The Groovy 2 Tutorial

An introductory text for budding Groovy developers

Duncan Dickinson

The Groovy 2 Tutorial

An introductory text for Groovy developers

Duncan Dickinson

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

This version was published on 2016-05-03

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 - #groovytutorial #groovylang

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**
Something Wrong? ii
Bookmarks ii
Conventions Used in This Book iii
Your Rights and This Book iv

Legal Notices . . . . . **vi**

I Getting started 1

1. Introduction **2**

2. Installing Groovy **3**
Don't Install Groovy 3
Install a Java Virtual Machine 3
Install Groovy 4

3. Your first lines of Groovy **7**
Examining the script 8

4. Running a script **10**

5. Compiling Groovy **12**

6. Comments **13**
Single-line comments 13
Multi-line comments 13
Usage to avoid 14
Groovydoc 15

7. Statements **16**
Usage to avoid 17

8. The `assert` statement **18**
Handling failed assertions 19

9. Reserved Words **21**

10. Packages **22**
Using `import` 23
Built in Libraries 24
Useful third-party libraries 25

11. Grape **26**

II Variables 28

12. Introduction **29**

13. Declaring Variables **30**
Variable names 32
Data Types 32

14. Objects **34**
Declaring and using a class 34
Useful Methods 38
Existing classes 39
Classes and literal values 41

15. Booleans **43**
Useful Methods 43

16. Numbers **44**
Integers 44
Decimals 44
Scientific notation 45
Number Systems 46
Useful Methods and Properties 46

17. Strings **48**
Escape sequences 48
GStrings 50
Multiline Strings 51
Building Strings 52
Useful Methods 52

18. Collections **55**
Lists 55
Maps 59

19. Arrays **64**

Manipulating arrays . . . 66

20. Ranges . . . 68
Half-Open Ranges . . . 69
Ranges of Objects . . . 69
Ranges and List Indexes . . . 71
Ranges and Loops . . . 71
Useful Methods . . . 72

21. Regular Expressions . . . 75
Regular Expression Syntax . . . 76
Useful Methods . . . 78

22. Data types . . . 82
Groovy's use of types . . . 82
Using a specific type . . . 83
The `null` Value . . . 84
Available data types . . . 84
Type Conversions . . . 85

III Operators . . . 88

23. Introduction . . . 89
Arithmetic and Conditional Operators . . . 89
String Operators . . . 89
Regular Expression Operators . . . 90
Collection Operators . . . 90
Object Operators . . . 90

IV Control Flow Statements . . . 91

24. Introduction . . . 92

V Exceptions . . . 93

25. Introduction . . . 94

VI Methods . . . 95

26. Introduction . . . 96

VII Closures 97

27. Introduction **98**

VIII Object-oriented programming 99

28. Introduction **100**

IX More object-oriented programming 101

29. Introduction **102**
The Shapes demo 102

X Going further 103

30. The great beyond **104**
Build large applications 104
Use the Groovy ecosystem 105

Colophon **106**

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 sections 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.

Enjoy!

Duncan (@groovytutorial[4])

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/

[4] https://twitter.com/groovytutorial

Something Wrong?

If you find something wrong I'd really appreciate you letting me know via the GitHub issue system[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.

I've been establishing www.groovy-tutorial.org[12] to supplement this book with practical Groovy tutorials. Keep an eye out for new content!

Great blogs and sites that provide Groovy coding examples:

- Mr Haki[13] - A really useful site containing heaps of Groovy code examples.
- InfoQ[14] - Another useful site for Groovy articles.

The book Groovy in Action (2nd Edition)[15] 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:

[5] https://github.com/groovy-tutorial/groovy-tutorial/issues
[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/8/docs/api/
[10] http://docs.oracle.com/javase/tutorial/
[11] http://google-styleguide.googlecode.com/svn/trunk/javaguide.html
[12] http://www.groovy-tutorial.org/
[13] http://mrhaki.blogspot.com.au/search/label/Groovy
[14] http://www.infoq.com/groovy
[15] http://www.manning.com/koenig2/

- StackOverflow's Groovy tag[16] is really useful
  - search for an answer before posting a question
  - Check out their article "How do I ask a good question?[17]" - it's an excellent outline for asking in a way people may answer
- The Groovy Mailing lists[18] 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:

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.

The book formatting/layout process can cause code to be broken over to another line. In these cases a backslash (\) is added at the end of the first line:

[16] http://stackoverflow.com/questions/tagged/groovy
[17] http://stackoverflow.com/help/how-to-ask
[18] http://groovy-lang.org/mailing-lists.html

```
def x = 10 \
+ 1
```

Unfortunately this syntax isn't always Groovy-compliant and can cause the code to fail. If a code listing doesn't seem to work your best bet is to remove the backslash and newline.

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.

All code samples are licensed under the Apache License, Version 2.0[19]. If you would like to browse a subset of the code examples used in this book you'll find them in the GitHub repository[20].

The "source" for this book is written in Markdown, based on the LeanPub Manual[21]. You can access the source from my GitHub 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 :)

[19] https://www.apache.org/licenses/LICENSE-2.0.html
[20] https://github.com/groovy-tutorial/groovy-tutorial/tree/master/manuscript/code
[21] https://leanpub.com/help/manual
[22] https://github.com/groovy-tutorial/groovy-tutorial
[23] ... or pekignese

Legal Notices

Acknowledgement of trademarks:

- Java™ is a registered trademark of Oracle and/or its affiliates.
- Apple® and OS X® are trademarks of Apple Inc., registered in the U.S. and other countries
- Microsoft® and Windows® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.
- Spring IO® and Grails™ are trademarks/service marks of Pivotal Software, Inc. and its subsidiaries and affiliates
- Gradle™ is a trademark of Gradle, Inc. in the United States and/or other jurisdictions.
- Android™ is a trademark of Google Inc.
- Git™ is a trademark of the Software Freedom Conservancy.

None of the companies listed above endorse this book.

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 book. Provided you have web access you can skip through to the next chapter and get straight into writing some code!

However, please note that the web console has some limitations and won't run code that attempts to access URLs and files.

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).

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

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

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 (1.8).

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.

[3] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[4] http://openjdk.java.net/
[5] http://groovy-lang.org/download.html

Mac OSX and Linux

SDKMAN! is the best tool for getting Groovy running on your system. The homepage is http://sdkman.io/ but you don't need to visit it to run an installation.

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.sdkman.io | bash
```

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

```
sdk 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. 2.4.4). To install Groovy you now just call `sdk` as below:

```
sdk install groovy 2.4.4
```

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:

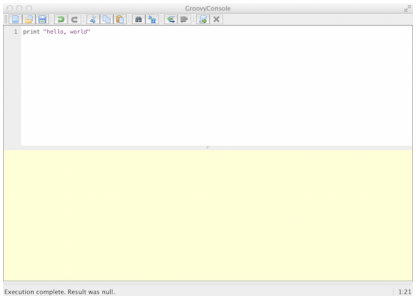
Start the console

```
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:



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:

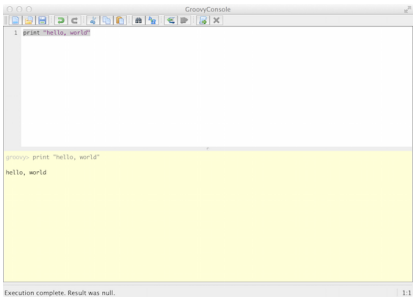
Let's get groovy

```
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:



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.

Groovy has a flexible syntax and can accept either `print 'hello, world'` or `print('hello, world')`

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

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:

Hello,world - Java style

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

Running java

To run the code above we'd need to run the following from the command line:

```
javac Hello.java
java -cp . Hello
```

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.

We'll explore this further in the Organising your code chapter.

6. Comments

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

Comments are not 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.

[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 colon (`:`):

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()
```

The section on Exceptions will explain this syntax in more depth.

Annotations

You may have noticed `@Log` in that last code snippet. This is called an annotation[1] and is marked using an "at" (@) sign followed by a class name. These are used to markup code and have a number of purposes including error detection and generating functionality.

[1] https://docs.oracle.com/javase/tutorial/java/annotations/

Although it's Java-focussed, check out the Programming with Assertions guide[2] for more information.

[2] 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 Organising your code chapter.
- 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 puts 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[5].

`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

[5] We'll describe how these are written in the Class Methods and Variables chapter.

`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[6]
2. Google Guava[7]

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[8] came along to make it easier to grab a copy of your dependencies. This then lead to The (Maven) Central Repository[9] 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.

[6] http://commons.apache.org/
[7] https://code.google.com/p/guava-libraries/
[8] http://maven.apache.org
[9] http://search.maven.org

11. Grape

Whilst you can use Maven or (even better) Gradle[1] to grab dependencies, Groovy includes a dependency manager called Grape[2] 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[3] would really help me. I can search the Maven Central Repository and find what I need[4]. In fact, that web page even tells me how to use the library with Grape:

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:

[1] http://gradle.org
[2] http://groovy-lang.org/grape.html
[3] https://hc.apache.org/index.html
[4] http://search.maven.org/#artifactdetails%7Corg.apache.httpcomponents%7Chttpclient%7C4.3.6%7Cjar

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

12. 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.

13. 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.

When writing a Groovy script (such as in the Groovy Console) you may notice that you don't need to explicitly declare a variable before using it - this is due to how scripts are run by Groovy. The Groovy documentation[1] provides more detail as to how/why this is.

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`):

[1] http://docs.groovy-lang.org/latest/html/documentation/index.html#_variables

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.

14. 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** and **Fields** are variables that hold data about the object
  - For example, a person object may have properties such as `name` and `email`
  - There is a difference between Properties and Fields but we'll look into that later.
- **Methods** are a means for accessing and manipulating the object's properties
  - For example a person object may have methods such as `getName()` and `setName(name)`
  - Methods can take parameters and/or return values. For example: `getName()` would return the person's name; and `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 use 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

```
class Person {
    def name
    def email
}

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

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

Existing classes

The great strength/benefit/bonus of an object-oriented programming platform such as Java 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

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

//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
```

15. 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`

16. 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. 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 that 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
```

17. Strings

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

GStrings let us escape Java's 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 `\n` all over the place. 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')
```

The `replace` method 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.

Lastly, and a favourite of mine, is `toURL()`. This converts a String to a `URL` object which, in Groovy has a great `text` property that lets us load the text of our favourite web page:

```
println 'http://www.example.com/'.toURL().text
```

18. 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]
```

Ranges can also be used in the subscript notation and, as demonstrated in the example below, return a list containing the items whose 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'
```

19. 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
  - `new Number[2]` sets `point` to be an empty array that can contain two (2) elements of the `Number` class (or a subtype thereof).
  - Don't use `def` as we're specifying the data type
- 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.

20. Ranges

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 on 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"
}
```

21. 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 section on 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.

22. 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	`234231`
`F` or `f`	Float	`3.1415f`
`D` or `d`	Double	`3.1415d`
`G` or `g`	BigInteger	`1_000_000g`
`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 3.1415926535f == 3.1415926535d
```

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 3.1415926535.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.

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 Object 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

23. 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`, `in`	Type Comparison

IV Control Flow Statements

24. 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 section.

The `return` statement will be explored in the Methods section and the Closures section.

V Exceptions

25. 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

VI Methods

26. Introduction

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.

VII Closures

27. 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"

VIII Object-oriented programming

28. Introduction

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

IX More object-oriented programming

29. Introduction

Now that you have a handle on constructing classes we'll explore the various Groovy constructs for creating a more extensible codebase:

- Interfaces: define a set of method signatures that are then implemented by a class (or classes).
- Traits: add abilities to a class
- Inheritance: Allows a class to inherit the functionality of another class and extend it.
  - When `ChildClass` inherits from `ParentClass`, we say that `ChildClass` is a *subclass* of `ParentClass` and that `ParentClass` is a *superclass* of `ChildClass`[1].

First up, we'll look at how to organise your code before those classes get out of hand.

The Shapes demo

Throughout this section I'll build up an example library for working with two-dimensional shapes. You'll see this in chapters with the **The shapes demo** prefix and a full listing is available as an appendix.

As the Shapes demo source code is laid out as a larger project and divided into packages, you won't be able to run it via `groovyConsole`. To support you in trying out the demo I've setup the shapes demo[2] mini site. This provides a number of handy resources:

- A guide to building (compiling) the code both directly using `groovyc` and with the Gradle build tool[3].
- Links to download the code
- Various reports on the code

Once you've read through the chapters in this section, head to the shapes demo[4] mini site and give it a try.

[1] See: Wikipedia: Inheritance (object-oriented programming)
[2] http://www.groovy-tutorial.org/shapes-demo/
[3] http://gradle.org/
[4] http://www.groovy-tutorial.org/shapes-demo/

X Going further

30. 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] to be essential companions in your Groovy programming 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 has gaps but is coming together very nicely. 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.

[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/

For those coming from the Java world, Gradle is an excellent replacement for Apache Maven[10] and Apache Ant[11].

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.

[10] http://maven.apache.org/
[11] http://ant.apache.org/
[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