang.Math.pow(5, 3)`

Precedence

Multiplicative operators have precedence over additive operators.

```
assert 10 - 1 * 10 == 0
```

If the result above is surprising and the expected result was 90 then parentheses “`()`” should have been used:

```
assert (10 - 1) * 10 == 90
```

The elements within parentheses have precedence over the rest of the evaluation. This results in `(10 - 1)` being evaluated first and the result being multiplied by 10.

If we consider Pythagoras’ theorem: ({$$}a$^{2 + b}$2 = cˆ2{$$}) the operator precedence will yield the correct answer without requiring parentheses:

```
assert 3 * 3 + 4 * 4 == 5 * 5
```

However, we could use parentheses purely for the sake of clarity:

```
assert (3 * 3) + (4 * 4) == (5 * 5)
```

More power

Naturally, we should have used the power operator for those calculations : `assert 3**2 + 4**2 == 5**2`

Nested parentheses can be used to further delineate an expression. The innermost parentheses are evaluated first, then moving outwards:

```
assert ((10 - 1) * 10) / 2 == 45
```

In the equation above, `(10 - 1)` is evaluated first, the result (9) is then multiplied by 10 and that result (90) being divided by 2.

For significantly more complex calculations such as the quadratic equation (below) parentheses are required if the calculation is to be performed in a single expression:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Part implementation of the quadratic equation

```
def a = 5
def b = 6
def c = 1
def x
x = ((-1 * b) + Math.sqrt((b**2) - (4 * a * c))) / (2 * a)
assert x == -0.2
```

Not for production

The solution given above is not a complete implementation of the equation - it is provided for demonstration purposes only.

29. Relational Operators

Less than, greater than and so on - it's all about how the operands relate.

Similar to the Equality Operators, the expressions involving Relational Operators return a boolean result (`true` or `false`).

The relational operators

Operator	Name
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to
`<=>`	Spaceship

All of the following operations resolve to `true`:

It's all `true`

```
assert 5 > 2
assert 4 >= 3
assert 4 >= 4
assert 8 < 9
assert 6 <= 7
assert 7 <= 7
```

Ordinarily, the operands used in a relational comparison can be compared in a meaningful manner. If they are different data types then the operands need to be able to find a common type for comparison (such as both being numbers) - the following code will cause and exception because Groovy can't be expected compare a string with a number in this way:

```
if ('easy' < 123) println "It's easier than 123"
```

Spaceship

The spaceship operator comes from the Perl programming language. The Spaceship operator is most often seen where sorting is done.

The spaceship operator

Operator
`<=>`

In the example below the `sort` function uses the closure to define the sort algorithm and this is where the spaceship lands:

UFO sighting

```
def nums = [42, -99, 6.3, 1, 612, 1, -128, 28, 0]

//Descending
println nums.sort{n1, n2 -> n2<=>n1 }

//Ascending
println nums.sort{n1, n2 -> n1<=>n2 }
```

The following table indicates the result for spaceship expressions (LHS = left-hand side, RHS = right-hand side):

Expression	Result
LHS less than RHS	-1
LHS equals RHS	0
LHS greater than RHS	1

The following assertions all resolve as true:

```
assert 2 <=> 2 == 0
assert 1 <=> 2 == -1
assert 2 <=> 1 == 1
```

Overloading the relational operators

The `compareTo` method is used by Groovy to assess the result of relational operations:

```
assert 1.compareTo(2) == -1
```

There is a reasonable assumption that the two operands can be coerced (cast) into a similar type. This is why `1.compareTo('cat')` just won't work.

Java's `Comparable`[1] interface is implemented by classes that allow instances to be compared. Custom classes can determine their own appropriate algorithm for the `Comparable`'s `compareTo` method and this will be available when you use the relational operators.

Overloading relational operators

```
class Num implements Comparable {
    def val

    @Override
    int compareTo(obj) {
         if (val < obj.val) {
             return -1
         } else if (val > obj.val) {
             return 1
         } else {
             return 0
         }
    }
}

def a = new Num(val: 2)
def b = new Num(val: 5)
def c = new Num(val: 2)

assert a < b
assert b > a
assert a != b
assert a == c
```

You'll notice that I've tested `a != b` and `a == c` - these equality operators actually calls the `compareTo` method. There's been a bit of discussion about how Groovy handles == and the underlying `equals` and `compareTo` methods so if you're looking to overload these operators it'd be worth your time checking up on what the Groovy developers are planning[2].

[1]http://docs.oracle.com/javase/8/docs/api/index.html
[2]http://blackdragsview.blogspot.fr/2015/02/getting-rid-of-compareto-for.html

30. Increment and Decrement Operators

Number goes up. Number goes down.

The increment operator increments a value to its next value. When you increment or decrement a variable using ++ or -- the variable is modified to the new value.

Inc and dec

Operator	Name
++	Increment
−	Decrement

The increment and decrement operators come in two flavours, prefix and postfix:

- Prefixes are assessed *before* the statement is evaluated
  - `assert ++5 == 6`
- Postfixes are assessed *after* the statement is evaluated
  - `assert 5++ == 5`

Using inc and dec

```
assert 10++ == 10
assert ++10 == 11
assert --10 == 9
assert 10-- == 10
```

The increment and decrement behaves differently depending on the type of value being used.

Booleans don't increment/decrement

Numbers increment/decrement by 1:

```
def num = 10
num++
assert num == 11
```

Characters move to the previous (--) or next (++) character:

```
def ch = 'c'
ch--
assert ch == 'b'
```

Strings are a little odd and it is the last character in the string that is affected:

```
def str = 'hello'
str++
assert str == 'hellp'
```

Enums[1] will cycle through the enum values:

```
enum Priority {
    LOW, MEDIUM, HIGH
}
def task = Priority.LOW
task++
assert task == Priority.MEDIUM
```

BUT be aware that you'll cycle back to the beginning of the value list. The following example is a good example of where you can easily get caught out:

```
def task = Priority.LOW
task--
assert task == Priority.HIGH
```

Overloading the Increment and Decrement Operators

By overloading `next` and `previous` methods, a custom class can support the increment and decrement operators.

The example below demonstrates a class that increments/decrements by 2:

[1]We'll get to Enums much later.

Overloading increment and decrement

```
class StepTwo extends Object {
    def value

    StepTwo(val) {
        this.value = val
    }


    def next() {
        value += 2
        return this
    }

    def previous() {
        value -= 2
        return this
    }

    String toString() {
        return "I have a value of ${this.value}"
    }
}

def two = new StepTwo(3)
println two
two++
println two
two--
println two
```

31. Conditional Operators

These help you lay out your logic.

You'll most often see Conditional-and (`&&`) and Conditional-or (`||`) used in conditional statements such as `if`. We also use them in expressions such as `assert` to determine if a statement is `true` or `false`.

The Conditional Operator (`?:`) is a short-form variant of the `if-else` statement that really helps with code readability. It's also referred to as the "elvis" operator and the "ternary" operator.

The conditional operators

Operator	Name
`&&`	Conditional-and
`||`	Conditional-or
`?:`	Conditional operator

What Is Truth?

All of the following statements resolve as `true` and the assertions all pass:

```
assert 1
assert true
assert 'Hello'
```

Obviously `false` is false but so is `0`, `''` (empty string) and `null`.

The complement operator (`!`) can be used to negate an expression, allowing the following assertions to pass:

```
assert !false
assert !0
assert !''
assert !null
```

Evaluation

Conditional operators are evaluated left-to-right. The assertion in the following code passes as the result of `true && false` is `false` but is then or'd with `true`, resulting in `true`:

```
assert true && false || true
```

Conditional-And

Conditional-and uses the boolean AND to determine if a statement is `true` or `false`. In order to be `true`, both the left-hand and right-hand operands must evaluate to true, as described in the truth table below:

AND Truth Table

LHS	RHS	Result
0	0	0
1	0	0
0	1	0
1	1	1

In a conditional-and statement, both expressions are always evaluated. In the example below, `++counter` is evaluated (giving counter now equal to 1) before the conditional is assessed:

```
def counter = 0
def result = true && ++counter

assert result == true
assert counter == 1
```

Conditional-Or

Conditional-or uses the boolean OR to determine if a statement is `true` or `false`. In order to be `true`, either the left-hand or right-hand operands must evaluate to true, as described in the truth table below:

OR Truth Table

LHS	RHS	Result
0	0	0
1	0	1
0	1	1
1	1	1

In a conditional-or statement, each expression is evaluated in left-to-right order until either an expression resolves to `true` or no expressions are left (resulting in `false` overall). Once an expression is evaluated to `true` no further evaluation of the conditional is performed. This is important to keep in mind if you have an expression performing an operation that you later rely on as it may never be evaluated. In the example below I demonstrate a similar block of code used in the conditional-and section but I'll use a conditional-or. The final assertion (`assert counter == 1`) will fail as `++counter` is never evaluated:

```
def counter = 0
def result = true || ++counter

assert result == true
assert counter == 1
```

Conditional Operator

The conditional operator (`?:`) is most commonly used when assigning a value to a variable. A conditional expression is used for the first operand and placed to the left of the question-mark. If this resolves to `true` then the second operand is evaluated, otherwise the third operator is evaluated. This sounds a little confusing so let's look at an example:

A basic tax calculator

```
def salary = 100000

def taxBracket = salary < 75000 ? 'Bracket 1': 'Bracket 2'

assert taxBracket == 'Bracket 2'
```

In the code above the relational expression (`salary < 75000`) is evaluated and, in this case, resolves to `false` and the third operand (`Bracket 2`) is evaluated and assigned to `taxBracket`. As the operand is just a string there's no real evaluation but we can use any expression that will return a result.

The code below will calculate income tax based on the person's income:

More tax

```
def salary = 100000

def tax = salary < 75000 ? salary * 0.1: salary * 0.2

assert tax == 20000
```

Note that only one of the second *or* third operands are evaluated. For example, if the conditional resolves to `true` the third operand will not be evaluated.

A major benefit of the conditional operator is readability. Consider the previous code being re-written using an `if` statement and I trust you'll see that `?:` makes for more compact and readable code:

A less readable option

```
def salary = 100000
def tax = 0

if (salary < 75000) {
    tax = salary * 0.1
} else {
    tax = salary * 0.2
}

assert tax == 20000
```

Default values

The conditional operator is also really useful for default values - these are used when you want to make sure a variable always has a value. In the code below the `Person` class has been prepared to ensure that any instance that has not been explicitly given a value for the `name` property is assigned `Anonymous` as the name:

Using `?:` to set a default value

```
class Person {
    def name

    def Person(name = '') {
        setName(name)
    }

    def setName(name) {
        this.name = name ?: 'Anonymous'
    }
}

def jim = new Person()
println jim.name
```

Instead of writing `this.name = name ? name: 'Anonymous'` you'll notice that I didn't provide a second operand. This is another bonus for the ternary operator - if the conditional resolves to `true` and no second operand is provided, the result of the conditional is returned. This is a boring way to say that in `this.name = name ?: 'Anonymous'` if `name` is not false then it is assigned to `this.name`.

Use constants

It's usually best to use constants for default values - that way you can easily test a value in other sections of your code.

Avoiding NPEs

NPEs (Null-Pointer Expressions) are the bane of Java and Groovy programmers. You'll see a lot of code checking for `null` and this reduces readability. This is usually in the form `if (myObj != null) {...}` or using the conditional operator `(myObj != null) ? ... : ....`

As `null` evaluates to `false` in Groovy, the conditional operator provides a compact means by which to check if an object is `null` before trying to access the object:

```
def myObj = null
def dump =  myObj ? myObj.dump() : ''
```

In the example above I test `myObj` and, if it isn't `null` then `dump` is given the value returned by `myObj.dump()`. Otherwise an empty string (`''`) is returned.

As an alternative, the safe-navigation operator (`?.`) can be used to test for `null` prior to attempting access on the object:

```
def myObj = null
def dump = myObj?.dump()
```

The safe navigation operator is discussed further under Object Operators.

In Java

In the code below I've provided a small example of checking for `null` in Java. You'll note that we need to be explicit on our conditional (`(t == null)`):

Checking for null in Java

```
public class NullTest {
    public static void main(String[] args) {
        NullTest t = null;

        String output = (t == null) ? "Null" : "Not null";

        System.out.println(output);
    }
}
```

To prove that I'm not just bloating my code to prove a point, here's a slightly more compact version of the example:

Checking for null in Java - compacted

```
public class NullTest {
    public static void main(String[] args) {
        NullTest t = null;
        System.out.println(t==null ? "Null" : "Not null");
    }
}
```

You'll note that I didn't bother with the `output` variable and dropped the parentheses (they're not required). The code is reasonably compact but rewritten in Groovy it gets even tidier:

Checking for null in Groovy

```
class NullTest {
    static main(args) {
        def t = null
        println(t ? 'Not null' : 'Null')
    }
}
```

You still need to check for `null` in Groovy but the shortened conditional operator and the safe navigation operator really do help cut down on the boiler-plate stuff.

32. Bitwise Operators

They're bit-oriented and not so much "wise"

I have to admit that I haven't seen many instances of bitwise manipulation since my university assignments. That's not to say they're not used or not important - I've just not done a lot of programming that's called on bitwise operators.

The bitwise operators

Operator	Name
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR[1]
~	Bitwise negation (Not)
>>	Right shift
>>>	Right shift unsigned
<<	Left shift

Truth Tables

Truth tables describe the results of various logical operations. The truth tables below illustrate the NOT, AND, OR and XOR logic.

Not (~)	Result
Not 0	0
Not 1	1

And (&)	Result
0 AND 0	0
0 AND 1	0
1 AND 0	0
1 AND 1	1

[1]Known as an Exclusive OR

Or (|)	Result
0 OR 0	0
0 OR 1	1
1 OR 0	1
1 OR 1	1

Xor (^)	Result
0 XOR 0	0
0 XOR 1	1
1 XOR 0	1
1 XOR 1	0

Flag example

The unix file permission scheme uses binary flags for read, write and execute permissions on files. You see them when you run `ls -l` as something like `-rwxr-xr-x`. Essentially, this is a set of flags where binary `1` turns on that permission and binary `0` turns it off. The three elements I'll look at here are read (r), write (w) and execute (x):

- READ has the binary value 001 (decimal 1)
- WRITE has the binary value 010 (decimal 2)
- EXECUTE has the binary value 100 (decimal 4)

Let's look at the example code first and then I'll discuss it:

A bitwise example

```
//Create global variables for the permissions
READ = 0b100
WRITE = 0b010
EXECUTE = 0b001

println 'Checking for READ:'
checkFile(READ)

println 'Checking for WRITE:'
checkFile(WRITE)

println 'Checking for READ or EXECUTE:'
checkFile(READ | EXECUTE)
```

```
def checkFile (check) {
    def fileList = [:]
    for (i in 0..7) {
        fileList["File $i"] = i
    }

    for (file in fileList) {
        if (file.value & check) {
            println "$file.key (${displayFilePermission(file.value)}) meets crit\
eria"
        }
    }
}

def displayFilePermission(val) {
    def retval = ""
    retval <<= (READ & val)? 'r': '-'
    retval <<= (WRITE & val)? 'w': '-'
    retval <<= (EXECUTE & val)? 'x': '-'
    return retval
}
```

Please note that I'm not really happy with my example (above) but it's the best I could come up with.

First up I set the flags for each of the three elements using the `0b` prefix to indicate binary numbers:

```
READ = 0b100
WRITE = 0b010
EXECUTE = 0b001
```

I then call my `checkFile` method to see which permissions match what I'm seeking. The third call to `checkFile` is the more interesting as I OR two flags: `READ | EXECUTE`. If I OR the READ flag (`100`) with the EXECUTE flag (`001`) I get `101` (decimal 5):

Value	Binary			Operator
READ	1	0	0	OR
EXECUTE	0	0	1	
Result	**1**	**0**	**1**	=

The `checkFile` method does the checking for me. The first part of the method just creates a set of possible files - enough to cover the various variations of the `rwx` elements:

```
def fileList = [:]
for (i in 0..7) {
    fileList["File $i"] = i
}
```

Creating `fileList` each time the method is called is inefficient - that's why copying tutorial code into your production system is a bad idea :)

It's the second half of `checkFile` that does the important stuff:

```
for (file in fileList) {
    if (file.value & check) {
        println "$file.key (${displayFilePermission(file.value)}) meets criteria"
    }
}
```

The `if (file.value & check)` performs an AND on the check requested (e.g. `READ`) and the file's permissions. If the AND returns a result greater than `0` then the file's permission match the `check`. For example, a file with execute permission (`--x`) meets the `READ | EXECUTE` criteria:

Item	Value	Binary			Operator
`check`	READ | EXECUTE	1	0	1	AND
`file`	`--x`	0	0	1	
	Result	**0**	**0**	**1**	=

A file with read and write permission (`rw-`) also matches:

Item	Value	Binary			Operator
`check`	READ | EXECUTE	1	0	1	AND
`file`	`rw-`	1	1	0	
	Result	**1**	**0**	**0**	=

However, a file with only the write permission (`-w-`) will not successfully match:

Item	Value	Binary			Operator
`check`	READ | EXECUTE	1	0	1	AND
`file`	`-w-`	0	1	0	

Item	Value	Binary	Operator
	Result	**0 0 0**	=

Lastly, the `displayFilePermission` method just helps me display the permissions in the `rwx` format.

Some other quick points follow:

I can negate (~) a value to indicate that I want the inverse of a value, rather than ORing the other options individually:

```
println 'Checking for WRITE or EXECUTE:'
checkFile(~READ)
```

I can XOR (^) to aggregate the permissions but ignore intersections (where both variables contain the same flag):

```
def file1 = READ | EXECUTE
def file2 = WRITE | EXECUTE
println 'File 1: ' << displayFilePermission(file1)
println 'File 2: ' << displayFilePermission(file2)
println 'Result: ' << displayFilePermission(file1 ^ file2)
```

Shift Example

Shifting just moves the bits in a binary to the left (<<) or to the right (>>), depending on the left-hand operand. If we take the following list as a starting point we can see how progressive shifts left change a value:

- 0001 (binary) = 1 (decimal)
- 0010 (binary) = 2 (decimal)
- 0100 (binary) = 4 (decimal)
- 1000 (binary) = 8 (decimal)

With this is mind the following code demonstrates the left- and right-shift operators:

```
assert 2 << 1 == 4        //Left-shift once
assert 2 << 2 == 8        //Left-shift twice
assert 2 >> 1 == 1        //Right-shift once
```

The code below displays a table in which each row represents a value that's be left-shifted by one position more than the prior row:

A bit shifting example

```
def value = 0b0000_0000_0000_0000_1111_1111

println '| Shift    | Hex       | Decimal   | Octal     | Binary                    |'
println '|---------|----------|----------|----------|-------------------------|'
(0..16).each {
    def shifted = value << it
    def hexDisplay = '0x' << Integer.toHexString(shifted).padLeft(6, '0')
    def binDisplay = Integer.toBinaryString(shifted).padLeft(24, '0')
    def decDisplay = "$shifted".padLeft(8, ' ')
    def octDisplay = Integer.toOctalString(shifted).padLeft(8, ' ')
    def shiftDisplay = "$it".padLeft(7, ' ')
    println "| $shiftDisplay | $hexDisplay | $decDisplay | $octDisplay | $binDis\
play |"
}
println '|---------|----------|----------|----------|-------------------------|'
```

Whilst it may not be an overly useful example you can also see the methods for displaying numbers using different number systems (hexadecimal, octal, decimal and binary)

33. Compound Assignment Operators

The compound assignment operators help make code that bit more compact by performing an operation in-place.

The compound assignment operators[1] really just conflate an operation that involves a variable which is, it turn, used to store the result. Let's look at an example to make this clearer. In the example below I really just want to add 10 to the `cost` variable:

```
def cost = 20
cost = cost + 10
assert cost == 30
```

By using a compound assignment operator I can clean up the code (in a very minor way) by performing the operation 'in place':

```
def cost = 20
cost += 10
assert cost == 30
```

The compound assignment operators

Operator	Name
*=	Multiply
/=	Divide
%=	Remainder
+=	Plus
-=	Minus
**=	Power
<<=	Bitwise left-shift
>>=	Bitwise right-shift
>>>=	Bitwise unsigned right-shift
&=	Bitwise And
ˆ=	Bitwise Xor
|=	Bitwise Or

[1]Also known as augmented assignment operators. See http://en.wikipedia.org/wiki/Augmented_assignment

34. String Operators

Groovy provides a very handy set of operators for working with Strings.

You'll spend a lot of your career manipulating strings so anything that makes them less of a hassle is nice. Groovy helps you with the following operators overloaded for your string work.

String operators

Operator(s)	Type
`+, <<`	Concatenate Operator
`<<= +=`	Append Operator
`-`	Remove Operator
`-=`	Remove In-place Operator
`*`	Repeat Operator
`*=`	Repeat In-place Operator
`++ --`	Increment and Decrement Operators

Concatenate Operator

The concatenate operator joins two strings together:

```
println  'It was the best of times.' << 'It was the worst of times.'
```

The above example is rather daft as we could have just put both strings together in the same set of quotes. You're more likely to see strings added to over the course of a program:

```
def quote = 'It was the best of times.'
quote = quote << 'It was the worst of times.'
println quote
```

Instead of using the concatenate you could have just used string interpolation:

```
def quote = 'It was the best of times.'
quote = "$quote It was the worst of times."
println quote
```

The + operator is used in the same manner as `<<`:

```
def quote = 'It was the best of times.'
quote = quote + 'It was the worst of times.'
println quote
```

As you'll see in later in this chapter it's best to use `<<` over +

Concatenation and Types

When you concatenate a string with a number Groovy will cast the number into a string. That means you can end up with 1 + 1 = 11 as the code below demonstrates:

```
assert '1' + 1 == '11'
assert 1 + '1' == '11'
```

If you're really wanting to add a string to a number you need to make sure you explicitly turn the string into a number:

```
assert '1'.toInteger() + 1 == 2
```

This may all sound a bit odd now but if you're trying to work out why your program's maths seems all wrong it's worth looking into where strings and numbers are being mashed together.

Append Operator

The append operator (`<<=`) conflates the assignment (=) and concatenate operators:

```
def quote = 'It was the best of times.'
quote <<= 'It was the worst of times.'
println quote
```

This saves you from having to use `quote = quote << 'It was the worst of times.'`

⚠ += also appends but don't use it - you'll see why in a moment.

Remove Operator

The remove operator (-) removes the first instance of a string or regular expression from another string. The easiest form just removes the first instance of a specific string - in the case of the example below, 'of ' is removed:

```
quote = 'It was the worst of times.' - 'of '
println quote
```

The example above will display `It was the worst times.`

A regular expression pattern can also be used if you want to use a pattern. In the example below, the first occurrence of "bat" or "rat" is removed, resulting in `cat  rat monkey` (*note the double space*)

```
println 'cat bat rat monkey' - ~/[br]at/
```

Remove In-Place Operator

Works just like the remove operator (-) but does the match to the variable as well as modifying it. As for the first remove example, a string can be provided for removal:

```
quote = 'It was the worst of times.'
quote -= 'of '
println quote
```

...and it can also use patterns:

```
def str = 'cat bat rat monkey'
str -= ~/[br]at/
println str
```

Repeat Operator

This is a great operator for those that love repetition! Let's print out `hello` ten-times, each time one a new line:

```
print 'hello\n' * 10
```

Repeat In-PlaceOperator

This one applies the multiplier against the variable and stores the result back in the variable:

```
def complaint = 'ow'
complaint *= 10
println complaint
```

I'll leave it to you to see what happens :)

Increment and Decrement Operators

The increment operator will move the last character of the string to its next value:

```
def str = 'hello'
str++
assert str == 'hellp'
```

The increment/decrement actually works across the Unicode character codes[1] so don't expect code to just use 'a' to 'z':

```
def str = 'fo%'
str--
assert str == 'fo$'
```

For a small experiment, try the following code - it will display a subset of the Unicode characters:

```
//\u00A1 is the Unicode character for an inverted exclamation mark
def chr = '\u00A1'
for (i in 161..191) {
    println chr
    chr++
}
```

I'm sure that this is useful somewhere.....

Warning: Strings Are Expensive!

Many programs build strings up over the course of their operation. This can start becoming very expensive in terms of program resources such as memory because, without the correct approach, the JVM has to copy strings to new memory locations each time you use concatenation.

Java developers turn to the `StringBuilder` and `StringBuffer` classes to make their string building more efficient. Groovy developers using dynamic types can use a few tricks to stay dynamic and ensure efficiency.

Let's take a look at two approaches to building a string. In the first example I'll use the += operator and perform 1000 concatenations:

[1]See the Unicode Character Code Charts

Timing the += concatenation

```
import java.text.DecimalFormat

def quote = 'It was the best of times. It was the worst of times.\n'

def str = ""

def startTime = System.nanoTime()
1000.times {
    str += quote
}
def endTime = System.nanoTime()

DecimalFormat formatter = new DecimalFormat("#,###")
def duration = formatter.format(endTime - startTime)

println "That took $duration nano seconds"
```

Timing issues

Timing in this manner isn't perfect as you won't get the same answer each time you run the script. However, you should notice a significant different between the two examples - enough to indicate that the second version is much more efficient.

In the next example I'll change just 1 thing: I'll use the `<<=` operator rather than `+=`:

Timing the <<= concatenation

```
import java.text.DecimalFormat

def quote = 'It was the best of times. It was the worst of times.\n'

def str = ""

def startTime = System.nanoTime()
1000.times {
    str <<= quote
}
def endTime = System.nanoTime()

DecimalFormat formatter = new DecimalFormat("#,###")
```

```
def duration = formatter.format(endTime - startTime)

println "That took $duration nano seconds"
```

When I run these scripts in groovyConsole I can see that the results are very different. When I ran each test 100 times and averaged the result I got:

- Example 1 (using +=): 24,215,520 ns
- Example 2 (using <<=): 191,490 ns

To me this is evidence enough for me to use '<<=' over +=!

Templates

If you find yourself building strings around boilerplate text - such as a form letter - consider using Groovy's templating system[2].

[2]http://www.groovy-lang.org/templating.html

35. Regular Expression Operators

These are useful for comparing strings to regular expressions. They still don't help you work out the regular expression.

The Variables tutorial covered regular expression (pattern) variables described the `find` and `match` methods. These operators are similar to these methods but return `true` or `false` if the pattern is found in (`find`) or matches (`match`) the first operand (a string).

RegEx operators

Operator(s)	Type
=~	Find
==~	Match

For these operations, the left-hand operand must be a string and the right-hand operand a regular expression pattern.

Find (=~)

Returns `true` if the string on the left-side contains the pattern on the right of the operator.

```
def regex = ~/:[0-9]+/
def httpUrl1 = 'http://www.example.com:8080/'
def httpUrl2 = 'http://www.example.com/'

assert httpUrl1 =~ regex
assert ! (httpUrl2 =~ regex)
```

Match (==~)

Returns `true` if the string on the left-side matches (completely) the pattern provided on the right of the operator

```
def regex = ~/https?:\/\/.*/

def httpUrl = 'http://www.example.com/'
def ftpUrl = 'ftp://ftp.example.com/'

assert httpUrl ==~ regex
assert ! (ftpUrl ==~ regex)
```

36. Collection operators

Working with Lists and Maps is made easier with these handy operators.

A number of operators are provided for working with Lists and Maps. Some overload operators such as `+` and `<<` whilst others (such as `in`) are more collection-oriented. Certain operators work with both Lists and Maps whilst others apply to only one.

Operator(s)	Type
`in`	Membership Operator
`<<`	Append operator
`+`	Addition operator
`-`	Subtraction operator
`+=` `-=`	Compound assignment operators
`*`	Spread Operator
`*.`	Spread-Dot Operator
`..`	Range Operator
`[]`	Subscript Operator

This chapter won't discuss the following operators as they've been described earlier:

- The Range operator creates a list of sequential values and is usually seen with numbers. This is how we created Ranges in the tutorial on Variables.
- The Subscript operator is used to access items in a List or a Map and this was also discussed in the tutorial on Variables.

To finish this chapter off I'll do a little bit of mucking around with set theory.

Membership Operator (Lists and Maps)

The `in` operator is used to determine if an item is "in" a list or is a key in a map.

Using the `in` operator

```
assert 6 in [1, 2, 6, 9]
assert !(3 in [1, 2, 6, 9])

def grades = ['Maths': 'A',
    'English': 'C',
    'Science': 'B'].asImmutable()

assert 'Science' in grades
assert !('French' in grades)
```

Append (Lists and Maps)

The `<<` operator adds a new element to an existing list:

```
def friends = ['Frank', 'Larry']
friends << 'Jane'
println friends
```

It's important to note that appending a list to a list will add a new element that contains the list in the right-hand operand:

```
def friends = ['Frank', 'Larry']
friends << ['Jane', 'Greg']
assert friends == ['Frank', 'Larry', ['Jane', 'Greg']]
```

In order to add the individual items of one list to another I need to use the `addAll()` method:

```
def friends = ['Frank', 'Larry']
friends.addAll(['Jane', 'Greg'])
assert friends == ['Frank', 'Larry', 'Jane', 'Greg']
```

Also check out the Spread operator

I can also use `<<` to append a new key:value pair to a map:

```
def grades = [:]
grades << ['Maths': 'A']
grades << ['English': 'C']
grades << ['Science': 'B']
println grades
```

If I was to add another line `grades << ['Science': 'F']` to the code above, the value for `Science` would be changed to `F` as the map's keys are unique.

Addition (Lists and Maps)

The addition operator (+) returns a ***new*** list with the right-hand operand added:

```
def friends = ['Frank', 'Larry']
assert friends + 'Jane' == ['Frank', 'Larry', 'Jane']
```

This is different to the append operation as the addition results in a new list whereas append adds to the existing list - we'll look into this when we get to compound assignment operators

When we add two lists together we get a union of the two lists returned:

```
def friends = ['Frank', 'Larry']
assert friends + ['Jane', 'Greg'] == ['Frank', 'Larry', 'Jane', 'Greg']
```

This is different to the `addAll()` method as a new list is returned rather than the items being added to the existing list.

Adding to a Set returns a set with the union sans any duplicates:

```
def set = [2, 4, 6, 8] as Set
assert set + [8, 10] == [2, 4, 6, 8, 10] as Set
```

The addition operator will either add a key:value pair to a map or alter the value held against an existing key. In the example below I create a new map item with a result for my French class and then change an existing map item with a reduced English score:

```
def grades = ['Maths': 'A',
    'English': 'C',
    'Science': 'B']

assert grades + ['French': 'F'] == ['Maths': 'A', 'English': 'C', 'Science': 'B'\
, 'French': 'F']
assert grades + ['English': 'D'] == ['Maths': 'A', 'English': 'D', 'Science': 'B\
']
```

Subtraction (Lists and Maps)

The subtraction (`-`) operator will return a ***new*** list with an element removed *if* the list contains the element:

```
assert [2, 4, 6, 8] - 6 == [2, 4, 8]
```

A list can also be subtracted from a list, returning a new list containing items in the left-hand operand (`[2, 4, 6, 8]`) that are not in the right-hand operand (`[2, 6, 12]`):

```
assert [2, 4, 6, 8] - [2, 6, 12] == [4, 8]
```

In the example below my attempt to remove `Gary` doesn't do anything as he's not in the list (this doesn't cause an exception) but I do succeed in un-friending `Frank`:

```
def friends = ['Frank', 'Larry', 'Jane']
assert friends - 'Gary' == ['Frank', 'Larry', 'Jane']
assert friends - 'Frank' == ['Larry', 'Jane']
```

When subtraction is applied to a Map the right-hand operand needs to be a key:value pair. In the example below I attempt 3 things:

1. I attempt to remove `['English': 'D']` but it's not in `grades` so nothing happens
2. I attempt to remove `['French': 'F']` but it's not in `grades` so nothing happens
3. I attempt to remove `['English': 'C']` and **it is** in `grades` so the removal occurs.

Subtracting from a map

```
def grades = ['Maths': 'A',
    'English': 'C',
    'Science': 'B']

assert grades - ['English': 'D'] == ['Maths': 'A', 'English': 'C', 'Science': 'B\
']
assert grades - ['French': 'F'] == ['Maths': 'A', 'English': 'C', 'Science': 'B']
assert grades - ['English': 'C'] == ['Maths': 'A', 'Science': 'B']
```

Compound Assignment Operators (Lists and Maps)

Just as we saw with numbers, the addition and subtraction operators returns a value but don't actually change the variable involved in the operation. To change the value of `grades` I would have needed to assign the resultant back into the variable as follows:

```
grades = grades + ['French': 'F']
assert grades  == ['Maths': 'A', 'English': 'C', 'Science': 'B', 'French': 'F']
```

If we want to use the `grades` variable as the left-hand operand and change its value we can use the compound assignment operators. This means I could also have written the previous example using the += compound assignment:

```
grades += ['French': 'F']
assert grades  == ['Maths': 'A', 'English': 'C', 'Science': 'B', 'French': 'F']
```

Using the append operator in its compound form (<<=) is redundant.

Immutability and Assignment

Consider the following code and see if you're surprised:

```
def grades = ['Maths': 'A',
    'English': 'C',
    'Science': 'B'].asImmutable()

grades += ['French': 'F']
assert grades  == ['Maths': 'A', 'English': 'C', 'Science': 'B', 'French': 'F']
```

Groovy let me change something that's immutable! I should go to the mailing list and report this! The outrage!

Hang on! What `asImmutable()` does is set the elements of the list to be unchangeable but it doesn't make the `grades` variable immutable. As the + operator actually returns a new list value, Groovy is correct in assigning that new value to `grades`.

If I'd used `grades << ['French': 'F']` instead of grades += ['French': 'F'] I would get a `java.lang.UnsupportedOperationException` as I'm actually trying to add a new element to `grades`.

If I really want to make `grades` completely immutable (constant) then I'd need to use the `final` modifier and declare `grades` within a class. The code below demonstrates how I'd set up the class and ensure that attempts to change `grades` cause an exception:

```
class Report {
    final grades = ['Maths': 'A',
        'English': 'C',
        'Science': 'B'].asImmutable()
}

def myReport = new Report()
myReport.grades += ['French': 'F']
```

Running the code above will earn you a `groovy.lang.ReadOnlyPropertyException`.

Spread Operator (Lists)

The Spread operator extracts each element in the List into another list or a method's parameters. This is helpful when you need to include a list's individual items in another list or when your list can be used as parameters in a method call.

Extracting Into Lists

In the first example, one lists's items are extracted into another list:

```
def list = [1, 2, 6, 9]
def list2 = [*list, 12, 34]
assert list2 == [1, 2, 6, 9, 12, 34]
```

This usage looks rather like the `addAll()` method but you may need to be mindful as to the position in which the list is extracted. The example below uses `addAll()` but results in `list2` being ordered differently than in the previous example:

```
def list = [1, 2, 6, 9]
def list2 = [12, 34]
list2.addAll(list)
assert list2 == [12, 34, 1, 2, 6, 9]
```

In this last example I demonstrate an easy approach to creating a union of the two lists:

```
def list = [1, 2, 6, 9]
def list2 = [12, 34]
assert [*list, *list2] == [1, 2, 6, 9, 12, 34]
```

Extracting as Parameters

In the next example I extract the items in the `score` list out, each aligning with the parameters in the method signature:

```
def mean(num1, num2, num3) {
  (num1 + num2 + num3) / 3
}
def scores = [4, 8, 3]
assert mean(*scores) == 5
```

That last example is a little bit of a goldilocks moment - I have exactly the same number of items in the list as the method has parameters. I also have a pretty limited version of the `mean` method - it only works on 3 numbers. However, a method with a varargs parameter is a little less fairy tale:

```
def mean(...nums) {
  def total = 0
  for (item in nums) {
    total += item
  }
  return total / nums.size()
}

def scores = [4, 8, 3]
assert mean(*scores) == 5
```

One last example of using the spread operator:

```
def buyGroceries(...items) {
    for (item in items) {
        println item
    }
}

def shoppingList = ['apples', 'cat food', 'cream']
buyGroceries(*shoppingList)
```

Realistically, the `items` parameter is a list but it proves the point. We'll look into this type of method parameter in the tutorial on methods.

Multiply Operator

Note that you can use `*` as a form of multiplication involving lists but this doesn't return a list containing each element multiplied by the right-hand operand. Rather, the returned list just contains the original list elements repeated by the number of times set by the right-hand operand. In the example below I get `2, 4, 6` repeated `4` times:

```
def list = [2, 4, 6]
println list * 4
```

Spread-Dot Operator (Lists)

The `*.` operator calls an action (method) on each item in the list and returns a new list containing the results. In the example below I call the `reverse()` method on each list element:

```
println(['carrot', 'cabbage', 'cauliflower']*.reverse())
```

The spread operator mimics the `collect()` method - with the previous example being equivalent to the following:

```
['carrot', 'cabbage', 'cauliflower'].collect{it?.reverse()}
```

The spread operator makes the method call using the "Safe navigation Operator" (`?.`) to make sure that the list element isn't null - refer to the Object Operators section for more information. In the next example I include a `null` in the list and the returned list features the `null`:

```
println(['carrot', 'cabbage', null, 'cauliflower']*.reverse())
```

For maps I can't use `*.` so need to use the `collect()` method.

A Little Set Theory

Writing this chapter got me thinking about set theory[1] and how various aspects can be achieved in Groovy lists.

Membership

The `in` method gives us a membership check:

```
assert 4 in [2, 4, 6, 8]
```

Union

The addition operator provides us with the ability to performs unions:

```
assert [2, 4, 6, 8] + [1, 3, 5, 7] == [2, 4, 6, 8, 1, 3, 5, 7]
```

Complements

The subtraction operator (`-`) gives us set complement (difference):

[1] https://en.wikipedia.org/wiki/Set_theory

```
assert [2, 4, 6, 8] - [6, 8, 10] == [2, 4]
```

Intersection

The `disjoint()` method will return `true` if two lists don't contain any intersecting elements:

```
assert [2, 4, 6, 8].disjoint([1, 3, 5, 7]) == true
```

If `disjoint()` returns `false` then some elements intersect.

```
def list1 = [2, 4, 6, 8]
def list2 = [6, 8, 10]

assert ([*list1, *list2] as Set) - (list1 - list2) - (list2 - list1) == [6, 8] a\
s Set
```

Guava Sets Library

All this got me thinking further and looking into Google's Guava libraries - here's some code that uses Guava to scratch my set itch:

Sip some guava

```
@Grab(group='com.google.guava', module='guava', version='18.0')
import static com.google.common.collect.Sets.*

def list1 = [2, 4, 6, 8] as Set
def list2 = [6, 8, 10] as Set

println "Intersection: " << intersection(list1, list2)
println "Union: " << union(list1, list2)
println "Difference (list1 - list2): " << difference(list1, list2)
println "Difference (list2 - list1): " << difference(list2, list1)

println "Cartesian product of list1 and list2"
for (set in cartesianProduct(list1, list2)) {
    println " - $set"
}

println "Powersets of list1: "
for (set in powerSet(list1)) {
    println " - $set"
}
```

37. Object Operators

These operators help you check classes and instances.

It could be argued that all operators are object operators as nearly every variable or value in Groovy is an object. However, these operators are all about working with and checking on the object's structure.

Object operators

Operator(s)	Type
`?.`	Safe Navigation Operator
`as`	Casting Operator
`is`	Identity Operator
`instanceof`	Type Comparison
`.@`	Field Operator
`.&`	Method Reference

Safe Navigation Operator

The Safe Navigation operator (`?.`) checks to make sure that a variable isn't `null` before calling the requested method. Consider the following code:

Hitting a null

```
class Person{
    def name
}

def fred = new Person(name: 'Fred')

//various statements
fred = null
//various statements

println fred.name
```

As `fred` somehow became `null` at some point in the code, that call to `fred.name` causes a nasty `java.lang.NullPointerException` (aka the NPE). This happens a lot as variable (in this case `fred`) can end up being `null` for a number of reasons, including:

- The variable never gets set in the first place - perhaps the initialisation failed but we didn't catch it properly
- A method returns `null` instead of an object instance
- We get passed a parameter that has `null` value.

In order to stop the NPE you'll normally see developers using an `if` statement to check that the variable isn't `null` before trying to call a method:

Checking for null

```
class Person{
    def name
}

def fred = new Person(name: 'Fred')

//various statements
fred = null
//various statements

if (fred) {
    println fred.name
}
```

In the code above, `if (fred)` equates to true if `fred` isn't null - it's a handy bit of Groovy syntax we'll cover when we get to the `if` statement in the next tutorial.

Groovy's Safe Navigation operator saves some time and code. In the code below, Groovy checks that the `fred` variable isn't `null` before trying to access the `name` property - giving us a compact piece of code: `fred?.name`.

Safe navigation

```
class Person{
    def name
}

def fred = new Person(name: 'Fred')

//various statements
fred = null
//various statements

println fred?.name
```

You'll see that "null" is displayed - this is because `fred` is `null`. Groovy doesn't even try to access the `name` property.

Casting Operator

The Casting operator (`as`) changes the data type of a value or variable to the specified class. This is sometimes called "casting", "type conversion" or "coercing". You'll have seen this in action when we created a Set:

```
def nums = [1, 6, 3, 9, 3] as Set
```

For those that like to know the secret sauce, this actually casts `num` to be a `java.util.LinkedHashSet`. I know this because `nums.class.name` knows this.

The `as` tells Groovy that you want to convert the item to be of the specified data type (class) - in the example above I use `Set`. The code below demonstrates a few more conversions:

```
assert 3.14 as Integer == 3
assert 101 as String == '101'
assert true as String == 'true'
assert '987' as Integer == 987
```

You'll note that the cast can be lossy - `3.14 as Integer` caused the value to be truncated to `3`. Not all values can be cast to all types and code such as `'hello, world' as Integer` causes an exception.

Identity Operator

The Identity operator (`is`) determines if two variables are referencing the same object instance. This "operator" is really a method that you call by using `obj1.is(obj2)` to check if `obj1` and `obj2` reference the same instance.

As we saw in the chapters on Equality Operators and Relational Operators, Groovy uses the == operator to determine if two objects are equivalent based on their state. Using == for this purpose is really useful and improves code readability *but* it means that the traditional Java use of == to determine if two objects reference the same instance needs a replacement in Groovy. The `is` method is that replacement.

In the code below I describe a `Person` class and use a very helpful annotation (`@groovy.transform.EqualsAndHashCo` so that Groovy sets up the approach to determining if two instances of `Person` are the same - such that == returns `true`. I've decided that all people will have a unique identifier and, provided two instances have the same identifier, they're the same person. This means that all three variations (`fred`, `freddie`, `frederick`) of the person with the ID `345` are equal (==) to each other. However, by using `is` I can see that, whilst `fred` and `freddie` point to the same instance of Person, `frederick` points to a different instance.

Can I see some identification?

```
@groovy.transform.EqualsAndHashCode(includes="id")
class Person{
    def id
    def name
}

def fred = new Person(id: 345, name: 'Fred')
def freddie = fred
def frederick = new Person(id: 345, name: 'Frederick')

//Check that they're all the same person
assert fred == freddie
assert fred == frederick
assert freddie == frederick

//Check which variable points to the same instance
assert fred.is(freddie)
assert ! fred.is(frederick)
```

Type Comparison

The Type Comparison operator (`instanceof`) is used to determine if a variable is an instance of the specified class.

In this next example I check to make sure that `fred` is a `Person`:

Type comparison

```
class Person{
    def name
}

def fred = new Person(name: 'Fred')

assert fred instanceof Person
```

Checking the variable's type can be useful in dynamically typed languages such as Groovy as it lets us check before we call a property or method that may not be there:

```
class Person{
    def name
}

def fred = new Person(name: 'Fred')

if (fred instanceof Person) {
    println fred?.name
}
```

In my `Person` example I'm not really using the full benefits of object-oriented programming that we can leverage in Groovy - primarily because we're yet to get up to that. However, trust me when I say that class hierarchies and interfaces give us a handy way to build up a family tree of classes and that we can use `instanceof` to check if the object instance has a legacy that helps us achieve an outcome. For example, the `Integer` and `Float` classes are a subclass (child) of the `Number` class.

Meanwhile, under the sea

It's a bit like the Linnaean taxonomy[1]: `assert octopus instanceof Mollusc` and `slug instanceof Mollusc`. Check out the Encyclopedia of Life[2] to learn more about molluscs - those little guys are really cool.

[1]http://en.wikipedia.org/wiki/Linnaean_taxonomy
[2]http://eol.org/pages/2195/overview

In the example below I set up an `add` method that adds two numbers (handy!). Before I try to add those two numbers I use `instanceof` to make sure they're actually Numbers. If they aren't, I throw an exception at you.

```
def add(num1, num2) {
    if (num1 instanceof Number && num2 instanceof Number) {
        return num1 + num2
    }
    throw new IllegalArgumentException('Parameters must be Numbers')
}

assert add(1, 6) == 7
assert add(3.14, 9.2) == 12.34

add('Rabbit', 'Flower')
```

Granted, I could have declared `def add(Number num1, Number num2)` but that wouldn't be very dynamic of me.

Field Operator and Method Reference

I won't discuss these two operators to any depth at this point.

The Field operator (`.@`) provides direct access to an object's property (field) rather than using a getter/setter. *Use with a lot of caution or, even better, don't use it at all.*

The Method Reference operator (`.&`) returns a reference to an object method. This can be handy when you'd like to use the method as a closure. This is a very useful feature so use it at will!

In the example below I describe the `Person` class. When I then create an instance called `example` you'll notice that:

- `example.name = 'Fred'` causes `setName()` to be called
- `println example.name` causes `getName()` to be called
- `example.@name = 'Jane'` and `println example.@name` both access the `name` property directly.
- `def intro = example.&introduceSelf` sets `intro` as a pointer (closure) to the `introduceSelf` method.
  - Which is then called using `intro()`

Field operators and method references

```
class Person {
    def name

    def setName(name) {
        println 'You called setName()'
        this.name = name
    }

    def getName() {
        println 'You called getName()'
        return this.name
    }

    def introduceSelf() {
        println "Hi, my name is ${this.name}"
    }
}

def example = new Person()

//example.name actually calls the getter or setter
example.name = 'Fred'
println example.name

//example.@name directly access the field
example.@name = 'Jane'
println example.@name

//intro holds the reference to the introduceSelf method
def intro = example.&introduceSelf

//This next line calls introduceSelf()
intro()
```

IV Control Flow Statements

38. Introduction

Control flow statements give code the power to go beyond top-to-bottom execution of statements.

Most code needs to reflect a decision-making process. A decision may be an either-or in which we process one set of statements rather than another. Another decision may be to run the same set of statements for a finite (or infinite) period.

Groovy supports a range of control-flow statements:

- The conditional statements:
  - `if` and `if-else`
  - `switch`
  - `try-catch-finally`
- The looping statements:
  - `for`
  - `for`-each
  - `while`
- The branching statements:
  - `break`
  - `continue`
  - `return`

This section will describe the conditional and looping statements and discuss how and where the `break` and `continue` branching statements fit in.

The `try-catch-finally` statement will be explored in the "Exceptions" chapter.

The `return` statement will be explored in the "Methods and Closures" chapter.

39. Blocks

These blocks are not as fun as the wooden version but just as important in building programs.

This chapter is really just a note about syntax - feel free to move to the next chapter after a quick glance here.

Groovy uses curly brackets `{..}` to enclose blocks of code. These are primarily used to group a set of statements within elements such as control flow, class, method and closure declarations. Blocks also let you "partition off" parts of code so that items such as variables aren't visible to other parts of your code - referred to as "scope".

Groovy doesn't allow the use of anonymous code block such as the one below:

A sample block

```
{
    def count = 0
    assert count == 0
}
```

You need to label the block provided above if you want to use it in Groovy code:

A labelled block

```
Block1:{
    def count = 0
    assert count == 0
}
```

This limitation is primarily due to the closure syntax - labelling the block ensures it isn't confused with a closure.

Blocks appearing within blocks are called "nested blocks":

A nested block

```
Block1:{
    def count = 0
    assert count == 0
    NestedBlock: {
        assert count == 0
    }
}
```

If the above code was a Groovy script the block structures really would be redundant "noise" - you're not likely to see usage such as `Block1:{..}` very often. You're more likely to see blocks used with control flow statements (such as `if`):

```
if (true) {
    //some code
}
```

... in class declarations:

```
class Person {

}
```

... in method declarations:

```
def doStuff() {

}
```

... in closure declarations

```
{name ->
    println "Hello, $name"
}
```

Variable Scope

Variable Scope refers to the visibility of a variable from other blocks. A variable declared at the top level is visible to blocks nested within it. The reverse, however, is not true. This lets us declare variables within a block that is specific to that block (and any nested blocks) but doesn't interfere with the rest of the program.

In the example below, the code within `Block1` can "see" the `count` variable so both asserts work:

A sample block

```
def count = 0

Block1: {
    assert count == 0
}

assert count == 0
```

The second `assert` will fail as `count` is not in scope in the main body of the script:

```
Block1: {
    def count = 0
    assert count == 0
}

assert count == 0
```

In the next example, `count` is visible to the nested bock (`Block2`):

```
Block1:{
    def count = 0

    Block2:{
        assert count == 0
    }
}
```

What all of this means is that you need to keep an eye on where the variable can be “seen”. In the example below, the `volume` variable is visible (within scope) of the `if` block but the `dbLevel` variable is local to the `if` block.

```
def volume = 11

if (volume > 10) {
    def dbLevel = 'extreme'
    println "A volume of $volume is $dbLevel"
    //more code
}
```

I repeatedly make a mistake in which I declare a variable within a `try` block but I need to use the variable later on. The code below is an example of my mistake - `println myObj` will never work as `myObj` is not visible at that level.

```
try {
    def myObj = new Object()
} catch (any) {
    //Do something to handle the exception
}

println myObj
```

I've put the `def myObj = new Object()` into the `try` block as the instantiation may cause an exception. What I needed to do was separate the variable definition (`def`) from the creation of a new instance (instantiation):

```
def myObj

try {
    myObj = new Object()
} catch (any) {
    //Do something to handle the exception
}

println myObj
```

40. The `if` Statement

Like a Choose Your Own Adventure, the if statement lets you select what you want to have happen.

Few programs consist of a set of statements read one after another from top-to-bottom. At various points we need the code to evaluate one set of statements rather than another - depending on the current context in which the program is running. The `if` statement is key to directing which route to take.

As an example, let's say we have some code that displays the result of a division. It's very important that we don't try to divide a number by `0` as this causes an error. In the code below we use the `if` statement to check if the denominator is `0` before we perform the division:

A quick if

```
def numerator = 0
def denominator = 0

// ... various statements

if (denominator != 0) {
    println numerator / denominator
}
```

`if` evaluates the conditional expression between the parentheses - `(..)` - and will only process the statement block if the conditional result is `true`.

`if`-`else`

An `else` section can be provided where you want to process statements when the conditional result is `false`.

if - else

```
def numerator = 0
def denominator = 0

// ... various statements

if (denominator != 0) {
    println numerator / denominator
} else {
    println 'I refuse to divide by zero'
}
```

if - else if - else

`if` and `else` let you deal with situations where you have two possible outcomes but sometimes you might have a few conditions that you want to check for:

if - else if - else

```
def game = 'tennis'

if (game == 'chess') {
    println 'I like to play chess'
} else {
    if (game == 'tennis') {
       println 'I can play tennis if you want'
    } else {
        println "Sorry, I don't know how to play $game"
    }
}
```

The code above places another `if` check within the `else` block and checks if the player is asking for a different game ("tennis"). This looks (sort of) clean but start to picture a larger set of checks and the code gets confusing. Instead of nesting the second `if` within the `else` block, Groovy lets you use `else if`:

```
def game = 'tennis'

if (game == 'chess') {
    println 'I like to play chess'
} else if (game == 'tennis') {
    println 'I can play tennis if you want - just let me warm up'
} else if (game == 'golf') {
    println 'I can play golf if you want but I get very angry'
} else {
    println "Sorry, I don't know how to play $game"
}
```

The code above tidies up the nesting by allowing the second `if` to be part of the `else` block. Essentially you can provide a long series of checks in a single `if-else if` set and Groovy will evaluate each until one resolves to true. You can optionally provide an `else` block at the end to ensure that can all other outcomes (defined or otherwise) be dealt with.

When your `if-else` chain gets a little long, look to `switch`

It's important to note that once an `if` evaluates to `true`, no other `if` expressions are evaluated. It's a case of the first positive match wins. In the code below, the conditional `(game=='chess' || game == 'tennis')` is redundant because an earlier condition would have returned `true` (`(game == 'chess')`)

```
if (game == 'chess') {
    println 'I like to play chess'
} else if (game == 'tennis') {
    println 'I can play tennis if you want'
} else if (game=='chess' || game == 'tennis') {
    println 'Can you really play both of these at once?'
} else {
    println "Sorry, I don't know how to play $game"
}
```

That last point is also important as Groovy will also not evaluate any expressions used in further `if` expressions. In the example below, `--i` is never evaluated as the first `if` expression resolves to true:

```
def i = 10

if (++i > 10) {
        //do something
} else if (--i < 10) {
        //do something
}

assert i == 11
```

41. The `switch` Statement

When that if statement gets too big to handle, it's time to switch.

There are times when the number of possibilities are larger than a readable `if-else-if` statement can handle. There are also times where we want to execute a specific set of statements but then "fall through" to others. It's in these places that the `switch` statement comes in handy.

Let's take a look at an example:

Switch

```
switch (game) {
    case 'chess':
        println 'I like to play chess'
        break
    case 'go':
        println 'I can play go if you want'
        break
    case 'ludo':
        println 'I can play ludo if you want'
        break
    default:
        println "Sorry, I don't know how to play $game"
}
```

The `switch` statement evaluates an expression - in the example above it's the value of a variable (`game`) but it could be the result of an operation. This is called the switch value.

The rest of the `switch` block is broken up into `case`s and (optionally) a `default`. Each `case` is assessed against the switch value and the first match is selected. Each case is declared:

- prefixed by the keyword `case`, followed by
- an expression, and lastly,
- a colon `:`

The case expression can be a more complex expression but in our example above I've used a string value (e.g. chess). If the value of game was 'ludo' then the statements under case 'ludo': are evaluated.

The break statement indicates the end of the set of statements for the case and signals that the switch statement has completed. In the example above I've used break for every case but this isn't required. If break isn't provided, execution of the switch is said to "fall through" to the next set of statements. Essentially, Groovy will keep evaluating expressions until either a break is provided or the end of the switch block is reached.

Falling through can be useful if you want to perform the same set of statements for more than one case. The example snippet below provides an example of such a case (pun intended):

```
case 'snowboarding':
case 'snowball fight':
    println 'But it\'s summer!'
    break
```

Here's another example of falling through:

Falling through

```
def score = 2

println 'You win: '

switch (score) {
    case 3:
        println '- gift voucher'
     case 2:
        println '- toy'
    case 1:
        println '- stamp'
    default:
        println '- certificate'
}
```

In the code above the prizes are accumulated depending on your score and a score of 2 sees you win a toy, stamp and certificate - lucky you! Our default ensures that every child wins a prize, regardless of their score.

It can be useful to throw a comment in when you are "falling through" so as to help confirm that you didn't just forget to put in a break

The Versatile Groovy `switch`

The Groovy `switch` statement is much more powerful than Java's and can work across all data types.

> In Java `switch` is limited to the primitive types (and their object wrappers), Strings and Enums and the `case` expression is limited to a value (not an operation).

Groovy achieves through the use of the `isCase` method defined for the `Object` class and overloaded by subclasses such as `Pattern`. Essentially, the switch value is passed to the `case` instance. In the example below, `10.isCase(score)` would be called:

```
switch (score) {
        case 10:
                //etc
}
```

If this all sounds a little foreign, don't worry, just check out the following sections to see how versatile the `switch` statement can be.

Using Ranges

Ranges can be used in the `case` and is selected when the switch value is in the range.

Let's play some blackjack:

Switch with ranges

```
switch (hand) {
    case 1..16:
        println 'HIT'
        break
    case 17..21:
        println 'STAND'
        break
    default:
        println 'BUST'
        break
}
```

Using Regular Expressions

Groovy extends the Java regular expression `Pattern` class to provide for their use in `switch` statements. This can be really handy if you want to test for a number of patterns.

In the example below I set up a list of URI's[1] and assess them against regular expressions based on various URI formats.

The `each` method calls the closure for each value in a list

Switch with RegEx

```
def location = ['urn:isbn:0451450523',
                'http://en.wikipedia.org/wiki/Uniform_resource_locator',
                'HTTPS://secure.example.com/',
                'mailto:duncan@example.com',
                'fax:53454567567']

location.each {
    switch( it.toLowerCase() ) {
        case ~/^urn:.*/ :
            print 'This looks like a URN'
            break
        case ~/^https?:.*/ :
            print 'This looks like a HTTP(S) URL'
            break
        case ~/^mailto:.*/ :
            print 'This looks like an email address'
            break
        default:
            print 'Not sure what this is'
    }
    println " ($it)"
}
```

Using Class Checks

Groovy's `switch` can use a data type (Class) for comparison. Essentially, the switch will use the `instanceof` operator to compare the switch value with a class name provided in the `case`. In

[1]See http://en.wikipedia.org/wiki/Uniform_resource_identifier

the example below I iterate through a list containing elements of various types. I use the `switch` statement to then determine the type of each list item:

Switch with class checks

```
def objList = [ 10,
                'hello',
                [1, 5, 8],
                [name: 'Dave'],
                ~/\n/
              ]

for (item in objList) {
    switch (item) {
        case String:
            println 'You gave me a string'
            break
        case Number:
            println 'You gave me a number'
            break
        case List:
            println 'You gave me a list'
            break
        case Map:
            println 'You gave me a map'
            break
        default:
            println "Sorry, I can't handle instances of ${item.class}"
    }
}
```

42. The `for` Loop

Groovy supports for-in loops, for-each loops and the C-style for loop.

The `for` loop will repeat a block of code whilst its condition is `true`:

for

```
for (<condition>) {
        //statements
}
```

The `for-in` Loop

The for-each loop basically says "for each element is this collection, perform the following":

for-in

```
for (<var> in <iterator>) {
        //statements
}
```

The `for-in` loop may also be referred to as the for-each loop but I prefer the former as the `in` keyword is part of the syntax.

In the Groovy for-in loop we have a variable (`var`) provided as the next item in the `iterator`[1]. In most cases this is a collection of objects such as a list or map.

You can't change the iterator within a loop - it'd just be too confusing! That means code such as the following causes an exception and won't run:

[1]Many Groovy classes implement a feature (interface) named `Iterable` and these let you easily iterate over a collection of items. Lists and Maps are both iterable, making it very easy to loop through their contents.

```
def scores = [4, 8, 2]

for (i in scores) {
    scores << i**2
}
```

Using a range

Consider a variable called `countdown` to hold a range:

```
def countdown = (10..0)
```

By itself, `countdown` probably doesn't seem too useful but let's look at a `for` loop:

```
def countdown = (10..0)

for (i in countdown) {
    println "Launch in T-$i seconds"
}

println 'Blast off!'
```

So let's break down `for (i in countdown) {`:

- The `(...)` parentheses holds the condition
- The variable `i` will hold the next element in `countdown`
  - You can name this as you would any other variable
- The element to the right of the `in` reserved word is the variable being iterated (looped) over
- `{` starts the loop body
  - and its matching `}` closes it

`i` is commonly used to hold the value of each **i**teration

If you run this code in groovyConsole you'll see our launch sequence displayed.

To make our code even more compact (yet readable), we can use the literal value for the range (`10..0`) in our `for` condition:

```
for (i in 10..0) {
    println "Launch in T-$i seconds"
}
println 'Blast off!'
```

Lists

Looping through the items in a list is quite straight forward now you've seen the range example:

Looping through a list

```
def planets = [
    'Mercury',
    'Venus',
    'Earth',
    'Mars'
]

for (planet in planets) {
    println planet
}
```

Ranges work like Lists because, under the hood, ranges *are* Lists! Consider `1..3` - it's actually a list of `[1, 2, 3]`

Maps

Iterating through maps is much the same as we did for lists but the iteration variable consists of the key and the value for that map item. The next code segment will just display the value of each map item:

```
def domains = [
    'com': 'Corporate sites',
    'org': 'Non-commercial sites',
    'mil': 'Military sites'
]

for (site in domains) {
    println site
}
```

The code above will display a set of lines such as `com=Corporate sites` - illustrating that `site` contains a key/value pair.

Instead of printing out the iteration variable (`site`) we can be a little smarter and access the key (`site.key`) and value (`site.value`) individually:

```
def domains = [
    'com': 'Corporate sites',
    'org': 'Non-commercial sites',
    'mil': 'Military sites'
]

for (site in domains) {
    println site.key << ': ' << site.value
}
```

The Java for-each

Groovy supports Java's version of a for-each loop:

```
for (<Type> <var>: <iterator>) {
        //statements
}
```

Unlike Groovy's for-in loop, the Java version:

- uses `:` instead if `in`
- Requires you to declare a data type for the iteration variable
  - Which makes lists consisting of elements with different data types a little more tricky.

Re-writing the previous planets example in the Java for-each style we'd see the following:

```
def planets = [
    'Mercury',
    'Venus',
    'Earth',
    'Mars'
]

for (String planet: planets) {
    println planet
}
```

`for (def planet: planets)` would also work

The Java version has no real benefit over the Groovy approach so I'd stick with the Groovy `for (<var> in <iterator>)`. The Groovy approach also makes for easier iteration over maps.

If you really want to set a data type for your iteration variable you can still be Groovy about it:

```
def planets = [
    'Mercury',
    'Venus',
    'Earth',
    'Mars',
]

for (String planet in planets) {
    println planet
}
```

A C-style for Loop

Java (and Groovy) both offer the `for` loop syntax found in the C programming language:

C-style for

```
for (<init variable>; <condition>; <update expression>) {
        //statements
}
```

- `<init variable>` initialises the loop variable before the first iteration

- `<condition>` sets the condition to be met for the iteration to commence
- `<update expression>` is evaluated after each iteration

This next example does the same as `for (i in (10..1))` but is more verbose:

```
for (i = 10; i >= 0; i--) {
    println i
}
```

So what's happening in `(i = 10; i >= 0; i--)`?

- `i = 10` initialises the loop variable `i` to `10`
- `i >= 0` is the conditional that says "keep looping until `i` is no longer greater than or equal to 10"
- `i--` is evaluated after each iteration - `i` is decremented by `1`.

The `<update expression>` can be a more complex expression:

```
for (i = 0; i <= 20; i += 2) {
    println i
}
```

Infinite loops

The C-style `for` loop can let you set up infinite loops:

An infinite loop

```
for (;;) {

}
```

These are often used in event-based systems. Essentially, the program will enter a loop and await incoming events such as a mouse-click. It's a very interesting aspect to coding but outside the scope of these tutorials.

Staring into the abyss

The term 'infinite' should set off an alarm in your head - be careful as this type of coding can cause your system to stop responding!

The C-style loop doesn't protect you from altering the thing you're working on and, perhaps inadvertently, creating an infinite loop. That means that the code below needs the safety brake provided by (`&& i < 20`) as the loop keeps altering the size of `scores`. If you take out `&& i < 20` and run the code below it won't stop unless you interrupt it or you run out of system resources:

```
def scores = [4, 8, 2]

for (i = 0; i < scores.size && i < 20; i++) {
    println scores[i]
    scores << scores[i]**2
}
```

43. The `while` loop

The while loop comes in one model which will do just fine

The `while` loop will repeat a block of code whilst its condition is `true`:

While

```
while (<condition>) {
        //statements
}
```

Unlike the `for` loop, there is only the one syntax for `while` loops.

`while` loops feature the following:

- `<condition>` can be a value or expression - the loop will only iterate if `<condition>` is `true`.
- The `while` loop syntax does not manage an iterator variable - you must do this yourself
  - It's very easy to create an infinite `while` loop if you're not paying attention

The example below uses a common approach for `while` loops and sets a flag to indicate when the desired state has been reached and the loop can stop:

```
def flag = false
def num = 0

while (!flag) {
    num++
    if (num**2 == 64) {
        flag = true
    }
}

println "The square root of 64 is $num"
```

The code above will increment `num` by `1` each iteration and test to see if num^`2` is equal to `64`. Once the correct `num` is reached, `flag` is change to `true` and the `while` condition now resolves to `false` - indicating that the `while` loop should run the next iteration.

The `while` loop can also be used to create an infinite loop: `while(true) {}`

There is no `do...while` loop

Unlike Java, Groovy does not have a `do...while` loop[1]. Don't be sad.

[1]A ticket has been lodged to request one.

44. Branching statements

These statements may let you improve the efficiency of switches, loops and methods.

Groovy has three branching statements:

- `break`
- `continue`
- `return`

break

We first came across the use of `break` in the `switch` statement - it's used to indicate the end of the set of statements for the `case` and signals that the `switch` statement has completed.

The `break` reserved word is also used to exit out of a loop - even if there are more iterations possible. In the code below I iterate through the list until I reach the value `'Earth'` and then `break` out of the loop:

Break

```
def planets = [
    'Mercury',
    'Venus',
    'Earth',
    'Mars'
]

for (planet in planets) {
    println planet
    if (planet == 'Earth') {
        break
    }
}
```

When we looked at the `while` loop I gave an example of setting a `flag` variable and checking that as the loop's condition. This could have been refined using `break` and not using `flag` at all:

```
def num = 0

while (true) {
    num++
    if (num**2 == 64) {
        break
    }
}

println "The square root of 64 is $num"
```

Be cautious with this sort of code - it doesn't take much to create an infinite loop.

I'll refine that `while` loop just a little further:

```
def num = 0

while (++num) {
    if (num**2 == 64) {
        break
    }
}

println "The square root of 64 is $num"
```

As Groovy resolves a number other than `0` to be `true`, `++num` will allow the loop to commence and we still rely on `break` to get us out of the loop.

continue

The `continue` reserved word will cause the next iteration of the loop - ignoring anything within the rest of the current iteration. In the loop below I use `continue` to ignore `'Earth'`.

Continue

```
def planets = [
    'Mercury',
    'Venus',
    'Earth',
    'Mars'
]

for (planet in planets) {
    if (planet == 'Earth') {
        continue
    }
    println planet
}
```

Labelled branching

If you have a secret desire to create spaghetti code that quickly becomes unreadable then labels are for you!

Labels are used when you have nested loops - a loop within a loop. Both `break` and `continue` can be given a label that directs the program to another nesting level. In the example below I label the first loop `flowerlist` and, when I get to the colour `'Green'` in the inner loop, my `continue` is directed not at the inner loop but at the outer one labelled `flowerlist` - this is called a "labelled continue":

Using labels

```
def colours = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']

def flowers = ['daisy', 'rose', 'tulip']

flowerlist:
    for (flower in flowers) {
        for (colour in colours) {
            if (colour == 'Green') {
                continue flowerlist
            }
            println "$colour $flower"
        }
    }
```

As you can see, the label consists of a name, followed by a colon (`:`) that appears above the loop being labelled. The code above will display the following:

```
Red daisy
Orange daisy
Yellow daisy
Red rose
Orange rose
Yellow rose
Red tulip
Orange tulip
Yellow tulip
```

Perhaps our business logic dictated that no rainbow colour below 'Yellow' was to be associated with a flower - it's a silly example.

A labelled `break` works much the same way and defers control back to the loop designated by the label.

There are times when labels are useful but really think about what you need to do before resorting to them. For example, using a `break` in the code above would have done the job:

```
def colours = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']

def flowers = ['daisy', 'rose', 'tulip']

for (flower in flowers) {
    for (colour in colours) {
        if (colour == 'Green') {
            break
        }
        println "$colour $flower"
    }
}
```

return

The `return` statement is used to hand control back to the caller[1]. In most cases, `return` is used in a method or closure to return a value. In the most basic usage, the keyword `return` just appears on a line by itself:

[1]The caller is the section of code that called the method/closure.

```
//some code
return
```

Any code that appears below the `return` is unreachable and is never evaluated:

```
//some code...
return
//unreachable code...
```

Return can be provided a value and this is returned to the caller:

```
//some code
return 3.14
```

You can use an expression in the `return` statement and the result is returned:

```
//some code
return circumference / diameter
```

Using `return` within the body of a script[2] will cause it to exit as the command-line/terminal is the caller. When exiting you can choose to return a value - usually `0` if the script succeeds, or an error number if it fails.

[2]By body I mean the `main` part of the script.

V Exceptions

45. Introduction

Exceptions and Errors are just objects that can bring your program to a rapid halt.

Groovy takes on Java's approach to raising, handling and reporting errors and exceptions:

- **Errors** are serious problems that usually cause an application to halt ("crash"). In most cases you won't handle these - they'll happen and the program will fall over.
- **Exceptions** are less serious and applications, with good exception handling set up, may be able to keep going.

At the heart of this model is the `java.lang.Throwable`[1] class. Exceptions and Errors are are two sides to the `Throwable` family tree:

- `java.lang.Exception` is the superclass (parent) of all exceptions
- `java.lang.Error` is the superclass (parent) of all errors

When writing Groovy applications you're likely to interact with exceptions in the following ways:

- Catching exceptions (using `try-catch`) thrown by various classes, be they written by you, part of the Java/Groovy libraries or from a third-party.
- Throwing your own exceptions using (`throw`) to signal that something's gone wrong in your code.
- Being surprised and frustrated when exceptions "just happen"

Your goal should be to use defensive coding and testing that seeks to handle what you think is necessary but be prepared for issues to arise and use facilities such as logging to help you determine the root-cause.

[1]Check out the JavaDoc for Throwable

46. Common Exceptions

Let's take a look at some exceptions you're likely to see through the day.

In the following sections we'll thrown some errors and exceptions (on purpose) so that we can check out some of the most common children of `java.lang.Throwable` you're likely to see. As a bonus we'll also discover a few approaches to avoiding them.

java.lang.NullPointerException

The good old `NullPointerException` (NPE) will haunt your debugging sessions for years to come. Basically it means that you've tried to call a method or access a property on an object that isn't there (i.e. the variable is `null`). Let's cause an NPE - it's easy:

```
def tmp = null
tmp.class
```

To avoid these, check for `null` by:

- Using the Safe Navigation operator (`tmp?.class`)
- Checking the variable with an `if` statement before trying to access it

java.lang.AssertionError

This is an `Error`, not an `Exception` and occurs when your `assert` expression evaluates to `false`:

```
assert true == false : 'This cannot be'
```

`: 'This cannot be'` sets the error message for a failed assertion

When your Groovy assertions fail you actually seem to receive a `org.codehaus.groovy.runtime.powerassert.Powe`
- a subclass of `java.lang.AssertionError`.

The `assert` statement is usually seen in tests and small scripts. It's usually better to signal an incorrect state/situation using `throws` - more about them in a later chapter.

java.lang.NumberFormatException

Groovy can be used as a dynamic language so there are times where you may try to do something to a value that just can't be done. Trying to convince something that it can be a number when it really can't be will give you a `NumberFormatException`:

```
'Kitten' as Integer
```

If you really need a variable to be a specific type you could use the class check feature of the `switch` statement. In the code below I check if `value` is of a type within the `Number` family tree before I try to convert it to an `Integer`. If it isn't, I could `throw` and exception or handle it in some other way:

```
def value = 'kitten'

switch (value) {
    case Number:
        value = value as Integer
        break;
    default:
        println 'I should throw an exception'
}
```

groovy.lang.MissingPropertyException

This exception happens when you try to access an object's property but it doesn't have that property. In the example below, my `Person` class doesn't have a `name` property:

```
class Person {}
def jim = new Person()
jim.name
```

Using `try-catch` around this sort of exception can let you duck type[1] - an approach that uses an object's properties and methods to determine if something is possible.

I reckon that the `hasProperty()` method available on all Groovy objects is a cleaner approach than the `try-catch` option. We can check the object has the property before trying to use it and without causing an exception:

[1]https://en.wikipedia.org/wiki/Duck_typing

```
class Person {}
def jim = new Person()

if (jim.hasProperty('name')) {
    jim.name
}
```

groovy.lang.MissingMethodException

This exception appears when you call a method on an object that doesn't support the method. The lack of a `getName()` method on the `Person` class will cause an exception for poor old `jim`:

```
class Person {}
def jim = new Person()
jim.getName()
```

There's no `hasMethod()` method - it's called `respondsTo()`:

```
class Person {}
def jim = new Person()

if (jim.respondsTo('getName')) {
    jim.getName()
}
```

The `respondsTo()` method just checks if the method exists but we may want to be certain that the methods exists *and* has the parameter list we're after. To achieve this we need to call `respondsTo()` with a second parameter, a list of the method argument types we expect - `respondsTo(String name, Object[] argTypes)`.

java.lang.IndexOutOfBoundsException

These appear when you attempt to `get()` an index from a list that isn't there. The code below attempts to get the 5th element from a 3-element list:

```
def list = [0, 1, 2]
println list.get(5)
```

Note that if we'd written that code in a slightly different way, we'd get `null` returned rather than an exception raised:

```
def list = [0, 1, 2]
println list[5]
```

Checking `list.size()` (or the `length` property) before trying to access is another option:

```
def list = [0, 1, 2]

if (list.size() >= 5) {
    println list.get(5)
}
```

Of course the `for-in` loop will iterate through the list and not try to give you an element that isn't there.

> The same can't be said for the C-style `for` loop or `while`

```
def list = [0, 1, 2]

for (item in list) {
    println item
}
```

java.lang.ArrayIndexOutOfBoundsException

If you just had to use an array (instead of a list) then you'll get an `ArrayIndexOutOfBoundsException` if you attempt to use an array index that isn't there:

```
Integer[] nums = [0, 1, 2]
nums[5]
```

We can use the `length` property to make sure we don't try to access an element that isn't there:

```
Integer[] nums = [0, 1, 2]

if (nums.length >= 5) {
    nums[5]
}
```

The `for-in` loop is also handy for staying within the bounds:

```
Integer[] nums = [0, 1, 2]

for (item in nums) {
    println item
}
```

47. Anatomy of an Exception

Before we handle the exception let's see what it looks like.

`Throwable` and all of its descendants provide a number of useful methods. Let's start off with a very basic `try-catch` block so that we can then dissect the NullPointerException:

What a catch

```
try {
    def person = null
    person.getName()
} catch (any) {
    println "Exception received: ${any.class.name}"
    println()
    println "getMessage() returns: ${any.getMessage()}"
    println()
    println "getCause() returns: ${any.getCause()}"
    println()
    println 'getStackTrace() returned:'
    for (item in any.getStackTrace()) {
        println "${item.getFileName()}\t${item.getLineNumber()}\t${item.getClass\
Name()}\t${item.getMethodName()}\t${item.isNativeMethod()}"
    }
}
```

`any.class.name`
: Tells us the type of exception that was raised

  We get: `java.lang.NullPointerException`

`getMessage()`
: Returns the message that was set by the thrower of the exception

  We get: `Cannot invoke method getName() on null object`

`getCause()`
: This isn't always set but can be handy to know. Essentially this is set if the exception is "relaying" another exception.

  We get: `null`

`for (item in any.getStackTrace()) {..}`

The `getStackTrace()` method returns an array of `java.lang.StackTraceElement`. Each `StackTraceElement` can then be further dissected to see what was happening when the exception occurred.

In my example I output a tab separated row for each `StackTraceElement` - the methods that I call should be reasonably self-explanatory but you can always check the Java API[1] for more information.

To just dump out the stack trace I could have just used `printStackTrace()`

[1]http://docs.oracle.com/javase/8/docs/api/java/lang/StackTraceElement.html

48. Try-Catch-Finally

The try-catch-finally statement is used for catching and handling exceptions and errors.

The `try-catch-finally` statement can have three forms:

1. `try-catch`
2. `try-catch-finally`
3. `try-finally`

Groovy doesn't support the `try-with-resources` syntax available in Java. Groovy's closure infrastructure often takes care of closing resources for you (you may not even need a `finally` block).[1]

Scope

Before getting into the specific syntax it's important to point out that the `try` statement presents a block with its own scope. This means that variables declared within the `try` block can't been seen outside the block. In the example below the `assert` fails not because `score` doesn't equal `12` but because the `score` variable is out-of-scope:

```
try {
    def score = 12
} catch (any) {

}

assert score == 12
```

In fact, the `score` variable won't be visible to the `catch` block or a `finally block`. If you do need to access `score` outside of the `try` block then you should declare it before the `try`:

[1] I guess I'll have to make sure I explain this when we get to Files and Databases.

```
def score
try {
    score = 12
} catch (any) {

}

assert score == 12
```

try-catch

The basic format of `try-catch` is as follows:

`try-catch`

```
try {
        //statements...
} catch (<Exception type> <var>) {
        //statements...
}
```

If an exception occurs within the `try` block:

- The catch variable (`var`) is set to the exception instance
- If `Exception type` is provided (this is optional) then the `catch` block will only be activated if the exception is of that type.

In the following example I generate an exception on purpose by dividing `10` by `0`. The `catch` block is set up to catch any exception raised:

```
try {
    10 / 0
} catch (any) {
    println any.message
    any.printStackTrace()
}
```

As we saw in an earlier chapter, exceptions in Groovy are objects - they have fields and methods, allowing me to display the `message` within the exception as well as display the stack trace.

I like using `any` as the catch variable when I'm prepared to catch anything - it just reads nicely. However, you're also likely to see `e`, `exc` or a shortened version of the exception name (e.g. `npe` for a `NullPointerException`).

An Exception type can be provided for a `catch` block. In the example below I only catch an `ArithmeticException` - any other type of exception is raised up to the caller:

```
try {
    10 / 0
} catch (ArithmeticException e) {
    println 'I just caught an ArithmeticException'
}
```

A `try-catch` statement can consist of multiple `catch` blocks, each targeting specific exceptions. In this next example I explicitly catch `ArithmeticException` and have a default `catch` to pick up any other exception raised:

```
try {
    10 / 0
} catch (ArithmeticException e) {
    println 'I just caught an ArithmeticException'
} catch (any) {
    println 'What just happened?'
}
```

In this next example I explicitly catch `IllegalStateException` and `ArithmeticException` and have a default `catch` to pick up any other exception raised:

```
try {
    10 / 0
} catch (IllegalStateException e) {
    println 'I just caught an IllegalStateException'
} catch (ArithmeticException e) {
    println 'I just caught an ArithmeticException'
} catch (any) {
    println 'What just happened?'
}
```

In that last example, `IllegalStateException` will never be raised but I've used it here to indicate syntax. Later in this chapter we'll make this example more meaningful

If we wanted to handle a number of exception types (sometimes called a multicatch) in the same manner, the | operator can be used within the same `catch` to provide a separator for the exception types:

```
try {
    10 / 0
} catch (IllegalStateException | ArithmeticException e) {
    println 'I just caught an exception I want to handle'
} catch (any) {
    println 'What just happened?'
}
```

The last `catch` block (`catch (any)`) is a very useful one to reflect on when considering how you deal with exceptions. By providing a `catch` block you are flagging an intent to actually do **something useful** with an exception. If you don't actually intend to add any value to the situation then I'd suggest not catching the exception. That way it will pass up the chain to the calling code (which may chose to actually handle the exception). The buck stops at the top-level code (e.g. the script) and, without explicit handling, the exception will be displayed and the program halted.

My preference is to use `try-finally` if I just want to tidy up resources (such as files) if an exception occurs - that way the exception will pass up to the caller but I don't get in the way. In most cases I use the explicit form of `catch` and indicate which Exception type I am prepared to handle.

try-catch-finally

The `try-catch` statement can have an optional `finally` block. This is run regardless of if an exception is raised. The `finally` block doesn't receive any incoming parameter and can appear no more than once per `try`:

`try-catch-finally`

```
try {
        //statements...
} catch (<Exception type> <var>) {
        //statements...
} finally {
        //statements
}
```

In the code below, the `println` in the finally block will run regardless of whether an exception was raised or the `try` block completed successfully:

```
try {
    10 / 0
} catch (any) {
    println any.message
} finally {
    println 'I think we survived'
}
```

But why use `finally`? Exceptions are used in a large number of places, including:

- Failure to read/write a file
- Failure to access a database
- Trying to access `null`

The `finally` block comes in handy when you need to tidy up resources before either completing the `try` block or losing control to an exception. You'll most commonly see `finally` used to close files, disconnect from databases or return the system to a state in which it can continue.

try-finally

The third form of the `try` statement doesn't provide a `catch` block:

`try-finally`

```
try {
        //statements...
} finally {
        //statements
}
```

Essentially we're indicating that any exceptions will just be raised up to the caller but we need to tidy up some resources before losing control:

```
try {
    10 / 0
} finally {
    println 'Finally block has been evaluated'
}
```

Don’t `println` in production

My examples have largely performed a `println` and this alone is not acceptable in production code. What a more serious piece of code should do is attempt to recover from the problem, most likely log the issue, and raise the exception to the caller if there’s no possibility of repair.

49. How Much Should We try?

Just a small chapter on technique

Whilst you could wrap an entire script/method/closure in a huge `try` statement this will quickly get confusing - imagine how many `catch` blocks you'll need at the end of a 150-line script! Yes, you can `catch (any)` but you'd really have no clue what caused the exception.

Take the following example:

```
try {
    methodOne()
    methodTwo()
    //148 more lines of code :)
} catch (MethodOneException e) {

} catch (MethodTwoException e) {

}
```

I tend to "localise" my `try` blocks so that they deal more specifically with the exception arising from a specific method call:

```
try {
    methodOne()
} catch (MethodOneException e) {

}

try {
    methodTwo()
} catch (MethodTwoException e) {

}
```

Although that last example results in more code I suggest it's more useful as it helps localise the exception. In the first example I could end up catching a `MethodTwoException` from my call to `methodOne` - probably not what I really intended. If both methods throw the same exception type then localising really helps as I'll know which method threw the exception.

Additionally, if you go with the rule that you're only catching exceptions you're prepared to actually do something with, a lot of your code may not be wrapped with a `try` statement at all!

50. Causing an Exception

The `throw` keyword is used to intentionally cause an exception

The `throw` statement will cause an exception to be thrown. You'll use this inside your own code to either throw an exception type already provided by Java or Groovy or to throw an exception type that you have developed. Remembering that exceptions are classes, you can create your own subclass of `Exception` for your own project. Let's take a small step back for now and see how we throw an exception.

Often, `new` is used within a `throw` statement to initialise a new exception instance. The example below will `throw` a `new` instance of the `IllegalStateException`. As part of creating the new exception instance I pass it a message to help inform the user as to why they're getting an exception:

Throwing an exception

```
def numerator = 10
def denominator = -1

try {
    if (denominator < 0) {
        throw new IllegalStateException('I haven\'t learnt how to divide negativ\
e numbers')
    } else {
        return numerator / denominator
    }
} catch (any) {
    println "${any.message} (${any.class.name})"
}
```

You can use `new` to create a new instance of an exception but this won't cause the exception to be thrown - you'll need to `throw` it. The long-form version of the previous example would appear as:

```
def ise = new IllegalStateException('I haven\'t learnt how to divide negative nu\
mbers')
throw ise
```

Use the short-form - it's more concise and it's what other developers expect to see.

In the code above I `throw` an exception (`IllegalStateException`) to indicate a limitation in my program. As before, the `catch` will receive the thrown exception but this time it could be either the `IllegalStateException` or the `ArithmeticException`:

```
def numerator = 10
def denominator = -1

try {
    if (denominator < 0) {
        throw new IllegalStateException('I haven\'t learnt how to divide negativ\
e numbers')
    } else {
        return numerator / denominator
    }
} catch (IllegalStateException e) {
    println 'I just caught an IllegalStateException'
} catch (ArithmeticException e) {
    println 'I just caught an ArithmeticException'
} catch (any) {
    println 'What just happened?'
}
```

The code above makes a little more sense than my earlier example as `denominator` may be `0` or a negative number and we are concerned with both possibilities. As mentioned earlier, we can use `|` to handle more than one exception type within the same `catch` block:

```
def numerator = 10
def denominator = -1

try {
    if (denominator < 0) {
        throw new IllegalStateException('I haven\'t learnt how to divide negativ\
e numbers')
    } else {
        return numerator / denominator
    }
} catch (IllegalStateException | ArithmeticException e) {
    println 'Stand back, I know how to handle this'
} catch (any) {
    println 'What just happened?'
}
```

Constructing an Exception

There are a few approaches to constructing an exception, the main ones being:

`new IllegalStateException()`
: Creates a new instance without a message

`new IllegalStateException('Sorry, Dave, I cannot allow this')`
: Creates a new instance with a message

`new IllegalStateException('Sorry, Dave, I cannot allow this', e)`
: Creates a new instance with a message and provides the exception (`e`) that caused the exception you're now throwing.

Let's look at an example usage of the last variation. In the following code snippet I `catch` two possible exceptions and bundle `e` into a new exception instance but pass `e` to the new instance so that the caller could determine the cause:

```
catch (IllegalStateException | ArithmeticException e) {
    throw new IllegalStateException('Unable to perform operation', e)
}
```

Upon having the exception thrown at me I could use the `getCause()` method to determine if there was an underlying cause.

Creating Your Own Exception

Whilst you can write your own Exceptions (they're just objects after all), you should look to the pre-packaged ones and see if they meet your needs. Alternatively, consider if you can write a subclass that provides refinement to an existing exception rather than just extending `Exception`.

51. Catching Errors

You can catch an error but you probably shouldn't.

Before we go too far on this one please note that **errors indicate serious/abnormal problems and shouldn't be caught**.

Errors are different to exceptions as they indicate a fundamental issue rather than a recoverable problem. It's highly unlikely you'll ever need to handle an error so treat it like a cold and don't try to catch one. They may "crash" your program but it's likely any treatment that you try to apply will make things worse. For example, if somehow you managed to catch a `java.lang.OutOfMemoryError`, how would you respond?

I was unsure if I should even include this section but feel it more useful to flag that, whilst you can handle errors, you probably shouldn't.

One more time: **don't catch errors**.

Errors are objects and can be caught much like exceptions but the following won't work:

```
try {
    assert true == false
} catch (err) {
    println 'I caught the error!'
    println err.message
}
```

The code above doesn't do what we hoped as, by default, the `catch` is looking for `Exception` or one of its subclasses[1] - probably because we **shouldn't catch errors**.

In order to catch an error we must provide `Error` (or a subclass of `Error`) as the data type in the `catch`:

[1]This is why, when we write our own exception we extend `Exception` rather than implement `Throwable`.

```
try {
    assert true == false
} catch (Error err) {
    println 'I caught the error!'
    println err.message
}
```

52. Methods and Exceptions

Another small chapter on technique

The code in this chapter is rather skewed as we'd rarely throw an exception and catch it within the same `try-catch` block. Rather, we're most likely to throw an exception from a method back to the caller and we use `try` blocks to catch exceptions from methods we're calling.

Throwing an exception from a method results in no `return` value being returned to the caller - the caller needs to handle the exception. It's important to note that, where the `try` and/or `catch` block contain a `return` statement, the `finally` block will be evaluated **before** the `return` is actually processed. In the next two examples below, the `println` in the `finally` block will always display.

Using `return` in `catch`

```
def testMethod() {
    try {
        10 / 0
    } catch (any) {
        return
    } finally {
        println 'Finally block has been evaluated'
    }
}
```

Using `return` in `try`

```
def testMethod() {
    try {
        return 100
    } catch (any) {
        println 'Exception'
    } finally {
        println 'Finally block has been evaluated'
    }
}

testMethod()
```

We’re about to get to the tutorial on Methods.

VI Methods

53. Introduction

ℹ Methods help us break code into logical and reusable segments.

Methods (sometimes also called functions) are blocks of code that can be run more than once and encapsulate a segment of logic. We define a method by writing the code that will be run when the method is called. *Calling* a method is the process of your code asking the method to start.

Groovy, like Java, is object-oriented and works around classes. C and Pascal are procedural and work around functions. Whilst the methods described here may look a bit like C-style programming that lets you build libraries of functions, what is really happening is Groovy wraps your code in a `class` definition behind the scenes. You're only likely to create "raw" methods, like the ones below, as a means to break up your scripts. More usually you'll create methods within your classes.

Methods have a number of features:

1. Methods have names
    - this allows you to call your method in a meaningful way
2. Methods can accept parameters
    - these are inputs into your method that can affect how your method operates
3. Methods can return a result value
    - this can be captured by a variable from the code calling the method
4. Methods have their own scope
    - they can have their own variables and not inadvertently affect the rest of your program

We've already looked at various methods for use with variables such as lists and maps so you've seen methods being called throughout the previous chapters.

54. The Basics

Let's start by examining a new method we'll write to calculate the average of the numbers in a list:

```
def determineAverage(list) {
    return list.sum() / list.size()
}
```

Breaking that code up we can see:

1. The `def` reserved word is used to commence the method declaration
    - Much like we use when defining a variable
2. `determineAverage` is the name of the method
3. The method accepts a single parameter, `list`
4. A single value is returned using the `return` reserved word
    - In this case it's the result of `list.sum() / list.size()`

The method name (`determineAverage`) may look a bit odd but it uses a naming strategy called "lower Camel Case"[1]. The camel aspect is the use of upper-case letters to indicate individual words in the name (hence `Average`). The first word in the method name is a verb (`determine`) to indicate that a method "does" something.

Let's return to that `determineAverage` method and get a complete script together - you can copy and paste this into `groovyConsole` and run it to experience the method first-hand:

```
def scores = [2, 7, 4, 3]
def result = determineAverage(scores)
println result

def determineAverage(list) {
    return list.sum() / list.size()
}
```

Let's look at the main components of that script:

1. The `determineAverage` method is defined as before

[1]https://en.wikipedia.org/wiki/CamelCase

    - This can appear above or below the other code - Groovy doesn't care but I like to put them at the bottom of scripts so the reader is presented with the main script first
2. A new list of numbers is created: `def scores = [2, 7, 4, 3]`
3. The method is called with the `scores` variable passed as a parameter
4. The return value (result) of `determineAverage` is stored in the `result` variable.

In the example I called the method using `determineAverage(scores)` but, in many cases, I don't need to use the parentheses and `determineAverage scores` would have also worked. That's why `println 'hello, world'` works just fine. This works really well when you start to use Groovy to construct domain-specific languages[2].

[2]http://groovy-lang.org/dsls.html

55. Parameters

Parameters are method inputs that are used by the method to produce a result.

Let's look at the last example from the previous chapter:

```
def scores = [2, 7, 4, 3]
def result = determineAverage(scores)
println result

def determineAverage(list) {
    return list.sum() / list.size()
}
```

You might be wondering what happened to the `scores` variable once it was passed to `determineAverage` as a parameter. Basically, Groovy gave it another name (`list`) for use within the method. Inside the method, `list` is just another variable. This means that if `determineAverage` changes `list` in some way, this is reflected in the `scores` variable used in the main script:

A poor example

```
def scores = [2, 7, 4, 3]
def result = determineAverage(scores)
println result
println scores

def determineAverage(list) {
    list << 9
    return list.sum() / list.size()
}
```

The code above is **very** poorly behaved - it modifies `list` by adding a new item. Unless you provided documentation that explicitly states that you will change a parameter, most developers will assume that their parameters will be safely untouched by your method.

Deep down in the system, `scores` and `list` are names that point to the same piece of memory. Understanding how programming languages handle memory is an extremely important part of programming. I don't really cover it in this tutorial but start by looking up "memory management" in Wikipedia[1].

Declaring data types for parameters

Groovy lets you designate a data type for your parameters:

```
def determineAverage(ArrayList list) {
    return list.sum() / list.size()
}
```

As you start to develop classes and larger programs, methods create your Application Programming Interface (API). Such methods can be called by other people's code and they could be using another JVM language (such as Java). It can make their life a little easier if you indicate the data types you're expecting for your parameters. Alternatively, you can stay true to dynamic typing and let people know through your documentation.

Multiple parameters

Let's look at another method - one that needs several parameters:

```
def callFriend(name, phone, message) {
    println "Dialling $name on $phone"
    println "Hi, $name - $message"
}
```

Either of these calls would work:

```
callFriend('Barry', '0400 123 456', 'Did you see that local sporting team?')

callFriend 'Alex', '07 3344 0000', 'Could you please check on my pets whilst I\'\
m away?'
```

Each parameter may be typed if needed:

[1]https://en.wikipedia.org/wiki/Memory_management

```
def callFriend(String name, String phone, String message) {..}
```

You can provide a mix of typed and untyped parameters but this is a little messy and I think it’s bad form so I can’t be bothered encouraging such an action by providing an example.

56. Default Values for Parameters

Parameters can have a default value to indicate the most likely usage

One or more parameters can be defined with a default value. This can be really useful if most calls to the method will use the defaults:

```
def displayMessage (message,
                    title = 'Important message:',
                        border = true) {

    def borderText = ''

    if (border) {
         borderText = '-------------------------'
    }

    println """\
    $borderText
    $title
    \t $message
    $borderText
    """
}
```

The `displayMethod` can be called in a number of ways:

- `displayMessage 'Preparing to shut down. Please save your work'` -
- `displayMessage 'The system appears to have crashed', 'Error!'`
- `displayMessage 'Be prepared for the happiness patrol', 'Public announcement:', false`

When you get to method overloading and other object-oriented concepts you'll see that default parameter values can aid in reducing the variations of a method that you might need to provide.

57. Named Arguments

Groovy uses a sneaky Map to provide named argument functionality

You can use named arguments by having the first parameter be a Map. Consider the method below:

```
def displayMessage (options, message) {

    def borderText = ''
    if (! options.containsKey('border') || options.border) {
         borderText = '--------------------------'
    }

    def title = 'Important message:'
    if (options.title) {
         title = options.title
    }

    println """\
    $borderText
    $title
    \t $message
    $borderText
    """
}
```

The `options` parameter is actually a Map and this lets the caller use an interesting Groovy syntax when calling the method. Instead of passing a Map (`[:]`) to the `options` parameter, the caller can use the `key: value` format in their method call. This lets us call `displayMessage` in many ways, including:

- `displayMessage(title: 'Canberra', border: true, 'The capital of Australia')`
- `displayMessage title: 'Time', "It is now ${new Date()}"`
- `displayMessage border: false, 'Hang in there little buddy!'`

My recommendation is to use named parameters for non-essential parameters and to make sure that your method can operate correctly if a named parameter is not provided.

If others are to be using your method or **you** need to remember which named parameters are available, then you'll make sure that you add some useful documentation to the method.

58. Variable Arguments (Varargs)

Some methods are happy to take as many parameters as you can muster.

There are times where we want to pass a variable number of parameters to the method. However, the parameter list for a method is fixed.

One approach is to use a list for a catch-all parameter, such as `items` does in the code below:

```
buyGroceries 'The Corner Store', ['apples', 'cat food', 'cream']

def buyGroceries(store, items) {
    println "I'm off to $store to buy:"
    for (item in items) {
        println "  -$item"
    }
}
```

Whilst the list path is an option, Groovy supports the use of variable arguments (varargs) using the "three-dot" (...) notation for the last (and only the last) parameter:

```
buyGroceries 'apples', 'cat food', 'cream'

def buyGroceries(... items) {
    for (item in items) {
        println item
    }
}
```

We can set a specific data type for the `items` parameter by placing the type before the `...`:

```
def buyGroceries(String... items) {
    for (item in items) {
        println item
    }
}
```

Let’s return to the first example in this chapter and rewrite it using varargs:

```
buyGroceries 'The Corner Store', 'apples', 'cat food', 'cream'

def buyGroceries(store, ... items) {
    println "I'm off to $store to buy:"
    for (item in items) {
        println "  -$item"
    }
}
```

So the `items` parameter is actually a list inside `buyGroceries` but the caller just passes a series of values to the method.

Varargs last

Putting a parameter after the variable arguments parameter doesn’t make a great deal of sense as it’d be tricky to work out where `items` finished. So, even if `def buyGroceries(store, ...items, travelTime)` was legitimate (and it isn’t) I’d suggest that readability is lost and `travelTime` should appear before `items`. Alternatively, going back to using an array/list for `items` would solve the problem.

59. Return value

Most methods are built to return an answer.

When we started this tutorial I provided a very basic method for calculating averages. I've rewritten it slightly to use varargs and this is a good starting point into using the `return` statement:

```
println determineAverage(10, 20, 30, 40)

def determineAverage(... list) {
    return list.sum() / list.size()
}
```

In the code above I `return` the average to the caller so, instead of printing out the result I could also assign it to a variable: `def avg = determineAverage(10, 20, 30, 40)`.

Using the `return` reserved word isn't required as Groovy will return the result of the last statement:

```
println determineAverage(10, 20, 30, 40)

def determineAverage(... list) {
    list.sum() / list.size()
}
```

You can use `return` to explicitly exit a method. By itself, `return` actually returns `null`. In the useless method I provide below, I explicitly provide `return`:

```
def printer(message) {
    println message
    return
}

printer('hello, world')
```

That use of `return` in the last bit of code was redundant as the method would exit anyway (it had nothing left to do). However, this can be handy if the last expression to run in a method returns a value that *we don't want* as the return value for our method.

Anything after a return is inaccessible, as illustrated by my even more useless method:

```
def printer(message) {
    println message
    return
    println 'Help, I don\'t exist'
}
```

That last line in the method will never, ever, ever be called. **But** if I really wanted it to be called I can use the `try-finally` approach:

```
def printer(message) {
    try{
        println message
        return
    } finally {
        println 'Help, I don\'t exist'
    }
}
```

Now, that last bit of text will be displayed as it's in a `finally` block. This example is rather daft but it serves to illustrate how `finally` can be used to clean up something like an open file before the `return` is actioned.

Multiple Returns

You can have more than one `return` statement in a method but only one will ever be evaluated. This is really handy as it localises the return and prevents the method from further evaluation. You can also place a `return` at the very end of the method block to ensure that the method always returns a value. In the code below I use two `return` statements in the `switch` but also have `return false` at the bottom of the method just in case something slips through (most likely when I add in code at a later date):

```
def checkAnimalAsPet(animal) {
    switch(animal){
        case 'dog':
        case 'cat':
            return true
        default:
            return false
    }
    return false
}
```

```
assert checkAnimalAsPet('cat')
assert checkAnimalAsPet('dog')
assert checkAnimalAsPet('lion') == false
assert checkAnimalAsPet('pterodactyl') == false
```

You'll note that, in the `checkAnimalAsPet` method I have a `switch` with no breaks. Essentially, the `return` is breaking out of the switch and the method all at once.

Declaring data types for return values

A data type can be declared for the return value by replacing `def` with the class name: `Number determineAverage(... list){..}`:

```
println determineAverage(10, 20, 30, 40)

Number determineAverage(... list) {
    return list.sum() / list.size()
}
```

This is very handy if your method is to be accessed as part of an API, especially by Java programs.

Sequential method calls

In many examples I have used a method's returned value to set a value of a variable, in an `assert` or as the input to a `println`. As the return value has its own type, we can actually call a method straight from the method call. This can be useful if one method call is just an intermediary step towards our goal and we don't want to explicitly store its return value.

In the example below I call the `tokenize` method which returns a List of the words in the `poem` I then call the `size` method for that list to determine how many words are in the poem:

```
def poem = '''\
Once a jolly swagman camped by a billabong
Under the shade of a coolibah tree,
And he sang as he watched and waited till his billy boiled:
"Who'll come a-waltzing Matilda, with me?"
'''

poem.tokenize().size()
```

Safely navigate the sequence

If we needed to be cautious we could have used the safe navigation operator: `poem?.tokenize()?.size()`

60. Throwing an exception

Your method can throw an exception.

A method is able to throw an exception just as we saw in the Exceptions tutorial. In the code below I throw an exception if the caller to `determineAverage()` tries to pass a String through as a parameter:

```
def determineAverage(...values) throws IllegalArgumentException {
    for (item in values) {
        if (item instanceof String) {
            throw new IllegalArgumentException()
        }
    }
    return values.sum() / values.size()
}

//This works:
assert determineAverage(12, 18) == 15

//This does not work - we get an exception
determineAverage(5, 5, 8, 'kitten')
```

None of that code is new to you except for the `throws IllegalArgumentException` that forms part of the method's signature[1]. The use of `throws` helps describe our method a little better by making callers aware of what to expect.

Multiple exceptions can be listed against `throws`, as seen in the example below:

[1]This is the section of the method definition contain the return type, method name, parameters and thrown exceptions. As always, Wikipedia has some further information

```
def determineAverage(...values)
  throws IllegalArgumentException, NullPointerException {
    for (item in values) {
        if (item instanceof String) {
            throw new IllegalArgumentException()
        }
    }
    return values.sum() / values.size()
}
```

Groovy doesn't require that you explicitly provide a `throws` listing if your method throws an exception or passes up an exception that it doesn't handle. However, if your method is to be used by others, I'd suggest that including `throws` is worth the effort.

You may note that, in that last example, I placed `throws` on a second line - this helps readability as it breaks up the display of the signature just slightly.

61. Documenting a method

GroovDoc helps us document our methods

The `groovydoc` command that comes with the Groovy installation can be used to generate HTML documentation from comments within your code. GroovyDoc is based on JavaDoc[1] and uses much the same syntax.

Let's look at a Groovy method that features GroovyDoc comments:

```
/**
 * Returns the average of the parameters
 *
 * @param values  a series of numerical values
 * @throws IllegalArgumentException if a values parameter is a String
 * @returns The average of the values
 */
def determineAverage(...values)
  throws IllegalArgumentException {
    for (item in values) {
        if (item instanceof String) {
            throw new IllegalArgumentException()
        }
    }
    return values.sum() / values.size()
}
```

Taking a look at the commenting:

- The multi-line comment block starts with `/**` to indicate that this is a GroovyDoc
- The first piece of text provides the summary of the method. It's one line and meant to be terse.
- A set of `@param` tags can be provided to describe each parameter.
  - The format is `@param <parameter name> <summary>`
  - You don't provide the parameter type even if you declare one

[1] http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html#documentationcomments

- Each exception that can be thrown by the method is listed against a `@throws` tag and provides a summary as to when the exception may be thrown.
  - The format is `@throws <exception class> <summary>`
- The `@returns` tag describes what the method will `return`
  - The format is `@returns <summary>`

If you copy the sample code into a file named `Average.groovy` you can then run the following command in your command line/terminal:

```
groovydoc -d doc Average.groovy
```

This will produce a directory named `doc` that contains a set of documentation files. Inside the `doc` directory you'll see `index.html` - open this in a browser to view your documentation.

As you click through the various links you'll find the documentation for the `determineAverage()` method. It'll contain the following information (but look a lot prettier):

---

java.lang.Object determineAverage(java.lang.Object... values)

Returns the average of the parameters

throws:
IllegalArgumentException if a parameter is a String

returns:
The average of the values

Parameters:
values - a series of numerical values

If you keep clicking links in the html files but can't find it, look in the `DefaultPackage` directory for a file name `Average.html` - that'll be what you're after.

62. Techniques

Here are some ideas to help your method writing but not your method acting.

I'd like to tell you that your programming career will be all about writing perfect code that never has problems but I'd just be lying. Here are some techniques to help make sure your methods are more robust and helpful to other programmers.

Valid parameters

We understand that the method `determineAverage(...values)` is expecting a list of numbers and have used a reasonably clear naming strategy (`determineAverage`) to display that the method is number-oriented *but* what happens when our colleague gives us something like:

```
determineAverage(5, 5, 8, 'kitten')
```

Clearly, *kittens* aren't something that the *average* calculation can understand[1]. If you try to run that code you'll get a nasty error that basically says your code has failed. Don't be too hard on your colleague though - perhaps they've loaded data from a file that's become corrupted by felines.

Comment your method

Firstly, make sure that `determineAverage` has some useful documentation such as:

[1]Pun intended

```
/**
 * Returns the average of the parameters
 *
 * @param values  a series of numerical values
 * @throws IllegalArgumentException if a parameter is a String
 * @returns The average of the values
 */
def determineAverage(...values)
    return values.sum() / values.size()
}
```

In the example above I've just added a GroovyDoc comment block that describes what the method does, its parameter and what it will return. At the very least, other developers will see how they *should* be using my method.

Check the parameters

Next, I can be more defensive in my coding and make sure that the method has a prerequisite that needs to be met before it attempts to run.

```
/**
 * Returns the average of the parameters
 *
 * @param values  a series of numerical values
 * @throws IllegalArgumentException if a parameter is a String
 * @returns The average of the values
 */
def determineAverage(...values)
  throws IllegalArgumentException {
    for (item in values) {
        if (item instanceof String) {
            throw new IllegalArgumentException()
        }
    }
    return values.sum() / values.size()
}

//This works:
assert determineAverage(12, 18) == 15

//This does not work - we get an exception
determineAverage(5, 5, 8, 'kitten')
```

The approach above checks to make sure that no parameter is a `String` - if you pass one to the method you'll get an exception thrown back at you. In reality I should make sure that only numbers can be passed in and my check won't pick up a `Boolean` value - more on this in a moment.

What do you think would happen if I called the method with no parameters - `determineAverage()`?

(pause)

Well, the division would attempt to divide by zero and that's a fail so I need to also check that `values` isn't empty (I'll leave out the comments for brevity):

```
def determineAverage(...values)
  throws IllegalArgumentException {
    for (item in values) {
        if (item instanceof String) {
            throw new IllegalArgumentException()
        }
    }

    if (!values) {
        return 0
    }

    return values.sum() / values.size()
}
```

Note that if no parameters are passed, I return `0`. I really don't like returning `null` from methods as it makes other developers then have to check for `null`. I also don't want to raise an exception - I'm happy enough to say that the average of no values is `0`.

Get really typed

If I really want to get specific with the data types I'll take as parameters and return from the method then I can switch to static typing. I can make sure that all my parameters are of type `Number` (or one of its subtypes) and that I will return a value of type `Number`. The code below really gets specific about data types:

```
/**
 * Returns the average of the parameters
 *
 * @param values  a series of numerical values
 * @throws IllegalArgumentException if a parameter is a String
 * @returns The average of the values
 */
Number determineAverage(Number...values) {

    if (!values) {
        return 0 as Number
    }

    Integer sum = values.sum() as Integer
    Integer n = values.length
    Number average = (sum / n) as Number
    return  average
}
```

The following two calls to the method would work:

```
assert determineAverage(2, 7, 4, 4) == 4.25
assert determineAverage() == 0
```

…but the following two calls won't work:

```
determineAverage(2, 7, 4, 4, 'kitten')
determineAverage(2, 7, 4, 4, true)
```

If you are writing a method that needs to be very specific about data types for parameters and/or return values then this is the way to go.

Testing

I'd get into a lot of trouble from experienced developers if I just left this chapter without mentioning testing. I'm hoping that one day there'll be a Testing tutorial added to this book but, until then, check out Unit Tests section[2] on the Groovy website.

Oh ok, here's a little example using Spock[3]!

[2]http://groovy-lang.org/testing.html

[3]http://docs.spockframework.org/en/latest/

Firstly, this won't run in your groovyConsole. You need to copy the code into the online Spock web console[4][5] and then click on "Run Script":

A Spock example

```
import spock.lang.Specification

class MathDemo {
    /**
     * Returns the average of the parameters
     *
     * @param values  a series of numerical values
     * @throws IllegalArgumentException if a parameter is a String
     * @returns The average of the values
     */
    static determineAverage(...values)
      throws IllegalArgumentException {
        for (item in values) {
            if (! (item instanceof Number)) {
                throw new IllegalArgumentException()
            }
        }

        if (!values) {
            return 0
        }

        return values.sum() / values.size()
    }
}

class AvgSpec extends Specification {

    @Unroll
    def "average of #values gives #result"(values, result){
        expect:
            MathDemo.determineAverage(*values) == result

        where:
            values       || result
            [1,2,3]      || 2
```

[4]http://meetspock.appspot.com

[5]I've published the code to make it easy for you but can't promise that this link will always work.

```
            [2, 7, 4, 4] || 4.25
            []           || 0
    }

    @Unroll
    def "determineAverage called with #values throws #exception"(values, excepti\
on){
        setup:
           def e = getException(MathDemo.&determineAverage, *values)

        expect:
            exception == e?.class

        where:
            values       || exception
            ['kitten', 1]|| java.lang.IllegalArgumentException
            [99, true]   || java.lang.IllegalArgumentException
            [1,2,3]      || null
    }

    Exception getException(closure, ...args){
        try{
            closure.call(args)
            return null
        } catch(any) {
            return any
        }
    }
}
```

When you run this in the Spock web console you should get:

```
AvgSpec
 - average of [1, 2, 3] gives 2
 - average of [2, 7, 4, 4] gives 4.25
 - average of [] gives 0
 - determineAverage called with [kitten, 1] throws class java.lang.IllegalArgume\
ntException
 - determineAverage called with [99, true] throws class java.lang.IllegalArgumen\
tException
 - determineAverage called with [1, 2, 3] throws null
```

So there's a lot going on here that we haven't covered in the tutorials so far but let's try to break it down:

1. I wrapped the `determineAverage()` method in a class named `MathDemo` and made it a `static` method
    - I won't explain this here - just trust me that you can call `MathDemo.determineAverage()`
    - But do note that I've changed `determineAverage()` to better check that the parameters are numbers.
2. I then created a spock test `Specification` subclass called `AvgSpec`
    1. The first test is `def "average of #values gives #result"(values, result)`
        1. This runs a series of tests using the data table in the `where:` block
        2. Yes, that's right, Groovy will let you use a string as the name of the method - that's v.cool but you can't use interpolation (`${}`).
    2. The second test is `def "determineAverage called with #values throws #exception"(values, exception)`
        1. This checks to make sure that the `IllegalArgumentException` is thrown for incorrect parameters

As I say, there are a number of topics such as classes and closures that I haven't covered - this example was just a quick one and should make sense as you learn about those additional concepts.

VII Closures

63. Introduction

A closure is a method/function that is either named (stored in a variable) or anonymous (passed directly to a method).

Closures represent a reference to a function (method) that is accompanied by a referencing environment. This means that you can store function references in a variable and pass them to a function and that the original scope of the closure is available. Confused? Well, let's get into it!

Terminology

Whilst I tend to use the terms "function" and "method" somewhat interchangeably when discussing Groovy, people discussing closures more generally use the term "function". Some languages have very specific meanings for terms such as "function", "procedure", "method" but I'd suggest that if you use the term "method" or "function", most Java and Groovy developers will get your meaning. Alternatively, you could try to be more universal and call them all "subroutines"

64. Introducing Closures

In the last tutorial we looked at methods and this prepares us as we start to look at closures. If you take a look at the following code you're likely to quickly see what the `printer()` method does:

```
def printer(){
    println 'Hello, world'
}

def cls = this.&printer
cls()
```

So what about `def cls = this.&printer`? Well, `.&` is the Method Reference operator and it returns a reference to the `printer()` method. I use `this` in the script as I need to reference the current instance in which I'm running - remember that Groovy wraps the script in a class.

Once I have the reference I can then call the closure with `cls()`.

Note the following:

1. When I define the closure (`def cls = this.&printer`) I don't put the parentheses after the method name (`printer`)
2. When I call the closure I pass in the parameters

Say I set this up a little differently and create a `Test` class with two `printer` methods - one that takes a parameter and one that doesn't:

```
class Test {
    static printer(){
        println 'Hello, world'
    }

    static printer(name) {
        println "Hello, $name"
    }
}

def cls = Test.&printer
cls()
cls('Newman')
```

You'll see if you run that last example that the call to the closure (`cls`) will result in the associated method being called depending on the parameters I provide.

Anonymous Functions

In the first examples of this chapter I used the Method Reference operator to point to an existing method that I had defined in a class. Closures can also be defined using anonymous functions. This lets us create a function at the point we define the variable.

In the next example I create an anonymous function using the regular block syntax (`{..}`) and store the function reference in `cls`:

```
def cls = { println 'Hello, world' }
cls()
```

That's pretty nifty! We can define a function when needed and store it in a variable. This variable can then be passed to methods and other closures as a parameter.

The 'it' parameter

Anonymous functions get a single parameter named `it` by default. That means that you can use `it` as a parameter inside your function and `it` will contain the parameter passed in the call to the closure.

Let's write a version of the `printer` method that takes a parameter:

```
def cls = { println "Hello, $it" }
cls('Jerry')
```

More Terminology

I'll use the terms "closures" and "anonymous functions" interchangeably. Whilst we know that there's a difference between the two we usually use anonymous functions with closures rather than via the Method Reference operator.

65. Parameters

Closures can take parameters - just like methods.

We just saw that closures have an in-built `it` parameter but we can specify a parameter when we declare the closure:

```
def cls = { name -> println "Hello, $name" }
cls('Jerry')
```

In the example above I declare the `name` parameter and this replaces `it` - in fact, if I called `it` within the closure I'd get an exception.

The syntax of closures is starting to become evident:

1. Start the closure with {
2. List parameters, followed by ->
3. Write a set of statements for the closure body
4. End the closure with }

Each parameter is separated using a comma (`,`):

```
def cls = { name, pet -> println "Hello, $name. How is your $pet?" }
cls('Jerry', 'cat')
```

As the closure gets more complicated I like to break it up over a number of lines. I start the closure and declare the parameters on the first line. The body of the closure then follows much the same as a method body and a final line closes off the closure:

```
def cls = { name, pet ->
    println "Hello, $name. How is your $pet?"
}
```

Closure parameters let me do the same things I can do with method parameters:

1. Use data types for parameters
2. Provide default values
3. Varargs
4. Named parameters

Parameter data types:

```
def cls = { String name, String pet ->
    println "Hello, $name. How is your $pet?"
}
```

Default values:

```
def cls = { name, pet = 'dog' ->
    println "Hello, $name. How is your $pet?"
}
```

Varargs:

```
def cls = { name, ...pets ->
    println "Hello, $name. How are your $pets?"
}

cls('Herman', 'cat', 'dog', 'spider')
```

Named parameters:

```
def cls = { grades, name ->
    println "This year $name received the following grades: $grades"
}

cls(maths: 'D', science: 'C', art: 'A', 'Albert')
```

So closures and methods are rather similar - there’s no black magic going on.

66. Returning Values

Closures can return values just like methods.

Just like methods, closures can return a value. In fact both closures and methods always return a value.

In this next example, the result of the last expression is returned (the value of `num1` or `num2`):

```
def max = { num1, num2 ->
    if (num1 >= num2) {
        num1
    } else if (num2 > num1) {
        num2
    }
}

println max(14, 6)
```

Alternatively, we can use the `return` statement to explicitly exit the closure and return a value:

```
def myClosure = { num1, num2 ->
    if (num1 >= num2) {
        return num1
    } else if (num2 > num1) {
        return num2
    }
}

println myClosure(14, 6)
```

Unlike a method, closures can't explicitly define the data type of a return value. You might take a crack at `Integer maxNumber = {num1, num2 -> ...}` to set the return type but this statement won't work as `maxNumber` doesn't hold the result of the closure - it holds a reference to the closure.

`Closure myClosure = { num1, num2 -> ...}` will work as the `myClosure` variable is actually of the `Closure` data type.

67. Closures as Method Parameters

Closures get really interesting when you start passing them around

We often say closures are first-class citizens in Groovy. This means that you can use them across the language in a similar manner to other first-class citizens, such as variables.

The best example to start our exploration is the `each` method that's available to your collections (lists and maps). `each` can be called on a list or a map and is passed a closure as a parameter, allowing you to perform operations on each member of the collection. In the next example I call `each` on the `numList` variable and pass it a very basic closure as a parameter:

```
def numList = [6, 9, 11, 4, 7]
def myClosure = { println it }
numList.each(myClosure)
```

However, we can avoid `myClosure` altogether as we don't really need to use it anywhere else in our code. So, we use an anonymous closure - one that doesn't get a name (i.e. assigned to a variable). This is really useful if we don't need to use the closure outside of the method being called:

```
def numList = [6, 9, 11, 4, 7]
numList.each({ println it })
```

Whilst the closure can be placed within the `(...)` parentheses, this becomes cumbersome for larger anonymous closures so Groovy lets us drop the parentheses:

```
def numList = [6, 9, 11, 4, 7]
numList.each { println it }
```

For a final example, we can call the `each` method directly against the literal array, just to prove that Groovy has a versatile and flexible syntax:

```
[6, 9, 11, 4, 7].each { println it }
```

Methods with Closure Parameters

Your own methods can accept one or more closures as a parameter. When doing this we usually follow a basic convention of:

- Use `closure` as the parameter name
- Put the `closure` parameter at the end of the parameter list

In the example below, the `mutator` method accepts a closure as the second parameter:

```
def mutator(value, closure) {
    closure(value)
}

mutator 10, {it**2}
```

We are able to call `mutator` in a number of ways:

- `mutator(10, {it**2})`
- `mutator 10, {it**2}`
- `mutator(10) {it**2}`

Those last two options are very useful if you're going to pass a non-trivial closure as it helps the reader see that the last parameter is a closure and not some random block.

Here's another example, a method `findPrimes` that accepts a list of numbers (such as a range) and a closure. The method loops through the list and, when it determines the item is a prime number it will call our closure:

```
def findPrimes(list, closure) {
    for (num in list) {
        def isPrime = true
        if (num != 2) {
            for (divisor in (2..num/2)) {
                if (num % divisor == 0) {
                    isPrime = false
                    break
                }
            }
        }
        if (isPrime) {
            closure(num)
        }
    }
}

def numList = (1..100)
findPrimes numList, { println "I found a prime: $it" }
```

If I wanted to be specific about my parameter data types, the correct data type for `closure` is `Closure`. This lets me prepare a more static method signature of `def findPrimes(List list, Closure closure)`

One last round at this one - this time to set a return value from the `findPrimes` method. The code is not really different to the previous example but it throws a number of items together: a typed method signature, a closure parameter, and a return value for the method (the list of primes).

```
List findPrimes(List list, Closure closure) {
    def result = []
    for (num in list) {
        def isPrime = true
        if (num != 2) {
            for (divisor in (2..num/2)) {
                if (num % divisor == 0) {
                    isPrime = false
                    break
                }
            }
        }
        if (isPrime) {
            result << num
            closure(num)
        }
    }
    return result
}

def primes = findPrimes 0..100, { println "I found a prime: $it" }
```

68. Loops and closures

Some collection methods that accept a closure look like a loop but you do need to take a little care.

A number of Groovy types (such as lists and maps) support methods like `each` which call a closure on each member of a collection. These are similar to loops (`for` and `while`) but each call to the closure is discreet and you can't use the `break` branching statement to exit as you would with a loop. However, you can use `return` as somewhat of a proxy for `continue`.

In this chapter we'll take a look at a number of these methods.

each

We looked at `each` in the previous chapter but let's do one more. In the example below I determine the average of a list of numbers and then use the `each` method to tell us something about each number's relationship to the average:

```
def scores = [2, 4, 6, 8, 10]
def total = 0
def average = 0

for (i in scores) {
    total += i
}

average = total / scores.size

println "The average of the scores ($scores) is $average"

scores.each {
    print "The value $it is "

    if (it > average) {
       println 'above average'
    } else if (it < average) {
        println 'below average'
    } else {
        println 'average'
    }
}
```

collect

The collect method calls a closure for each item in a list and returns a list of the results from each closure call. This next example takes a list of test scores and uses the closure's algorithm for returning a list of grades:

```
def grades = [45, 70, 95, 51].collect {
    switch(it) {
        case (90..100):
            'A'
            break
        case (70..89):
            'B'
            break
        case (50..69):
            'C'
            break
        default:
            'F'
    }
}

println grades
```

sort

The `sort` method, when called on a list, will use the closure to evaluate a sorting algorithm and alter the list to reflect its sorted form.

In the next example I provide a very basic sorting closure - one that just returns the length of the string it's passed. This means that the `sort` method will return a list with the shortest string first:

```
assert ['cat', 'rabbit', 'ox'].sort{it.length()} == ['ox', 'cat', 'rabbit']
```

In this use of `sort` the closure accepts a single parameter and returns a numerical value. `sort` uses this result to determine the list item's new place in the sorted list. A string of length `2` will be placed at an earlier index to a string with a length of `6`.

It's important to remember that the list is changed by the `sort` method - the next example highlights this as the `animals` variable is different after `sort` is called.

```
def animals = ['cat', 'rabbit', 'ox']
animals.sort{it.length()}
assert animals == ['ox', 'cat', 'rabbit']
```

When the `sort` method is passed a closure that accepts two parameters then it works through the list by comparing neighbours. As sort changes the list, these neighbours change, resulting in the closure undertaking a number of comparisons, at least equal to the number of list items. This is powerful stuff so let's look at a sorting closure I used when discussing Relational Operators:

```
def nums = [42, -99, 6.3, 1, 612, 1, -128, 28, 0]

//Ascending
nums.sort{n1, n2 -> n1<=>n2 }

assert nums == [-128, -99, 0, 1, 1, 6.3, 28, 42, 612]
```

69. Useful Methods That Use Closures

Groovy has a number of ready to go uses for closures

Groovy adds a number of methods to `java.lang.Object` that gives you the power of closures baked into many of the data types you'll use on a daily basis.

any

The `any` method is passed a closure that evaluates a criteria (condition). If any element in the collection meets that criteria, `true` is returned. Importantly, `any` will only iterate through the list until it the criteria is met.

```
def scores = [10, 8, 11, 6]
assert scores.any {it > 10} == true
```

`find` and `findAll`

The `find` method locates the first item in a collection that meets the criteria set in the closure:

```
def animals = ['dog', 'rat', 'cat', 'mouse']
assert animals.find {it in ['rat', 'mouse', 'wild pig'] } == 'rat'
```

The `findAll` method is similar to `find` but returns all collection items that meet the criteria set by the closure:

```
def animals = ['dog', 'rat', 'cat', 'mouse']
assert animals.findAll {it in ['rat', 'mouse', 'wild pig'] } == ['rat', 'mouse']
```

`split`

The `split` method splits a collection into two lists: the first list contains all items that meet the criteria set by the closure parameter and the second list contains all remaining items.

In the example below I use the `split` method to get a list of those who got a score over 100 and those that didn't.

```
def players = [
    [name: 'Fred', topScore: 120],
    [name: 'Sally', topScore: 200],
    [name: 'Felix', topScore: 101],
    [name: 'Albert', topScore: 12],
    [name: 'Jane', topScore: 20]
]

def result = players.split {it.topScore > 100}

result[0].each {println "$it.name is in the hall of fame"}

result[1].each {println "$it.name missed out"}
```

`with`

The `with` method provides a handy approach to calling several methods and manipulating fields for an object.

In the example below I use `with` to perform a set of operations on an instance of the `Person` class:

```
class Person {
    def name
    def email
    def mobile

    def printBusinessCard(){
        println "$name"
        println "e: $email"
        println "m: $mobile"
    }
}

def john = new Person()

john.with {
    name = 'John Smith'
    email = 'john@example.com'
    mobile = '0401 999 888'
    printBusinessCard()
}
```

This approach can be really useful when creating an object. In the snippet below I create a new `Person` and set up their details at the same time:

```
def stacey = new Person().with {
    name = 'Stacey Jane'
    email = 'stacy@example.com'
    mobile = '0401 333 666'
    return it
}
stacey.printBusinessCard()
```

70. Closure scope

Closures can call into the context from which they were created.

In the first chapter on closures I mentioned that: Closures represent a reference to a function (method) that is accompanied by a referencing environment. Up to now we've mainly used closures as methods that can be referenced by variables or passed as parameters. Methods declared in classes can access the fields in their class and closures can also access items such as variables available in the context from which they were created.

I've tried to demonstrate this concept in the example below. You'll notice that my basic closure refers to a variable `code` that isn't a declared parameter of the closure nor is it a variable declared within the closure. Rather, `code` references a variable available within the scope in which the closure is declared - i.e. `code` can be accessed by the `doubleCross` closure as both are declared in the same block.

A small example of context

```
def spyMethod(closure) {
    closure()
}

def code = 'eagle'

//This is the closure
def doubleCross = {println "The code is $code"}

spyMethod(doubleCross)

code = 'badger'
spyMethod(doubleCross)
```

A More Involved Example

My `doubleCross` was quite basic and perhaps makes the usefulness of this concept appear to be ho-hum. In fact, it's extremely useful and opens the door to techniques such as Domain Specific

Languages - something for you to research later. For now, I'll take you through a step-by-step example of a more complex use of closure context.

First of all I'll create a very basic class to describe an address book Contact. For each contact I'll record their name and phone number. I'll also define a closure named `call` and this is a simple function that I can use when calling the contact.

```
class Contact {
    def name
    def phone

    def final call = { message ->
        println "Calling ${this.name} on ${this.phone} with the message: '$messa\
ge'"
    }
}
```

Final?

You may notice the `final` I used in the `call` definition. This keyword lets me set `call` to being a constant - it can't be changed after it's been defined

Once I've defined the `Contact` class I'll create a couple of contacts - Andrew and Sally.

```
def andrew = new Contact(name: 'Andrew', phone: '+61 432 321 736')
def sally = new Contact(name: 'Sally', phone: '+61 400 800 900')
```

I'll then create a list of Contacts I need to call back (perhaps my mobile phone battery died) and add Andrew and Sally. You'll note that I don't add the Contact instances to the call-back list, I actually add the closure for each contact. Think of this as the statement "I'll add to the list the activity of calling the contact".

```
def needToCallBack = []
needToCallBack << andrew.call
needToCallBack << sally.call
```

Imagine that a whole lot of other code now "does stuff" and eventually we discard the `andrew` and `sally` variables:

```
andrew = null
sally = null
```

Now this is where it gets interesting. What if my mobile battery is now recharged and I want to start calling people back? Surely the fact that I blew away my `andrew` and `sally` variables would make this impossible? Well, remember how my `needToCallBack` list contains references to the `call` closures? These closures actually hang on to their original context - the `andrew` and `sally` instances.

All of this means that I can now use the `each` method on `needToCallBack` and I can realise those closures:

```
needToCallBack.each { it('It\'s Duncan here, sorry I missed your call') }
```

This will now cause each of the `call` closures to be called - allowing me to get back in touch with `andrew` and `sally`.

I broke up the code in the discussion so present it all here in full for you copy and try out for yourself:

The Complete Script

```
class Contact {
    def name
    def phone

    //A closure for calling the contact
    def final call = { message ->
        println "Calling ${this.name} on ${this.phone} with the message: '$messa\
ge'"
    }
}

def andrew = new Contact(name: 'Andrew', phone: '+61 432 321 736')
def sally = new Contact(name: 'Sally', phone: '+61 400 800 900')

def needToCallBack = []
needToCallBack << andrew.call
needToCallBack << sally.call

// Lots of other code ......

andrew = null
sally = null

needToCallBack.each { it('It\'s Duncan here, sorry I missed your call') }
```

VIII Object-oriented programming

71. Introduction

ℹ At its core, Groovy is an object-oriented programming language.

This section introduces a range of object-oriented terminology and provides you with a set of resources that will help you really hone those object-oriented programming skills.

Object-oriented programming is often referred to by its acronym "OO" or "OOP" and has been around for some time. We won't delve into history here - check out Wikipedia[1] if you'd like to learn more. For now, let's describe the essentials of OOP through a Groovy lens.

The object-oriented programming approach models concepts based on a combination of *properties* and *methods*. The concept being modelled may be a real-world entity such as a person, a car or a tree. It may also be a "virtual" concept such as a bank account, an HTTP server or a sales report.

The properties of an object (also referred to as fields) hold data that is important to the object's *state*.

Methods (less often referred to as functions) provide a means of accessing and manipulating the object's properties and behaviours. *Encapsulation* is an important concept in OOP and relates to the use of methods to hide and even protect an object's properties from direct access. Essentially, other code interacts with the object via its methods - often referred to as its interface.

[1]http://en.wikipedia.org/wiki/Object-oriented_programming

72. Expando

The Expando class lets you create objects on the fly.

Before getting stuck into formal object definitions, let's take a look at the `Expando` class that ships with Groovy. Expando provides a very flexible way to construct an object on the fly.

In the example below the `monster` variable is dynamically assigned a set of fields and methods (through the use of closures).

The Expando Monster

```
def monster = new Expando()

//Let's add some properties
monster.name = 'Dr Frankensteins Monster'
monster.owner = 'Dr Frankenstein'
monster.height = 8

//Add some closures (methods)
monster.getOwner = {owner}
monster.getName = {name}
monster.getHeight = {height}

monster.scareVillage = { village ->
    println "Look out $village here comes $name!"
}

monster.scareVillage('Darmstadt')
```

The `monster` Expando looks to be acting in a manner similar to a Map but it gives us more functionality. Whilst the properties can be assigned to an Expando instance in the same way as a Map, the addition of a closure illustrates the difference. In the example below the use of a Map to define the `vampire` variable demonstrates that the `scareVillage` can't rely on an instance field (`name`) so must refer back to the `vampire` map:

The Vampire Map

```
def vampire = [:]
vampire.name = 'Dracula'

vampire.scareVillage = { village ->
    println "Look out $village here comes $vampire.name!"
}

vampire.scareVillage('London')
```

By using the `Expando` class instead of a `Map`, the instance fields can be accessed.

Using Expando with CSV data

Expando can be useful when we want to consume data from a source (such as a file) and manipulate it as an object. One such example is an application that reads a comma-separated file (a CSV) that contains a header row and multiple subsequent data rows. The header rows tell us the fields in our data objects and we can start to throw in some methods to help handle our data.

I'll start with a full listing of a script that accepts CSV data (I'll just use a String but it could be from a file) about various contacts. Expando will help me then produce a vCard[1] for each contact. Take a read of the full listing and we'll then break it down into easier chunks.

The full CSV example

```
//This is a basic CSV (Comma-separated file)  described as a string
def table = '''\
name,email,phone
Fred,fred@email.com,555-678345
Alex,alex@email.com,555-987123
Simon,simon@example.com,555-567321
'''

//Read the CSV in and break it up by linebreaks
def csv = table.tokenize()

//Extract the first row - it contains the headers (field names)
header = csv[0].tokenize(',')
csv.remove(0)
```

[1]https://en.wikipedia.org/wiki/VCard

```
def contactList = []

for (row in csv) {

    Expando contact = new Expando()

    //Now setup the contact class with field names based on the header
    row.tokenize(',').eachWithIndex { key, value ->
       contact.setProperty header[value], key
    }

    contact.getVCard = {
        """BEGIN:VCARD
        VERSION:4.0
        N:$name
        EMAIL:$email
        TEL;TYPE=work,voice;VALUE=uri:tel:$phone
        END:VCARD"""
    }
    contactList << contact
}

for (contact in contactList) {
    println contact.getVCard()
}
```

The first bit of code just sets up a sample CSV data table:

```
def table = '''\
name,email,phone
Fred,fred@email.com,555-678345
Alex,alex@email.com,555-987123
Simon,simon@example.com,555-567321
'''
```

The handy `tokenize` method is used to break the CSV data into individual rows. The first row is actually a header row and we perform another `tokenize` to extract each header item (field) and then remove the header row from the data table:

```
def csv = table.tokenize()
header = csv[0].tokenize(',')
csv.remove(0)
```

A `contactList` is then defined - this will hold our list of contacts:

```
def contactList = []
```

The code now iterates through the data table. For each row, an instance of `Expando` is declared:

```
Expando contact = new Expando()
```

`tokenize` is again used to separate each field in the row. The `eachWithIndex` method passes the field value and its index into the closure's key and value parameters respectively. I use the index to work out the field name in `header` and set the value as the field from the row:

```
//Now setup the contact class with field names based on the header
row.tokenize(',').eachWithIndex { key, value ->
        contact.setProperty header[value], key
}
```

Another way to set the property

Instead of using `setProperty` I could have used Groovy's ability to use string interpolation to access a method/property: `contact."${header[value]}" = key`

The `getVCard` closure is then added to the contact - this will construct a vCard for use in other systems:

```
contact.getVCard = {
    """BEGIN:VCARD
    VERSION:4.0
    N:$name
    EMAIL:$email
    TEL;TYPE=work,voice;VALUE=uri:tel:$phone
    END:VCARD"""
}
contactList << contact
```

Once each row has been consumed we can display our `contactList`:

```
for (contact in contactList) {
    println contact.getVCard()
}
```

73. The basics of OO

Classes

Object-oriented programmers use *classes* to classify objects. As such, a class defines the fields (properties) and methods of an object.

In Groovy the `class` keyword if used when declaring a new class:

```
class Person {}
```

Classes are types

The definition of the `Person` class now provides the programmer with an new reference type.

Objects consist of fields (data) and methods for interacting with that data. For example, the code below describes a `Person` class with a `name` field and a `getName` method:

A basic class with a field and a method

```
class Person {
    def name

    def getName() {
        return this.name
    }
}
```

In order to create an instance of the `Person` class we use the `new` keyword and assign the result to a variable

```
def john = new Person(name: 'John')
println john.getName()
```

The call to `new Person(â€¦)` is actually using a special method called a “constructor”. Classes can define a variety of constructors to help with creating an instance and Groovy has some handy tricks we’ll explore later.

Instead of using the `def` keyword the variable could be declared as being of the `Person` type:

```
Person john = new Person(name: 'John')
```

Information hiding

Groovy lets you hide fields and methods from other classes through the use of the `private` and `public` keywords. This helps us improve the level of encapsulation[1] in our implementation. We'll get into this a lot more in subsequent chapters.

Instance and Class Elements

In the `getName` method you'll notice the keyword `this` is used. You can translate `this` to mean "this instance". Groovy supports instance fields and methods - those operating on a specific instance of the class. This means that your code isolates one instance from another and prevents you from inadvertently altering instances.

Static fields and methods are also supported - these apply at a class level. By preceding a field or a method with the keyword `static` we indicate that it is relevant to all instances of the class. In the example below I set the `specie` field to be static and this means I access it at the class level by referring to `Person.specie`.

Class fields use the `static` keyword

```
class Person {
    static specie = 'homo sapien'
    def name
}

println Person.specie
```

Constants

An (English) language lawyer might think that the keyword `static` indicates that the value of `specie` can't be changed. However, if you try `Person.specie = 'tubulidentata'` you'll see that we can turn Person into a different specie! This is definitely not what we want so we need to declare `specie` as a constant.

The `final` keyword precedes static/instance fields and methods to make them constant:

[1]https://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming)

Make it constant with `final`

```
class Person {
    static final specie = 'homo sapien'
    def name
}
```

Constructors

Constructors are a special type of method that is called when a new instance is created. We saw that Groovy provides a built-in constructor when we called `new Person(name: 'John')`. This takes a map in which the keys match the fields in the `Person` object.

To define a custom constructor we define a method with the same name as the class but without the `def` preceding it. The example below declares a constructor this way (`Person(name)`):

Constructing a person

```
class Person {
    def name

    Person(name) {
        this.name = name
    }

    def getName() {
        return this.name
    }
}

def john = new Person('John')
```

The default is now lost

By providing our own constructor, `new Person(name: 'John')` no longer works as it did before. In fact it will now do something rather odd - it will store a Map in John's name field!

Overloading

The overloading feature of Groovy classes allows us to create multiple versions of the same method. The parameter list for each version of the method varies to allow callers to provide either a different number of parameters (as in the example below) or with the same number of parameters but with different types. Overloading is useful but also consider using default values for parameters as this can help reduce the number of methods you need to write and maintain.

Overloading the `mean` method

```
class Math {
    static mean(num1, num2) {
        (num1 + num2) / 2
    }

    static mean(...nums) {
        nums.sum() / nums.size()
    }

}
println Math.mean(10, 20)
println Math.mean(2, 4, 6, 8)
```

Overloading constructors

Constructors are just methods so they can also be overloaded.

Interfaces

Interfaces provide a method for defining programming interfaces. Interfaces, for the most part, just define method signatures and not their implementation. Classes implement these interfaces in a manner that reflect the class's role/model/context.

The example below defines an interface named `Exercise` with a single method `run`. The `Athlete` class then implements the interface:

Implementing an interface

```
interface Exercise {
    def run(int distance)
}

class Athlete implements Exercise {
    def name
    def run(int distance) {
        println "I'm off for a ${distance}km run."
    }
}

def robert = new Athlete(name: 'Rob')
robert.run(10)
```

Inheritance

A *superclass* is one from which other classes inherit functionality. The “child” classes are referred to as being *subclasses*. A subclass inherits from a superclass through the use of the `extends` keyword.

In the code below, `StaffMember` is a subclass of `Person`. This allows `StaffMember` to access the `name` field defined in `Person`:

Inheritance in action

```
class Person {
    def name
}

class StaffMember extends Person {
    def staffID

    def getIdentification() {
        println "${this.name} - ${this.staffID}"
    }
}

def sally = new StaffMember(name: 'Sally', staffID: 765)
sally.getIdentification()
```

Unlike interfaces, superclasses can provide implemented methods and fields that subclasses can utilise. However, Superclasses can work somewhat like interfaces and their methods can be declared as `abstract` to force subclasses to provide their own implementation.

Multiple inheritance is not supported

Unlike some other OO languages (e.g. C++), Groovy does not support multiple inheritance. This means that a class cannot extend more than one superclass. However, Groovy classes can implement more than one interface. Groovy does support classes implementing multiple interfaces and traits.

Overriding methods

Subclasses can also override methods and fields implemented by superclasses. This lets subclasses provide more contextual implementations if needed. A subclass can refer directly to superclass fields and methods by using the `super` keyword.

In the example below, `StaffMember` is a subclass of `Person`. The `StaffMember` class overrides the `getName` method and prefixes a string to the `name` returned by the superclass.

Overriding methods

```
class Person {
    def name

    def getName() {
        return this.name
    }
}

class StaffMember extends Person {
    def staffID

    @Override
    def getName() {
        "Team member ${super.name}"
    }

    def getIdentification() {
        println "${this.name} - ${this.staffID}"
    }
}
```

```
def sally = new StaffMember(name: 'Sally', staffID: 765)
println sally.name
```

Stop the override

Superclass methods declared with the `final` keyword can’t be overridden.

Mark the override

You’ll notice that I put `@Override` before the `getName` method in `StaffMember`. This is a really handy annotation as it tells Groovy that the method is overriding one declared by the superclass. If I mis-spell the method name then Groovy will give me an error.

Traits

At first glance, traits can be considered as interfaces with an implementation but they offer a really useful approach to adding features or abilities to a class. A common example may be to add the flying trait to animals or vehicles. The trait may have its own fields and/or methods.

In the example below:

1. A very basic `Project` class is defined. It just stores the project name
2. An `Agile` trait provides some basic fields used to describe an agile aspect to projects. A very basic method (`startIteration`) gives us an example method to call.
3. A `Scrum` class is defined:
   1. It extends the `Project` class
   2. It implements the `Agile` trait
   3. Two Scrum-specific fields are added (`productOwner` & `scrumMaster`)
4. I can then create an instance of `Scrum` and provide it information for both the `Project/Scrum` hierarchy as well as the `Agile` trait.
   1. I then call the `Agile` trait’s `startIteration` method and our project is away!

Applying a trait

```
class Project {
    def name
}

trait Agile {
    def iterationLength = 4
    def backlog = []
    def developmentTeam = []

    def startIteration() {
        println """\
            We're staring our $iterationLength week iteration for $name
            Team members: $developmentTeam
            Backlog: $backlog
            """
    }
}

class Scrum
    extends Project
    implements Agile {

    def productOwner
    def scrumMaster
}

def project = new Scrum().with {
    name = "Project X"
    iterationLength = 2
    productOwner = 'Charlie'
    scrumMaster = 'Bobby'
    developmentTeam = ['Dean', 'Sam']
    backlog << 'As a User I want to press buttons'
    backlog << 'As an Admin I want to lockout users'
    return it
}

project.startIteration()
```

Packages

As discussed earlier, Groovy allows programmers to group objects into `packages`. In the following snippet the `Person` class is allocated to the `myobjects` package:

Put classes into packages

```
package myobjects

class Person {}
```

Packages are central to using others' code in your programs and allowing your code to be used elsewhere.

Summary

This chapter has provided an extremely brief overview of object-oriented programming supported by small Groovy examples. The following chapters in this section will explore fundamental aspects of fields and methods and the subsequent section will dive far deeper into how to really get OO with Groovy.

Incomplete section

This section is yet to be completed. Watch this space.

74. Creating a Class

Who doesn't want a classy application?

The basics of Objects was covered as part of the section on Variables and Data Types. Let's refresh:

- A class is defined using the `class` keyword: `class MyClass{}`
- Class names use camel case format in which the first letter in each word appears in upper case.
  - Don't use underscores as proxy spaces (e.g. `My_Class`)
- A new *instance* of a class is created using the `new` keyword: `def fido = new Dog()`

In this chapter we'll delve more into the main class elements: properties, fields and methods.

Properties and Fields

A basic person class with three properties

```
class Person {
    def name
    def email
    def mobile
}


def astrid = new Person()
astrid.name = 'Astrid Smithson'
astrid.email = 'astrid@example.com'
astrid.mobile = '0418 111 222'
```

This type of class can be really handy when you just need a structure for storing and handing around data. Say for example you load a record from a database and want to feed it to other classes and their methods for processing.

```
def astrid = new Person(name: 'Astrid Smithson', email: 'astrid@example.com', mo\
bile: '0418 111 222')
```

I

```
def astrid = new Person(name: 'Astrid Smithson', email: 'astrid@example.com')
```

If you try to use `println` to display the details for `astrid` you'll get something like `Person@46423706`. It's not very useful - just the type of the Object and an identifier. If you want to find out the field values, call the `dump` method that all objects inherit from `Object`:

```
println astrid.dump()
```

class Person { String name String email String mobile }

Fields

Default values

Properties and fields are much like

Getters and Setters

```
class Person {
    def name
    def email
    def mobile
}

def jess = new Person()

jess.setEmail('jess_at_example.com')

println jess.dump()

class Person {
    def name
    def email
    def mobile
    def setEmail(email) throws IllegalArgumentException {
        def regex = ~/[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+/
        if (email.matches(regex)) {
            this.email = email
            return
        }
        throw new IllegalArgumentException('Incorrect email format')
    }
}

def jess = new Person()

jess.setEmail('jess@example.com')
println jess.dump()

//This will fail
jess.setEmail('jess_at_example.com')
println jess.dump()
```

Methods

75. Constructors

```
class Person {
    def name
    def email
    def mobile

    Person(name) {
        this.name = name
    }
}

def astrid = new Person('Astrid Smithson')

println astrid.dump()
```

Now that you've supplied a constructor you'll lose the built-in named argument constructor. Be warned that this isn't always obvious! If we create a new instance using named arguments, our `dump` will show us that `astrid`'s name becomes a list:

```
def astrid = new Person(name: 'Astrid Smithson', email: 'astrid@example.com', mo\
bile: '0418 111 222')

println astrid.dump()
```

What just happened? Remember that Groovy provides a helpful option for providing named arguments in methods. As a constructor is just a specific type of method, Groovy sees the parameters as a map and stores it in the `name` field.

Working With Getters and Setters

If you've

76. Packages

77. Access Modifiers

Classes are used to embody an abstraction of some real or virtual entity. You can probably guess what a `BankAccount` or a `EmailMessage` class embodies. In some cases we might be happy for other code to change a value held in a class instance or to run a method. However, in the case of a `BankAccount` class we are likely to need some sort of *information hiding* that moderates what other code is allowed to do:

- The `balance` field shouldn't be directly accessible. Rather, two methods `deposit` and `withdrawal` have to be used.
- The `withdrawal` method should not allow you take take out more money than the current balance.

In Groovy we can implement the business logic just described through the use of *Access Modifiers*.

Access modifiers are keywords (`public`, `private`, `protected`) that determine if another class can use an element of a class (such as a property, field or method).

- `public` elements can be accessed by all other classes and code
- `private` elements are only accessible from within the class itself
- `protected` elements

For our `BankAccount` class we can make the `deposit` field `private`:

A basic BankAccount with information hiding

```
class BankAccount {

    private balance = 0

    public withdrawal(amount) {
        if (amount > balance) {
            throw new Exception('Insufficient balance')
        }

        balance -= amount
    }

    public deposit(amount) {
        balance += amount
    }
}

def acct = new BankAccount()
acct.deposit(100)
acct.withdrawal(150)
```

In the example above I set the `balance` field to `private` and then provide two `public` methods to allow for other code to perform a `deposit` or a `withdrawal`. The latter method even throws an exception if you try to take out too much.

Now here's "the rub". Groovy doesn't actually enforce the access modifier. That means that, given my `BankAccount` class I could still write `acct.balance = 1_000_000` and access the field directly. A Python programmer might shrug at this and state that it's a matter of respecting the original programmer's intention. A Java programmer might be shocked that Groovy doesn't throw an exception or an error. I'm usually pretty happy with the Python approach but if it was really a concern I could add the following method to my `BankAccount` class:

```
private setBalance(amount){}
```

Groovy generates setters and getters for fields (such as `balance`) and properties. In the case of the `balance` field, the setter method named `setBalance` is actually called when I do something like `acct.balance = 1_000_000`. Knowing this, I overrode the setter Groovy would have created with my own version that does nothing and I also used the `private` modifier. This does two things:

1. The `private` modifier reinforces that other developers should try to directly change the value of `balance`
2. If the other developers just don't listen then I ignore their attempt to change the `balance`.

Whilst my empty `setBalance` method helps prove a point, having to do that too often will reduce readability and annoy me with having to write lots of vacant code to protect my code from others who don't listen. Essentially I take the approach that developers are (usually) sensible people that understand what they're being asked not to fiddle with. If they fiddle then they can expect a weird outcome. So feel free to write a method with the signature `private doSomethingEvil()`, add some code that deletes all files and see who doesn't pay attention :)

Access modifiers are not security

Groovy and Java both support a technique called *reflection* that allows programmers to programmatically examine other code (such as class definitions) and even change their behaviour. This means that even your Java code can be changed by another developer if they're using your code in their application. There are methods for preventing this[1] but it's beyond the scope of this book.

[1] If you want to really get into this topic, start with http://docs.oracle.com/javase/tutorial/essential/environment/security.html

Applying Access Modifiers

The following access methods are available:

- Classes:
  - `public`
  - `protected`
- Fields
  - `public`
  - `private`
  - `protected`
- Methods
  - `public`
  - `private`
  - `protected`

I'd simplified things earlier when I said that *properties* are sometimes also referred to as *fields*. The Groovy documentation discriminates between a *field* and a *property* and the difference really boils down to:

> *Fields* have an access modifier and *Properties* do not

78. Useful Annotations

ToString

```
@groovy.transform.ToString
class Person {
    def name
    def email
    def mobile

    Person(name) {
        this.name = name
    }
}

def astrid = new Person('Astrid Smithson')

println astrid
```

Instead of using the full library path with `@groovy.transform.ToString` you can use the `import` keyword:

```
import groovy.transform.ToString

@ToString
class Person {
        //...
}
```

`@ToString(includeNames=true)`

`@ToString(includeFields=true):`

```
@groovy.transform.ToString(includeNames=true, includeFields=true)
class Person {
    def name
    private email
    private mobile

    Person(name) {
        this.name = name
    }
}

def astrid = new Person('Astrid Smithson')

println astrid
```

EqualsAndHashCode

```
@groovy.transform.EqualsAndHashCode
class Person {
    def name
    def email
    def mobile
}

def agentSmith = new Person(name: 'Agent Smith')
def agentSmith2 = new Person(name: 'Agent Smith')

assert agentSmith == agentSmith2
```

TupleConstructor

```
@groovy.transform.TupleConstructor
class Person {
    def name = 'Anonymous'
    def email
    def mobile
}

def unknown = new Person()
def astrid = new Person('Astrid Smithson')
def john = new Person('John Hancock', 'john@example.com')
def kelly = new Person('Kelly Grant', 'kelly@example.com', '044 555 555')
```

Canonical

The `Canonical` annotation brings together functionality from a suite of other annotations:

- `ToString`
- `EqualsAndHashCode`
- `TupleConstructor`

  @groovy.transform.Canonical class Person { def name def email def mobile }

This saves you from needing to stack your annotations:

```
@groovy.transform.ToString
@groovy.transform.EqualsAndHashCode
@groovy.transform.TupleConstructor
class Person {
    def name
    def email
    def mobile
}
```

Immutable

The `Immutable` annotation provides the same features as `Canonical` but locks down new instances so that they can't be changed after creation.

```
@groovy.transform.Immutable
class Person {
    String name
    String email
    String mobile
}

def krusty = new Person(name: 'Krusty')
krusty.email = 'heyhey@example.com'
```

79. Let Go of That Instance!

Sometimes your variable (e.g. a class instance) can end up holding a large amount of data. For example, you might have placed the text of the complete works of Shakespeare into a property. The JVM performs a process called *Garbage Collection* so as to clean up data that you are no longer using. By assigning a variable to `null` we flag that the data previously held by the variable is no longer needed:

```
myInstance = null
```

For small Groovy scripts this may never be an issue but, for long running applications, data hanging around and not being used can start to soak up a serious amount of resources, especially memory. Once you've finished with a variable it's worth assigning its value to `null` to let the JVM know you don't need it anymore.

You don't always need to explicitly set variables to `null` - this would make your code far less readable. The JVM knows that once a variable is no longer in scope, it's no longer accessible and, thus, is no longer needed. This means that if you have a variable declared inside a method, that its value is no longer needed once the method has completed. The same goes for variables declared within scopes such as loops.

There is an important caveat however, if multiple variables refer to the same data then the JVM can only release resources once all references have "unlatched". Let's examine this in the code below:

Example of multiple references to the same data

```
class SampleText {
    def text
}

def shakespeare = new SampleText(text: 'It was the best of times....')
def marlow = shakespeare
shakespeare = null
println marlow.text
```

I've defined a variable (`shakespeare`) to hold a new instance of the `SampleText` class and then said that another variable (`marlow`) points to that instance of `SampleText`. My call to `marlow.text` will still work despite my setting `shakespeare` to `null`. In this case we say that "`marlow` still holds a reference to the `SampleText` instance". This means that the JVM can't release the resources held by the instance until all references are set to `null`. I need to set `marlow` to `null` to completely release the resources.

Keeping track of references

Building up unused references to data is how a program will use up all of your system's memory and slowing everything to a crawl over time. When you have variables or collections of variables (such as lists) that you keep and add to over a long period of time you need to make sure you clean up what you don't need.

There's a lot more to garbage collection than I want to cover here but if you spend a few moments searching "Java Garbage Collection" you'll be able to delve deep into the topic.

80. Static Methods and Variables

81. Interfaces

```
interface SocialNetwork {
    def friend(Member friend)

    def unFriend(Member enemy)
}

class Member implements SocialNetwork {
    String name
    def friends = [] as Set

    @Override
    def friend(Member friend) {
        friends << friend

    }

    @Override
    def unFriend(Member enemy) {
        friends -= enemy
    }
}

def jim = new Member(name: 'Jim')
def gina = new Member(name: 'Gina')
def andrew = new Member(name: 'Andrew')

print 'Jim makes some friends: '
jim.friend(gina)
jim.friend(andrew)
jim.friends.each{print "$it.name "}

println '\nJim makes an enemy'
jim.unFriend(andrew)

print 'Jim now has these friends: '
jim.friends.each{print "$it.name "}
```

82. Traits

83. Inheritance

Abstract classes

84. Enums

85. Other OO Topics

Static Type Checking and Compilation

```
class Person {
    String id
    Integer getId() {id}
}

def pete = new Person(id: 12.3)
println pete.id
```

When you try to run the code above you'll get a Groovy runtime exception (`org.codehaus.groovy.runtime.typehar`

```
@groovy.transform.TypeChecked
class Person {
    String id
    Integer getId() {id}
}

def pete = new Person(id: 12.3)
println pete.id
```

Generics

Note how Groovy just lets us pass in the String:

```
public class Box<T> {
    private T value
    public void set(T value) { this.value = value }
    public T get() { value }
}

Box<Integer> iBox = new Box<>()
iBox.set('rabbit')

println iBox.get()
```

Java won't compile code equivalent to that above.

Inner Classes

Java developers

IX Going further

86. The great beyond

That covers most (not all) of the Groovy syntax. My goal was to introduce you to the “core” syntax of Groovy so that you can start programming with a good working knowledge in the language. From this point I hope you’ll find this book and the Groovy documentation[1] an essential companion to your efforts.

There are further Groovy features you may like to use in your projects:

- Develop a domain specific language[2]
- Try your hand at metaprogramming[3]
- Utilise various Groovy modules:
    - JSON[4] (JavaScript Object Notation)
    - XML[5]
    - Templates[6]

The Groovy Documentation

At time of writing, the Groovy Documentation still had many gaps. Don’t let this put you off as you’ll usually be able to find your answer with a web search or through the mailing lists[7].

As I mentioned at the very beginning of this book, Groovy in Action (2nd Edition)[8] is also a great reference for those wanting to go further.

Build large applications

Gradle[9] is a build automation tool that should be your go-to when building non-trivial programs. In fact, I would suggest that checking out Gradle is a great next-step after reading this book.

For those coming from the Java world, Gradle is an excellent replacement for Apache Maven[10] and Apache Ant[11].

[1]http://www.groovy-lang.org/documentation.html
[2]http://www.groovy-lang.org/dsls.html
[3]http://www.groovy-lang.org/metaprogramming.html
[4]http://www.groovy-lang.org/json.html
[5]http://www.groovy-lang.org/processing-xml.html
[6]http://www.groovy-lang.org/processing-xml.html
[7]http://www.groovy-lang.org/mailing-lists.html
[8]http://www.manning.com/koenig2/
[9]https://www.gradle.org/
[10]http://maven.apache.org/
[11]http://ant.apache.org/

Use the Groovy ecosystem

There are several high-quality projects that use Groovy[12], it's worth checking them out:

- Grails[13] - a full-stack web application framework for the Java Virtual Machine
  - That means it's a great tool-set for building web applications
- Griffon[14] - a desktop application toolkit
- Spock[15] - a testing framework
- CodeNarc[16] - a code analysis tool for Groovy

Whilst it's not written in Groovy, the Spring Boot[17] project is worth a look as you can use Groovy to quickly write some nifty applications using the Spring framework.

[12]http://www.groovy-lang.org/ecosystem.html
[13]https://grails.org/
[14]http://new.griffon-framework.org/
[15]https://code.google.com/p/spock/
[16]http://codenarc.sourceforge.net/
[17]http://projects.spring.io/spring-boot/

Colophon

I wish I had a 19th century engraving on the cover so that I could tell you about a really cool animal. If I did I would use the Pied butcherbird[18], perhaps accompanied with a picture from one of John Gould's[19] books. I would then tell you that this small-ish bird has a beautiful song and a really friendly composure. My resident (wild) butcherbirds like to sing on my deck when it's raining, follow me when I mow and, occasionally, bathe under the sprinkler on hot days.

[18]http://en.wikipedia.org/wiki/Pied_butcherbird
[19]http://en.wikipedia.org/wiki/John_Gould