

C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Groovy for Domain-Specific Languages

Extend and enhance your Java applications with Domain-Specific Languages in Groovy

Fergal Dearle

[PACKT] open source*
PUBLISHING

community experience distilled

Groovy for Domain-Specific Languages

Extend and enhance your Java applications with Domain-Specific Languages in Groovy

Fergal Dearle



open source community experience distilled

BIRMINGHAM - MUMBAI

Groovy for Domain-Specific Languages

Copyright © 2010 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, Packt Publishing, nor its dealers or distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2010

Production Reference: 1210510

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847196-90-3

www.packtpub.com

Cover Image by Karl Moore (karl.moore@ukonline.co.uk)

Credits

Author

Fergal Dearle

Editorial Team Leader

Akshara Aware

Reviewer

Chanwit Kaewkasi

Guillaume Laforge

Robert F. Castellow

Project Team Leader

Lata Basantani

Project Coordinator

Shubhanjan Chatterjee

Acquisition Editor

Douglas Paterson

Proofreader

Dirk Manuel

Development Editor

Rakesh Sejwal

Graphics

Geetanjali Sawant

Technical Editor

Shadab Khan

Production Coordinator

Aparna Bhagat

Copy Editor

Lakshmi Menon

Cover Work

Aparna Bhagat

Indexer

Rekha Nair

Monica Ajmera

About the Author

Fergal Dearle is a seasoned software development professional with 23 years of experience in software product development across a wide variety of technologies. He is currently the principal consultant with his own software development consulting company, Dearle Technologies Ltd., engaged in design, development, and architecture for new software products for client companies. In the past, Fergal has worked in lead architect and developer roles for Candle Corporation on the OMEGAMON product, which is now part of IBM's Tivoli product suite, and as development manager for the Unix implementations of Lotus 1-2-3. In the early 1990s, Fergal led the team at Glockenspiel that developed CommonView, the first object-oriented UI framework for Microsoft Windows. The team was awarded one of the first ever Jolt Productivity Awards by Dr Dobbs Journal.

I would like to thank the team at Packt for their patience and forbearance during the writing of this book. With special thanks to Doug, Rajashree, Rakesh, and Shubhanjan for keeping me on the straight and narrow. I would also like to thank David Petherick, "The Digital Biographer" for introducing me to the wonders of cask strength malt Scotch whiskey and for his words of wisdom and encouragement both of which gave me the courage to take on this project in the first place. Finally, but by no means least, I want to thank my girls Caroline, Eabha, Nessa, and Sadhbh who put up with my long absences during the writing, without whose love and tolerance I would never have completed this book.

About the Reviewers

Chanwit Kaewkasi is a lecturer at the School of Computer Engineering, Suranaree University, Thailand, where he received his BEng (First-class honours) in Computer Engineering. He received MEng in computer engineering from Chulalongkorn University, Thailand. Currently, he is doing a PhD in computer science at the University of Manchester, United Kingdom. His research involves an aspect-oriented approach to performance improvement of a dynamic language.

Chanwit has been an active contributor to the Groovy and Grails community since 2006. He was the very first committer of the Grails plugin project, where he maintains two of them—XFire and ZK plugin for Grails. In 2008, he joined the Google Summer of Code program for the development of a just-in-time compiler for Groovy.

Guillaume Laforge is the project lead of Groovy, the highly popular and successful dynamic language for the JVM. He co-authored Manning's bestseller "Groovy in Action" with Dierk König, and is working for SpringSource (a division of VMWare) where he's working full time on cool and Groovy stuff. You can meet Guillaume at conferences around the world where he evangelizes the Groovy dynamic language, Domain-Specific Languages in Groovy, and the agile Grails web framework.

Rob F. Castellow is the president of PAC Enterprises LLC, a contract and development company responsible for providing quality professional services. He has provided services in the development of several J2EE-based projects for large corporations in the telecommunications and financial services sectors.

Rob graduated in 1998 with a Masters in Electrical Engineering from the Georgia Institute of Technology, and began his career developing embedded systems. Rob soon found that all the excitement was in developing enterprise systems and has been working on J2EE-based applications ever since.

Rob is an enthusiast of new technologies. When he is not proofreading books on Groovy DSLs, he can be found developing Grails applications, attending user groups, reading books, managing/developing several websites, or playing with his two sons.

Table of Contents

Preface	1
Chapter 1: Introduction to DSL and Groovy	7
DSL: New name for an old idea	8
The evolution of programming languages	9
General-purpose languages	9
Spreadsheets and 4GLs	10
Language-oriented programming	11
Who are DSLs for?	11
A DSL for process engineers	12
Stakeholder participation	13
DSL design and implementation	14
External versus internal DSL	14
Operator overloading	15
Groovy	17
A natural fit with the JVM	17
Groovy language features	18
Static and optional typing	18
Native support for lists and maps	18
Closures	19
Groovy operator overloading	20
Regular expression support	20
Optional syntax	20
Groovy markup	21
Summary	24
Chapter 2: Groovy Quick Start	25
How to find and install Groovy	25
Running Groovy	26
The Groovy script engine—groovy	27
Shebang scripts	29
The Groovy shell: groovysh	30

The Groovy console: groovyConsole	32
The Groovy compiler: groovyc	33
Groovy IDE and editor integration	33
Netbeans	33
Eclipse	34
IntelliJ IDEA	34
Other IDEs and editors	34
Introducing the Groovy Language	34
Module structure	34
Groovy shorthand	36
Assumed imports	36
Default visibility, optional semicolon	37
Optional parentheses and types	37
Optional return keyword	39
Assertions	41
Autoboxing	42
Strings	43
Regular expressions	43
Methods and closures	46
Control structures	49
Groovy truth	49
Ternary and Elvis operators	50
Switch statement	51
Loops	52
Collections	53
Ranges	53
Lists	54
Maps	55
Operators	57
Spread and spread-dot	57
Null safe dereference	58
Operator overloading	59
Summary	59
Chapter 3: Groovy Closures	61
What is a closure	62
Closures and collection methods	63
Closures as method parameters	64
Method parameters as DSL	65
Forwarding parameters	66
Calling closures	68
Finding a named closure field	70
Closure parameters	71
Parameters and the doCall method	72

Passing multiple parameters	74
Enforcing zero parameters	75
Default parameter values	75
Curried parameters	76
Closure return values	78
Closure scope	78
this, owner, and delegate	81
Summary	83
Chapter 4: Example DSL: GeeTwitter	85
Twitter	85
Working with the Twitter APIs	86
Using Twitter4J Java APIs	88
Tweeting	88
Direct messages	89
Searching	90
Following	92
Groovy improvements	93
A Groovier way to find friends	93
Groovy searching	95
Removing boilerplate	96
Refactoring	97
Fleshing out GeeTwitter	100
Improving search	102
Adding a command-line interface	105
Adding built-in methods	106
Summary	111
Chapter 5: Power Groovy DSL features	113
Named parameters	114
Named parameters in DSLs	116
Builders	117
Builder design pattern	117
Using Groovy Builders	119
MarkupBuilder	120
Namespaced XML	121
GroovyMarkup and the builder design pattern	125
Using program logic with builders	128
Builders for every occasion	129
NodeBuilder	130
SwingBuilder	132
Griffon: Builders as DSL	135
Method pointers	136

Metaprogramming and the Groovy MOP	137
Reflection	137
Groovy Reflection shortcuts	139
Expando	140
Categories	142
MetaClass	144
Pretended methods (<code>MetaClass.invokeMethod</code>)	145
Understanding this, delegate, and owner	146
How Builders work	149
ExpandoMetaClasses	151
Replacing methods	152
Adding or overriding static methods	153
Dynamic method naming	153
Adding overloaded methods	154
Adding constructors	155
Summary	156
Chapter 6: Existing Groovy DSLs	157
The Grails Object Relational Mapping (GORM)	158
Grails quick start	158
The grails-app directory	159
DataSource configuration	159
Building a GORM model	161
Using domain classes	163
Modeling relationships	165
Associations	165
Composition	175
Inheritance	176
Querying	178
Dynamic finders	179
GORM as a DSL	180
Gant	180
Ant	180
AntBuilder	183
Gant and AntBuilder	184
ATDD, BDD with GSpec, EasyB, and Spock	187
GSpec	188
EasyB	191
Spock	193
Blocks	195
BDD DSL style	196
Summary	197

Chapter 7: Building a Builder	199
Builder code structure	199
Closure method calls	200
Resolve Strategy: OWNER_FIRST	203
Pretended methods	206
invokeMethod	206
methodMissing	207
Closure delegate	208
BuilderSupport	209
BuilderSupport hook methods	209
A database builder	214
FactoryBuilderSupport	220
Summary	226
Chapter 8: Implementing a Rules DSL	227
Groovy bindings	228
Exploiting bindings in DSLs	231
Closures as built-in methods	231
Closures as repeatable blocks	231
Using a specification parameter	233
Closures as singleton blocks	234
Using binding properties to form context	235
Storing and communicating results	237
Bindings and the GORM DataSource DSL	239
Building a Rewards DSL	240
Designing the DSL	240
BroadbandPlus	240
Reward types	242
The Reward DSL	242
Handling events: deferred execution	247
Convenience methods and shorthand	249
The Offers	250
The RewardService class	251
The BroadbandPlus application classes	255
Testing with GroovyTestCase	257
Summary	260
Chapter 9: Integrating it all	261
Mixing and matching Groovy and Java	261
Calling Groovy from Java	263
POJOs and POGOs	263
Calling Java from Groovy	266
Privacy concerns	267
Interfaces in Java and Groovy	268
Resolving dependencies	271
Dependency injection with Spring	273

GroovyClassLoader	276
Integrating scripts	279
GroovyShell	282
CompilerConfiguration	282
GroovyScriptEngine	283
Summary	284
Index	285

Preface

The Java virtual machine runs on everything from the largest mainframe to the smallest microchip and supports every conceivable application. But Java is a complex and sometimes arcane language to develop with. Groovy allows us to build targeted single-purpose mini languages, which can run directly on the JVM alongside regular Java code.

This book provides a comprehensive tutorial on designing and developing mini Groovy-based Domain-Specific Languages. It is a complete guide to the development of several mini DSLs with a lot of easy-to-understand examples. This book will help you to gain all of the skills needed to develop your own Groovy-based DSLs.

Groovy for Domain-Specific Languages guides the reader from the basics through to the more complex meta-programming features of Groovy. The focus is on how the Groovy language can be used to construct domain-specific mini languages. Practical examples are used throughout to demystify these seemingly complex language features and to show how they can be used to create simple and elegant DSLs. The examples include a quick and simple Groovy DSL to interface with Twitter.

The book concludes with a chapter focusing on integrating Groovy-based DSL in such a way that the scripts can be readily incorporated into the reader's own Java applications. The overall goal of this book is to take Java developers through the skills and knowledge they need to start building effective Groovy-based DSLs to integrate into their own applications.

What this book covers

Chapter 1, *Introduction to DSL and Groovy*, discusses how DSLs can be used in place of general-purpose languages to represent different parts of a system. You will see how adding DSLs to your applications can open up the development process to other stakeholders in the development process. You'll also see how, in extreme cases, the stakeholders themselves can even become co-developers of the system by using DSLs that let them represent their domain expertise in code.

Chapter 2, *Groovy Quick Start*, covers a whistle-stop tour of the Groovy language. It also touches on most of the significant features of the language as a part of this tour.

Chapter 3, *Groovy Closures*, covers closures in some depth. It covers all of the important aspects of working with closures. You can explore the various ways to call a closure and the means of passing parameters. You will see how to pass closures as parameters to methods, and how this construct can allow us to add mini DSL syntax to our code.

Chapter 4, *Example DSL: GeeTwitter*, focuses on how we can start with an existing Java-based API and evolve it into a simple user-friendly DSL that can be used by almost anybody. You'll learn the importance of removing boilerplate code and how you can structure our DSL in such a way that the boilerplate is invisible to our DSL users.

Chapter 5, *Power Groovy DSL Features*, covers all of the important features of the Groovy language, and looks in depth at how some of these features can be applied to developing DSLs.

Chapter 6, *Existing Groovy DSLs*, discusses some existing Groovy DSLs that are in current use and are free to download.

Chapter 7, *Building a Builder*, explains how Groovy provides two useful support classes that make it much simpler to implement our own builders than if we use the MOP. You'll see how to use `BuilderSupport` and `FactoryBuilderSupport` to create our own builder classes.

Chapter 8, *Implementing a Rules DSL*, takes a look at Groovy bindings to see how they can be used in our DSL scripts. By placing closures strategically in the binding, you can emulate named blocks of code. You can also provide built-in methods and other shorthand by including closures and named Boolean values in the binding. These techniques can be used to a great effect to write DSL scripts that can be read and understood by stakeholders outside of the programming audience.

Chapter 9, *Integrating it all*, covers the many different ways in which you can integrate Groovy code into Java. You'll explore the issues around tightly integrating the two languages at compile time. You'll see how this can lead to dependency issues arising when Java code references Groovy classes and vice versa. You'll take a look at how you can use dependency injection frameworks like Spring to resolve some of these issues.

Who this book is for

This book is for Java software developers who have an interest in building domain scripting into their Java applications. No knowledge of Groovy is required, though it will be helpful. The book will not teach Groovy, but will quickly introduce the basic ideas of Groovy. An experienced Java developer should have no problems with these and can move quickly onto the more evolved aspects of creating DSLs with Groovy.

No experience of creating a DSL is required. The book should also be useful for experienced Groovy developers who have so far only used Groovy DSLs such as Groovy builders and would like to start building their own Groovy-based DSLs.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
class Customer {  
    int id  
    String name  
}
```

Any command-line input or output is written as follows:

```
$ groovyc monitor.groovy  
$ java -cp $GROOVY_HOME/embeddable/groovy-all-1.6.6.jar:. monitor  
HEAP USAGE  
Memory usage : 1118880  
Memory usage after GC: 607128
```

New terms and important words are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.



Downloading the example code for the book

Visit https://www.packtpub.com/sites/default/files/downloads/6903_Code.zip to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Introduction to DSL and Groovy

Java, and the Java platform with all of its frameworks and libraries, has by now become an all-encompassing universe for the software developer. The **Java Virtual Machine (JVM)** runs on practically every device from the smallest embedded chip to the largest mainframe. For the first time ever a whole gamut of application domains, from mobile games on your phone to mission critical enterprise applications are supported by this one-language platform.

The Groovy language is possibly one of the most important things to land in the Java universe in recent times. When developing with Java we were never constrained from doing what we wanted by the availability of libraries or frameworks. Arguably the only constraint that remains is the language itself. Like all traditional object-oriented languages, even Java requires a lot of boilerplate and scene-setting when coding. In Java, as a general purpose language, there is no problem that we cannot code a solution for. Sometimes, however, we might like to express the solution in a more concise style of coding that is supported by dynamic languages such as **Ruby** and **Python**. Groovy brings the power and flexibility of a dynamic scripting environment to the Java platform.

One of the big benefits of Groovy is how its dynamic features support the development of **Domain-Specific Languages (DSL)** or "**mini languages**", which we can run directly on the JVM alongside your existing Java code. Groovy DSLs integrate seamlessly into the Groovy language itself in such a way that it's not always apparent where the regular Groovy code stops and the DSL starts.

In fact, large parts of almost any Groovy application are written using Groovy-based DSLs. For instance, a new developer starting out with Groovy might assume that the builder code he uses to output some XML is a part of the core Groovy language. But it is, in fact, a mini internal DSL implemented using the Groovy metaprogramming features.

As developers, we could simply approach Groovy as a blackbox of features that includes a slew of useful and interesting DSLs. There is already a whole range of useful DSLs out there for us to exploit, such as the **Grails Object Relational Mapping (GORM)** provided in the Grails framework, or the Ant-based build system **DSL Gant**. Groovy's main appeal is that it is so easy to build our own DSLs with it. We can gain a lot by using Groovy in its vanilla form and exploiting all the nice DSLs that it comes with, and building our own DSLs that can run on the JVM alongside our Java code and seamlessly integrate with it. This book is all about encouraging you to realize the full potential of Groovy by teaching you how to build your own DSLs. We will learn that with Groovy this is surprisingly easy.

By the end of this book I hope that you will have the knowledge and the confidence to start building your own DSLs with Groovy, and be able to integrate them into your Java applications. To begin with, in this chapter we will take some baby steps. This chapter will give you a brief background to DSLs and their usage. We will also dip a toe into the Groovy language, and briefly touch on the features of the language that distinguish it from Java and make it a great tool for developing DSLs on top of the Java platform.

DSL: New name for an old idea

The term DSL has been around for just a few years. It describes a programming language that is dedicated to specific problem domain. The idea is not new. DSLs have been around for a long time. One of the most exciting features of UNIX has always been its mini languages. These include a rich set of typesetting languages (`troff`, `eqn`, `pic`), shell tools (`awk`, `sed` and so on), and software development tools (`make`, `yacc`, `lex`).

The Java platform has a multitude of mini DSLs in the form of XML config files for configuration of everything from EJBs to web applications. In many JEE applications, **Enterprise Java Beans (EJB)** need to be configured by using an XML configuration file, `ejb-jar.xml`. While the `ejb-jar.xml` is written in the general purpose language XML, the contents of the file need conform to a **Document Type Definition (DTD)** or XML schema, which describes the valid structure of the file.

XML configuration files can be found across a wide range of libraries and frameworks. Spring is configured by using a `spring-config.xml`, and Struts with `struts-config.xml`. In each case the DTD or schema defines the elements and tags, which are valid for the specific domain, be that EJB, Spring, or Struts. So `ejb-jar.xml` can be considered a mini DSL for configuring EJB, `spring-config.xml` is a mini DSL for configuring Spring beans, and so on.

In essence, DSL is a fancy name for something that we use every day of our professional programming lives. There are not many applications that can be

fully written in a single general-purpose language. As such we are the everyday consumers of many different DSLs, each of which is specific to a particular purpose.



Up to EJB 2.0 a DTD was required. In 2.1 the DTD was replaced by an XML schema, and from EJB 3.0 onwards the ejb-jar.xml is no longer required.



A typical day's work could involve working with Java code for program logic, CSS for styling a web page, JavaScript for providing some dynamic web content, and Ant or Maven to build the scripts that tie it all together. We are well used to consuming DSLs, but seldom consider producing new DSLs to implement our applications – which we should.

The evolution of programming languages

My own background is probably typical of many of my generation of old-school programmers. Back in 1986, I was a young software engineer fresh out of college. During my school and college years, I studied many different programming languages. I was fortunate in high school to have had a visionary Math teacher who taught us to program in Basic, so I cut my teeth programming as early as 1974. Through various college courses I had come in touch with Pascal, C, Fortran, Lisp, Assembler, and COBOL.

My school, college, and early professional career all reinforced a belief that programming languages were for the exclusive use of us programmers. We liked nothing better than spending hours locked away in dark rooms writing reams of arcane and impenetrable code. The more arcane and impenetrable, the better! The hacker spirit prevailed, and annual competitions such as the **International Obfuscated C Code Contest (IOCCC)** were born.

General-purpose languages

All of the teaching in college in those days revolved around the general-purpose languages. I recall sitting in class and being taught about the "two" types of programming language, machine language, and high-level languages. Both were types of general-purpose languages, in which you could build any type of application; but each language had its own strengths and weaknesses. The notion of a DSL was not yet considered as part of the teaching program. Nor was the idea that anyone other than a cadre of trained professional programmers (hackers) would ever write programs for computers. These days, the word hacker has bad connotations of being synonymous with virus writers and the like. In those days a good "hack" was an elegant programming solution to a hard problem and being called a hacker by one's peers was a badge of pride for most programmers.



The IOCCC runs to this day. The point of the contest is to write valid but impenetrable C code that works. Check out <http://www.ioccc.org> to see how not to write code.

The high-level programming language you used defined what type of application programmer you were. COBOL was for business application programming, Fortran was for scientific programmers, and C was for hackers building UNIX and PC software. Although COBOL and Fortran were designed to be used in a particular business domain they were still considered general-purpose languages. You could still write a scientific application in COBOL or a business application in Fortran if you wanted to. However, you were unlikely to try any low-level device driver development in COBOL.

Although it was possible to build entire applications in assembly language (and many people did), high-level languages, such as C, BASIC, and COBOL, were much better suited to this task. The first version of the world-beating spreadsheet Lotus 1-2-3 was written entirely in 8086 assembly language and, ironically, it was the rewrite of this into the supposed high-level language C that nearly broke the company in the late 1980s.

Languages such as C and C++ provide the low-level functionality in a high-level language, which enabled them to be used across a much greater range of domains, including those where assembly was utilized before. These days, Java and C++ compete with each other as the Swiss Army knives of general-purpose languages. There are almost no application domains to which both of these languages have not been applied, from space exploration, through enterprise business systems, to mobile phones.

Spreadsheets and 4GLs

Programs such as Lotus 1-2-3 and its precursor VisiCalc revolutionized people's view of who would program computers. A whole generation of accountants, financial analysts, scientists, and engineers came to realize that they could develop sophisticated turn key, solutions for themselves, armed only with a spreadsheet and a little knowledge of macros. Spreadsheet macros are probably one of the first DSLs to find their way out of the cloisters of the IT community and into the hands of the general business user.

Around this time, there was also much media attention paid to the new 4GL systems (4th Generation Languages). 4GLs were touted as being hugely more efficient for developing applications than traditional high-level languages, which then became known as 3rd generation languages (3GLs). From the hype in the media at the time, you would be forgiven for thinking that the age of the professional programmer

was coming to an end and that an ordinary business user could use a 4GL to develop his own business applications. I viewed this claim with a degree of healthy skepticism – how could a non-programmer build software?

Like DSLs, 4GLs were, generally speaking, targeted at particular problem spaces, and tended to excel at providing solutions in those narrow target markets. The sophistication of most applications in those days was such that it was possible to build them with a few obvious constructs. 4GLs tended to be turnkey environments with integrated tools and runtime environments. You were restricted by the environment that the 4GL provided, but the applications that could be built with a 4GL could be built rapidly, and with a minimal amount of coding.

4GLs differ from our modern understanding of a DSL. We generally think of a DSL as being a mini language with a particular purpose, and they do not generally impose an entire runtime or tool set on their use. The best DSLs can be mixed and matched together, and used in conjunction with a general purpose programming language such as C++ or Java to build our applications.

Language-oriented programming

Martin Fowler has spoken about the use of many mini DSLs in application development. He advocates building applications out of many mini DSLs, which are specific to the particular problem space, in a style of development called language-oriented programming. In a way, this style of programming is the norm for most developers these days, when we mix and match HTML, CSS, SQL, and Java together to build our applications.

The thrust of language-oriented programming is that we should all be going beyond exploiting these generally available languages and implementing our own DSLs that represent the particular problem space that we are working on. With a language-oriented programming approach, we should be building DSLs that are as narrowly-focused as the single application that we are currently working on. A DSL does not need to be generally applicable to be useful to us.

Who are DSLs for?

It's worth considering for a moment who the different types of users of a DSL might be. Most DSLs require some programming skills in order to get to grips with them, and are used by software and IT professionals in their daily chores, building, and maintaining and managing systems. They are specific to a particular technical aspect of system development. So the domain of CSS as a DSL is web development in general, and specifically page styling and layout. Many web developers start from a graphic design background and become proficient as coders of HTML, CSS, and JavaScript simply because it gives them better fine-grained control of the design process.

Many graphic designers, for this reason, eventually find themselves eschewing graphical tools such as Dreamweaver in favor of code. Hopefully, our goal in life will not be to turn everybody into a coder. Whereas most DSLs will remain in the realm of the programmer, there are many cases where a well-designed DSL can be used by other stakeholders in the development process other than professional developers. In some cases, DSLs can enable stakeholders to originate parts of the system by enabling them to write the code themselves. In other cases, the DSL can become a shared representation of the system. If the purpose of a particular DSL is to implement business rules then, ideally, that DSL should express the business rule in such a way that it can be clearly understood upon reading by both the business stakeholder who specified it and the programmer who wrote it.

A DSL for process engineers

My own introduction to the concept of DSLs came about in 1986 when I joined **Computer Products Inc. (CPI)** as a software engineer. In this case the DSL in question was sophisticated enough to enable the stakeholders to develop large parts of a running system.

CPI developed a Process Control System for its time. The system that was very innovative was primarily sold to chemical and pharmaceutical industries. It was a genuinely distributed system when most process control systems were based on centralized mini or mainframe computers. It had its own real-time kernel, graphics, and a multitude of device drivers for all types of control and measurement devices. But the most innovative part of the system, which excited customers most, was a scripting language called **EXTended Operations Language (EXTOL)**. EXTOL was a DSL in the purest sense because it drew the domain experts right into the development process, as originators of the running code.

With EXTOL, a chemical process engineer or chemist could write simple scripts to define the logic for controlling their plant. Each control block and measurement block in the system was addressable from EXTOL. Using EXTOL, a process engineer could write control logic in the same pseudo English that he used to describe the logic to his peers.

The following script could be deployed on a reactor vessel to control the act of half-filling the vessel with reactant from VALVE001.

```
drive VALVE001 to OPEN
when LEVELSENSOR.level >= 50%
drive VALVE001 to CLOSED
```

This was an incredibly powerful concept. Up to this point, most process control systems were programmed in a combination of high-level languages on the main process system, and relay logic on PLCs in the plant. Both tasks required specific programming skills, and could not generally be completed by the chemists or chemical engineers, who designed the high-level chemical processing undertaken at the plant. I recall a room full of white-coated chemists at one plant happily writing EXTOL scripts, as we commissioned the plant.

The proof of the pudding is always in the eating, and I don't recall a CPI engineer ever being called upon to write a single line of EXTOL code on behalf of a customer. Given an appropriate DSL that fit their needs, our customers could write all of the code that they needed themselves, without having to be programmers.

This shows the power of DSLs at their best. At this extreme end of the spectrum, a DSL becomes a programming tool that a domain expert can use independently, and without recourse to the professional programmer. It's important to remember, however, that the domain experts in this case were mostly process engineers. Process engineers are already well used to devising stepwise instructions, and building process flows. They will often use the same visual representations as a programmer, such as a flowchart to express a process that they are working on.

When devising a DSL for a particular domain, we should always consider the stakeholders who need to be involved in using it. In the case of EXTOL, the DSL was targeted at a technical audience who could take the DSL and become part of the system development process. Not all of our stakeholders will be quite as technical as this. But, the very least, the goal when designing a DSL should be to make the DSL understandable to non-technical stakeholders.

Stakeholder participation

It's an unfortunate fact that with many DSLs, especially those based on XML, the code that represents a particular domain problem is often only legible to the programming staff. This leads to a disconnect between what the business analysts and domain experts define, and what eventually gets implemented in the system. For instance, a business rule is most likely to be described in plain English by a business analyst in a functional specification document. But these rules will most likely be translated by developers into an XML representation that is specific to the particular rules engine, which is then deployed as a part of the application. If the business analyst can't read the XML representation and understand it, then the original intent of the rule can easily be lost in translation.

With language-oriented programming, we should aim to build DSLs that can be read and understood by all stakeholders. As such, these DSLs should become the shared living specification of the system, even if in the end they must, by necessity, be written by a programmer with a technical understanding of the DSL.

DSL design and implementation

DSLs can take many different forms. Some DSLs, such as Unix mini languages, (sed, awk, troff) have a syntactical structure, which is unique to that particular language. To implement such DSLs, we need to be able to parse this syntax out of the text files that contain the source code of that particular language. To implement our own DSL in this style involves implementing a mini compiler that uses lexing and parsing tools such as lex, yacc, or antlr.

Compiler writing is one particular skill that is outside the skill set of most application development teams. Writing your own parser or compiler grammar is a significant amount of effort to get into, unless the DSL is going to be used generally, and is beyond the scope of most application-specific DSLs.

EXTOL circumvented this problem by having its own syntax-sensitive editor. Users edited their EXTOL scripts from within the editor, and were prompted for the language constructs that they needed to use for each circumstance. This ensured that the scripts were always well formed and syntactically correct. It also meant that the editor could save the scripts in an intermediate pcode form so that the scripts never existed as text-based program files, and therefore never needed to be compiled.

Many of the DSLs that we use are embedded within other languages. The multitude of XML configuration scripts in the Java platform is an example of this. These mini DSLs piggyback on the XML syntax, and can optionally use an XML DTD or schema definition to define their own particular syntax. These XML-based DSLs can be easily validated for "well-formed-ness" by using the DTD or schema.

External versus internal DSL

We generally refer to DSLs that are implemented with their own unique syntax as external DSLs, and those that are implemented within the syntax of a host language as embedded or internal DSLs. Ideally, whenever building a new DSL, it would be best to give it its own unique and individual syntax. By designing our own unique syntax, we can provide language constructs, which are designed with both the problem domain and the target audience in mind.

If the intended user of the DSL is a non-programmer, then developing an XML-based syntax can be problematic. XML has its own particular rules about opening and closing and properly terminating tags that appear arcane to anybody except a programmer. This is a natural constraint when working with DSLs that are embedded/internal to another language. An XML-based DSL cannot help being similar to XML.

Embedded/internal DSLs will never be as free-form as a custom external DSL due to the constraints of the host language. Fortunately, Groovy-based DSLs are capable of being structured in a more human-readable format. However, they always need to use well-formed Groovy syntax, and there are always going to be compromises when designing Groovy-based DSLs that are readable by your target audience.

Operator overloading

Some general-purpose languages, such as C++, Lisp, and now Groovy, have language features that assist in the development of mini language syntaxes. C++ was one of the earliest languages to implement the concept of operator overloading. By using operator overloading, we can make non-numeric objects behave like numeric values by implementing the appropriate operators. So we can add a plus operator to a String object in order to support concatenation. When we implement a class that represents a numeric type, we can add the numeric operators again to make them behave like numeric primitives. We can implement a `ComplexNumber` class, which represents complex numbers as follows:

```
class ComplexNumber {  
public:  
    double real, imaginary;  
    ComplexNumber() { real = imag = 0; }  
    ComplexNumber(double r, double i) { real = r; imag = i; }  
    ComplexNumber& operator+(const ComplexNumber& num);  
};
```

To add one complex number to another, we need to correctly add each of the real and imaginary parts together to generate the result. We implement a plus operator for `ComplexNumber` as follows:

```
ComplexNumber& ComplexNumber::operator=(const ComplexNumber& num) {  
    real = num.real;  
    imag = num.imag;  
    return *this;  
}
```

This allows us then to add `ComplexNumber` objects together as if they were simple numeric values:

```
int main(int argc, const char* argv[]) {
    ComplexNumber a(1, 2), b(3, 4);
    ComplexNumber sum;
    sum = a + b;
    cout << "sum is " << sum.real << " ; "
        << sum.imaginary << "i" << endl;
}
```

One of the criticisms of the operator overload feature in C++ is that when using operator overloading, there is no way to control what functionality is being implemented in the overloaded function. It is perfectly possible – but not very sensible – to make the `+` operator subtract values and the `-` operator add values. Misused operator overloading has the effect of obfuscating the code rather than simplifying it. However, sometimes this very obfuscation can be used to good effect.

The preceding example illustrates what could be considered as a classic case of obfuscation in C++. If your use of C++ predicated the introduction of the standard C++ libraries and the streams libraries in particular, you would probably do a double take when looking at this code.

The example uses what has become commonly known as the stream operator `<<`. This operator can be used to send a character stream to standard output, the logic being that it looks very much like how we stream output from one program to another in a Unix shell script. In fact, there really is no such thing as a stream operator in C++ and what has been overloaded here is the binary left shift operator `<<`. I have to admit that my first encounter with a code like this left me perplexed. Why anybody would want to left shift the address of a string into another object was beyond me. Common use over the intervening years means that this is now a perfectly natural coding style to all C++ programmers. In effect, the streaming operator implements a mini internal DSL for representing streaming. It subverts the original language a little by using an operator out of context, but the end effect is perfectly understandable and makes sense.

During a fireside chat event at JavaONE some years ago, James Gosling was asked if he would ever consider operator overloading for the Java language, and the answer was a resolute no! Fortunately, we don't have to wait and see if Sun will ever add operator overloading to Java. With Groovy we can have it now. Groovy has an extensive set of features, including operator overloading, that allow us to implement feature-rich DSLs from within the language. We'll take a look at some of those features that distinguish it from Java, now.

Groovy

In the later chapters of this book, we will discuss the Groovy language in detail, but let's begin with a brief introduction to the language and some of the features that make it a useful addition to the Java platform.

The Java platform has expanded over the years to cover almost all conceivable application niches – from Enterprise applications, to mobile and embedded applications. The core strengths of Java are its rich set of APIs across all of these problem domains and its standardized **Virtual Machine (VM)** interface. The standard VM interface has meant that the promise of "write once, run anywhere" has become a reality. The JVM has been implemented on every hardware architecture and operating system from the mightiest mainframe down to the humble Lego Mindstorms robotic kits for kids.

On top of this standard VM, the list of APIs that have been built extends into every conceivable domain. In addition to the standard APIs that are apart of JME, JSE, and JEE, which are extensive in themselves, there are literally thousands of open source component libraries and tools to choose from. All of this makes for a compelling argument to use Java for almost any software project that you can think of.

For many years of its evolution, the JVM was considered to be just that – a virtual machine for running Java programs. The JVM spec was designed originally by James Gosling to be used exclusively for the Java language. In recent years, there have been a number of open source projects that have started to introduce new languages on top of the JVM, such as **JRuby** (an implementation of the Ruby language) and **Jython** (an implementation of the Python language and Groovy).

A natural fit with the JVM

Groovy differs from the above mentioned languages, as the Groovy language was designed specifically to be a new language to run on the JVM. Groovy is designed to be source-compatible with the Java language, as well as being binary-compatible at the byte code level.

James Strachan and Bob McWhirter started the Groovy project in August 2003 with the goal of providing a new dynamic and object-oriented language, which could run on the JVM. It took several existing dynamic languages, such as Ruby, Python, Dylan, and Smalltalk, as its inspiration. James had looked at the Python scripting language and had been impressed with the power that it had over Java. James and Bob wanted to design a language that had the powerful scripting features of Python, but stayed as close to the Java language as possible in terms of its syntax.

For this reason, Groovy is code-compatible with Java, and for this reason it is possible in most cases to take an existing .java file and rename it to .groovy and it will continue to work. Groovy has its own compiler, groovyc, which generates Java byte code from Groovy source files just as the javac compiler does. Groovyc generates class files, which run directly on the JVM. Methods defined in a Groovy class can be called directly from Java and vice versa.

Groovy classes and interfaces are 100% binary-compatible with their Java counterparts. Uniquely, this means that we can create a new Groovy class that extends a Java class or implements a Java interface. You can also create Java classes that extend Groovy classes or implement Groovy interfaces.

Groovy language features

Groovy adds a number of unique features that distinguish it from Java and allow developers to code at a higher level, and use a more abstract idiom, than is possible with Java. Placing all of these powerful features on top of a language that is code and API compatible with the Java platform is a powerful proposition.

Static and optional typing

In Java, as in other statically-typed languages, variables must first be declared with a type before they can have a value assigned to them. In Groovy, type can be left to be determined at the time of assignment. Groovy supports both static and optional typing, as follows:

```
String str1 = "I'm a String"  
str2 = "I'm also a String"
```

Both variables `str1` and `str2` are of type `String`. The late binding of the type in the Groovy-style assignment allows for a much less verbose code.

Native support for lists and maps

One of the great bugbears of the Java language is the cumbersome interfaces required for list and map manipulation. Groovy adds native support for all of the Java collection types through a very intuitive and readable syntax. The following code:

```
authors = [ 'Shakespeare', 'Beckett', 'Joyce', 'Poe' ]  
println authors  
println authors[2]
```

produces the output:

```
["Shakespeare", "Beckett", "Joyce", "Poe"]  
Joyce
```

Maps are also declared with ease:

```
book = [ fileUnder: "Software Development",
         title: "Groovy for DSL" , author: "Fergal Dearle"]
println book
println book['title']
println book.title
```

which produces the following output:

```
["fileUnder":"Software Development", "title":"Groovy for DSL",
 "author":"Fergal Dearle"]
Groovy for DSLClosures
```

Closures

Closures are one of the most powerful language features in Groovy. Closures are anonymous code fragments that can be assigned to a variable. Closures can be invoked by the `call` method as follows:

```
biggest = { number1, number2 -> number1<number2?number2:number1 }
// We can invoke the call method of the Closure class
result = biggest.call(7, 1)
println result
// We can use the closure reference as if it were a method
result = biggest(3, 5)
println result
// And with optional parenthesis
result = biggest 13, 1
println result
```

Closures can contain multiple statements and can therefore be as complex as you like. In the following example, we iterate through a list looking for the biggest number, and return it when we are done.

```
def listBiggest = { list ->
    def biggest = list[0]
    for( i in list)
        if( i > biggest)
            biggest = i
    return biggest
}
def numberList = [ 8, 6, 7, 5, 3, 9]
println listBiggest( numberList)
```

Groovy operator overloading

Operator overloading is a powerful feature of the C++ language. Java inherited many of the features of the C++ language, but operator overloading was significantly left out. Groovy introduces operator overloading as a base language feature.

Any Groovy class can implement a full set of operators by implementing the appropriate corresponding method in the class. For example, the plus operator is implemented via the `plus()` method.

Regular expression support

Groovy builds regular expression handling right into the language via the `=~` operator and matcher objects. The following example creates a regular expression to match all multiple occurrences of the space character. It creates a `matcher` object from this expression and applies it to a string by using the `replaceAll` method.

```
lorem = "Lorem ipsum dolor sit amet, consectetur adipisicing elit"
println lorem
matcher = string =~ " +"
removed = matcher.replaceAll(" ")
println removed
```

Optional syntax

Optional typing means that variable type annotations are optional. This does not mean that variables have an unknown variable type. It means that the type will be determined at run time based on the value that gets assigned to the variable. All of the following are legal syntax in Groovy:

```
int a = 3
b = 2
String t = "hello"
s = 'there'
```

Trailing semicolons at the end of statements are optional. The only time that you explicitly need to use a semicolon in Groovy is to separate statements that occur on the same line of code, as shown in the first and third lines in the following code:

```
int a = 3; int b = 4;
c = 2
d = 5; e = 6
```

Method call parentheses are also optional when the method being invoked is passed some parameters. We saw earlier, with closures, that we can invoke a closure through its reference as if it were a method call. When invoking a closure in this way we can also drop the parentheses when passing parameters.

```
println( a );
c = 2
print c
printit = { println it }
printit c
```

These make for a much looser programming style, which is closer to the scripting syntax of Ruby or Python. This is a big benefit when we are using Groovy to build DSLs. When our target audience is non-technical, being able to drop parentheses and semicolons will make our code much more legible. Consider the following example, where we have two methods or closures to get an account by `id` and then credit the account with some funds:

```
Account account = getAccountById( 234 );
creditAccount( account, 100.00 );
```

With optional types, such as parenthesis and semicolons, this can be used to write code that is far more legible to our target audience.

```
account = getAccountById 234
creditAccount account, 100.00
```

Groovy markup

Built in Groovy are a number of builder classes. There are markup builders for HTML, XML, Ant build scripts, and for Swing GUI building. Markup builders allow us to write code to build a tree-based structure directly within our Groovy code. Unlike API-based approaches for building structures, the tree-like structure of the resulting output is immediately obvious from the structure of our Groovy markup code. Consider the following XML structure:

```
<?xml version="1.0"?>
<book>
    <author>Fergal Dearle</author>
    <title>Groovy for DSL</title>
</book>
```

In Groovy markup, this XML can be generated simply with the following code fragment:

```
def builder = new groovy.xml.MarkupBuilder()
builder.book {
    author 'Fergal Dearle'
    title 'Groovy for DSL'
}
```

At first glance, this looks like strange special case syntax for markup. It's not! The structure of this code can be explained through the use of closures and the optional syntax that we've discussed in this chapter. We will go into this in great detail in *Chapter 5, Power Groovy DSL features* but it is interesting at this point to see how the clever use of some language features can yield a powerful DSL-like markup syntax.

Breaking down the above code a little, we can rewrite it as:

```
def builder = new groovy.xml.MarkupBuilder()
closure = {
    author 'Fergal Dearle'
    title 'Groovy for DSL'
}
// pass a closure to book method
builder.book(closure)
// which can be written without parentheses
builder.book closure
// or just inline the closure as a parameter
builder.book {
...
}
```

In other words, the code between the curly braces is in fact a closure, which is passed to the `book` method of `MarkupBuilder`. Parentheses being optional, we can simply declare the closure inline after the method name, which gives the neat effect of seeming to mirror the markup structure that we expect in the output.

Similarly, `author` and `title` are just method invocations on `MarkupBuilder` with the optional parentheses missing. Extending this paradigm a little further we could decide to have `author` take a closure parameter as well:

```
def builder = new groovy.xml.MarkupBuilder()
builder.book {
    author {
```

```
        first_name 'Fergal'  
        surname 'Dearle'  
    }  
    title 'Groovy for DSL'  
}
```

This will output the following nested XML structure:

```
<?xml version="1.0"?>  
<book>  
    <author>  
        <first_name>Fergal</first_name>  
        <surname> Dearle</surname>  
    </author>  
    <title>Groovy for DSL</title>  
</book>
```

The method calls on `MarkupBuilder` start off by outputting an opening XML tag, after which they invoke the closure if one has been passed. Finally, the XML tag is properly terminated before the method exits. If we analyze what happens in sequence, we can see that `book` invokes a closure that contains a call to `author`. Additionally, the `author` contains a closure with calls to `first_name`, `surname`, and so on.

Before you go to the Groovy documentation for `MarkupBuilder` to look for the `book`, `author`, and `surname` methods in `MarkupBuilder`, let me save you the effort. They don't exist. These are what we call pretend methods. We will see later in the book how Groovy's metaprogramming features allow us to invoke methods on closure that don't really exist, but have them do something useful anyway.

Already we are seeing how some of the features of the Groovy language can coalesce to allow the structuring of a very useful DSL. I use the term `DSL` here for Groovy builders, because that is essentially what they are. What initially looks like special language syntax for markup is revealed as being regular closures with a little bit of clever metaprogramming. The result is an embedded or internal DSL for generating markup.

Summary

So now we have a feel of DSLs and Groovy. We have seen how DSLs can be used in place of general-purpose languages to represent different parts of a system. We have also seen how adding DSLs to our applications can open up the development process to other stakeholders in the development process. We've also seen how, in extreme cases, the stakeholders themselves can even become co-developers of the system by using DSLs that let them represent their domain expertise in code.

We've seen how using a DSL that makes sense to a non-technical audience then means it can become a shared resource between programming staff and business stakeholders, representing parts of the system in a language that they all understand. So we are beginning to understand the importance of usability when designing a DSL.

We have dipped a tentative toe in the water by looking at some Groovy code. We've gained an appreciation of how Groovy is a natural fit with the Java language due to its binary and class level compatibility. We have touched on the features of the Groovy language that makes it unique from Java, and looked at how these unique features can be used as a basis for building on the base Groovy language with internal DSLs.

In the next chapter, we will go into more depth with the language itself and see how we can use these features to build programs. In subsequent chapters, we will dive deeper and see how the language can be exploited as an ideal platform for building DSLs on top of the Java platform.

2

Groovy Quick Start

In this chapter, we will jump straight into Groovy and start exploring the basics of the language. Before we do that, we need to learn a little about the Groovy toolset and set up an environment to work in. We have several different options for how to run Groovy. Groovy programs can be compiled and run standalone or as part of a larger application. They can be invoked from the command line as Groovy scripts or we can run them interactively through the Groovy shell or the Groovy console. Most IDE environments by now also have plug-ins available that allow us to run our Groovy programs directly within them.

- We will start out with a section on how to find the Groovy binaries and install them on your system.
- The next section guides you through running Groovy scripts by using the various shell tools provided with the Groovy download.
- Most of you will be using one of the popular IDE environments; so we'll look at the various integration options with the popular IDEs and programmer's editors.
- We'll finish up with the main part of the chapter, which is a whistle-stop tour of the Groovy language. We don't have the scope in this book to cover the whole language tutorial fashion, but by the end of the book, we will have covered all of the aspects of the language that you need to be able to write your own Groovy-based DSLs. For now, in this chapter, we will just touch on some of the main points that differentiate Groovy from its parent language—Java.

How to find and install Groovy

The Groovy project is hosted by codehaus.org at <http://groovy.codehaus.org> and can be downloaded as a ZIP archive or a platform-specific installer for Windows and certain Linux distributions. At the time of writing this book, the latest version of the language available is Groovy 1.7.

In five simple steps, you can run Groovy and start experimenting with the language.

1. Download the latest build from <http://groovy.codehaus.org/Download>.
2. Unzip the archive into a directory on your computer.
3. Set an environment variable in your command line or shell for GROOVY_HOME. This should point to the base directory to which you unzipped the archive.
4. Add the Groovy bin directory to your PATH. This will be %GROOVY_HOME%\bin (Windows) or \$GROOVY_HOME/bin on Linux and Unix systems.
5. Open a new command shell and test your setup by issuing the groovy version command **groovy -v**

If all goes well, you should see something like the following:

```
$groovy -v  
Groovy Version: 1.6.6 JVM: 1.6.0_03
```

Your Groovy installation relies on having a working Java version set up already on your computer. Groovy will work with any version of Java from 1.4.1 onwards, but if you want to use some of the language features such as generics and annotations, you will need to have a minimum of Java 1.5 installed. You can check your Java version with the command `java -version`, if you are not sure. You can find an upgrade at <http://java.sun.com/javase/downloads/index.jsp>.

Running Groovy

Now that you have Groovy installed, let's introduce some of the tools that come with the Groovy package. Groovy can be compiled into a Java class file and deployed as part of an application, the same as for any other Java class file. In addition to this, Groovy has several tools that allow us to execute a Groovy program as a script without the need to package it into a Java application.

There are three commands that we can use to launch a script. In the following sections, we will demonstrate these different methods of running Groovy scripts. As we progress through the book, you can use these methods to execute the Groovy scripts that we will describe.

The Groovy script engine—groovy

Let's start by writing a Groovy version of the ubiquitous Hello World program. We can start by creating a file called `Hello.groovy`, which contains the following code:

```
public class HelloGroovy {  
    public static void main(String [] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

To any Java developer, this looks strangely like Java code. That's because it is Java code. In the first instance, Groovy is Java source code compatible. Almost anything you write in Java is source-level compatible with Groovy. To prove it, let's try and run the following code as a script from the command line:

```
$groovy HelloWorld.groovy  
Hello, World!  
$
```

This is interesting, and it is a feature that we can make good use of in the future, but we are not gaining many of the benefits of Groovy by writing in Java. Let's rewrite this script to be more Groovy.

Groovy is also a scripting language, so we don't need to write our code within a class to execute it, and we don't need a static `main` method either. A lot of useful methods from the JDK, such as `println`, are provided as wrapper shortcuts by the Groovy class `DefaultGroovyMethods`. Thus, we can rewrite our Hello World program in one line of code as follows:

```
println "Hello, World!"
```

The Groovy script engine also defaults the filename suffix; so the following command works just as well:

```
$groovy Hello  
Hello, World!  
$
```

We can also invoke the Groovy script engine and pass it a single statement to execute:

```
$groovy -e "println 'Hello, World!'"  
Hello, World!  
$
```

We already know that Groovy is a language built on top of the JVM and that it runs as Java byte-code. So how is it possible for a Groovy script to run without being compiled to a Java class? The answer is that the Groovy scripting engine compiles scripts on the fly and loads the byte-code onto the JVM. We will find out how to compile our Groovy later in the chapter.

The important thing to note for now is that even though our scripts seem to be running on the command line, they are in fact running on a JVM, and as a result, we have access to all of the power of the JVM and Java APIs. To demonstrate, let's look at a more complicated script.

The Java management extension JMX is an extremely useful component of the Java platform. JMX is a framework for managing and monitoring applications, system objects, and devices. In JMX, resources are represented as MBeans and we can use the JMX APIs to access the resources to monitor the state of our application.

Typical JMX clients such as jManage tend to be heavy-weight GUI applications that allow application resources to be inspected. Sometimes I just want to monitor one or two values that are relevant to the performance of my application, such as its current heap usage, and log it to a file at intervals.

Consider the following Groovy script, `monitor.groovy`, which connects to the platform MBean of a remote JVM and monitors its heap usage before and after a garbage collection operation.

```
import java.lang.management.*  
import java.lang.management.ManagementFactory as Factory  
import javax.management.remote.JMXConnectorFactory as JMX  
import javax.management.remote.JMXServiceURL as ServiceURL  
  
def serverUrl = 'service:jmx:rmi:///jndi/rmi://localhost:3333/jmxrmi'  
def server = JMX.connect(  
    new ServiceURL(serverUrl)).MBeanServerConnection  
  
println "HEAP USAGE"  
def mem = Factory.newPlatformMXBeanProxy(server,  
    Factory.MEMORY_MXBEAN_NAME, MemoryMXBean.class)  
def heapUsage = mem.heapMemoryUsage  
println """Memory usage : $heapUsage.used"""  
mem.gc()  
heapUsage = mem.heapMemoryUsage  
println """Memory usage after GC: $heapUsage.used"""
```

You can try this script against any running Java application. Just add the following switches to the Java startup command of the application. This instructs your JVM to start up with an open JMX connection on port 3333. For simplicity we have abstained from using a secure connection or password authentication.

```
-Dcom.sun.management.jmxremote.port=3333  
-Dcom.sun.management.jmxremote.authenticate=false  
-Dcom.sun.management.jmxremote.ssl=false
```

After running the script and connecting to an application, we get instant feedback about the state of our heap usage:

```
$groovy memory.groovy  
HEAP USAGE  
Memory usage : 1118880  
Memory usage after GC: 607128
```

If you are not familiar with JMX, don't worry. The point to be grasped here is that Groovy unleashes the power of the JVM and APIs, and puts them at your disposal through the command line shell. My starting point for the above script was an existing Java program that implemented the JMX API calls I needed. This is a slightly cleaned up version of the script, which exploits some Groovy syntax that we will learn about later. In practically no time it is possible to drop Java API code into a Groovy script and—bingo—you have a command line version to play with. The possibilities for tool development are endless.

Shebang scripts

If you are running Groovy on UNIX, Linux, or MAC OSX, you can go one step further with this approach. The Groovy script engine is designed to work as a proper shell scripting language and supports the "shebang" #! characters. By placing `#!/usr/bin/env groovy` as the first line of a script, the shell will pass the remainder of the script for processing by groovy. These are commonly referred to as **shebang scripts**. We can modify our `Hello.groovy` as follows.

```
#!/usr/bin/env groovy  
println "Hello, World!"
```

Now if we change the permissions of the file to executable, we can call it directly from the shell:

```
$mv Hello.groovy Hello  
$chmod a+x Hello  
.Hello  
Hello, World!  
$
```

The Groovy shell: groovysh

The Groovy shell is a useful command line tool for trying out snippets of Groovy code interactively. The shell allows you to enter Groovy code line-by-line and execute it. We can run the **groovysh** command and immediately start entering Groovy statements.

```
$ groovysh  
Groovy Shell (1.6.6 JVM: 1.5.0_16)  
Type 'help' or '\h' for help.  
-----  
-----  
groovy:000> "Hello, World!"  
====> Hello, World!  
groovy:000>
```

The Groovy shell evaluates each line and outputs the return value of the statement. The above statement contains just one instance of a string, so the string "Hello, World!" is returned. By contrast, our single line Hello script outputs "Hello, World!" but returns null, as follows:

```
groovy:000> println "Hello, World!"  
Hello, World!  
====> null  
groovy:000>
```

This is an important point to remember if you make any use of the Groovy shell, as it sometimes can be the cause of unexpected error messages in otherwise correct scripts. The Groovy shell will try to interpret and print the return value of your statement whether it makes sense to do so or not.

With the Groovy shell, statements can span more than one line. Partially-complete Groovy statements are stored in a buffer until completion. You can use the **display** command to output a partially-complete statement, as follows:

```
groovy:000> class Hello {  
groovy:001> display  
001> class Hello {  
groovy:001> String message  
groovy:002> display  
001> class Hello {  
002> String message  
groovy:002> }  
====> true  
groovy:000>
```

The **load** command allows scripts to be loaded into **groovysh** from a file. However, some limitations of the Groovy shell make this problematic. The Groovy shell works by using an instance of the **GroovyShell** class to evaluate each line of script in turn. When we run any Groovy script, the variables local to the script are stored in the binding object. Because **groovysh** evaluates the script piecemeal as it encounters each line, only variables that find their way into the binding are preserved. So the following code works because the **message** variable is stored in the binding, which is shared between evaluations:

```
groovy:000> message = "Hello, World!"  
====> Hello, World!  
groovy:000> println message  
Hello, World!  
====> null  
groovy:000>
```

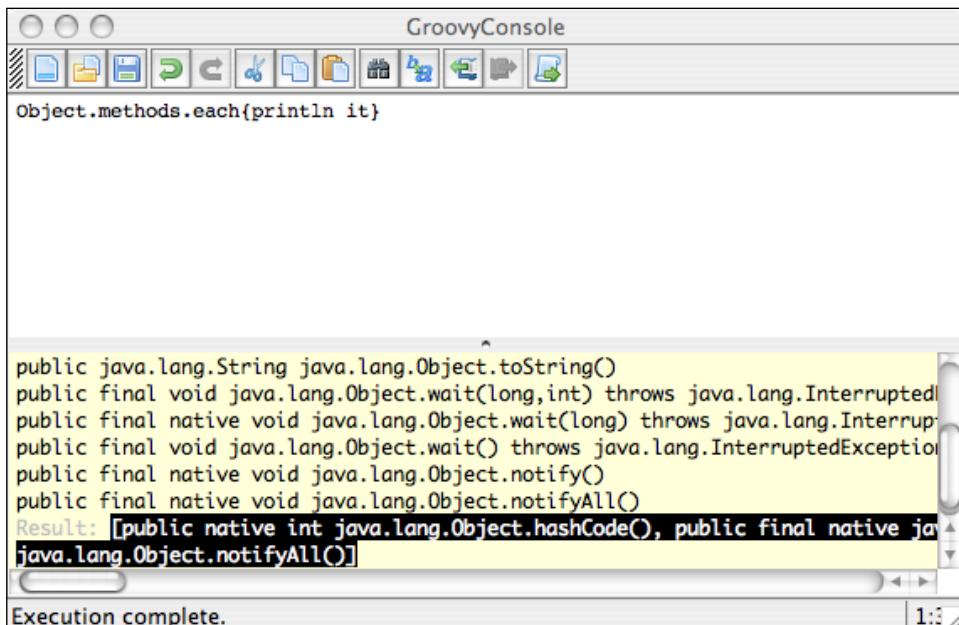
However, this version causes an error, as **message** is now treated as a local variable and not stored in the binding.

```
groovy:000> String message = "Hello, World!"  
====> Hello, World!  
groovy:000> println hello  
ERROR groovy.lang.MissingPropertyException: No such property: hello for  
class: groovysh_evaluate  
at groovysh_evaluate.run (groovysh_evaluate:2)  
...  
groovy:000>
```

The Groovy console: groovyConsole

The limited set of commands and crude command line operations make groovysh problematic for anything other than trying out single expressions or statements. A much more useful tool is the Groovy console. The Groovy console is a GUI editor and runtime environment. You only need to type your Groovy statements in the top pane and the output gets listed in the bottom pane.

You can launch the Groovy console from the command line with **\$groovyConsole command:**



In the **GroovyConsole** shown above we are trying out a handy feature of Groovy, which is the ability to find class methods on the fly. This simple line of code exploits a few of the Groovy language features. `Object.methods` is analogous to calling `getMethods()` for the `Object` class. Instead of a Method array, a Groovy list is returned containing the Method objects. We use the built-in iteration method to pass each element in the list to the closure that prints the method details. We will go over all of these language features in depth later.

On the other hand, `GroovyShell`, the class on which `groovysh` is built, is an extremely useful class. Later in this book we will use `GroovyShell` to evaluate DSL scripts on the fly, but other uses can be made of it including building an on-the-fly Groovy interpreter into your application, if necessary.

The Groovy console has several other features, such as the ability to select a part of the buffer and run it, and also has a useful Object browser. This allows you to inspect the last result object from the console. Combined with the ability to select part of the display buffer and run it independently, the console is the ideal sandbox for playing with code snippets – whether debugging existing code or learning the Groovy language.

The Groovy compiler: groovyc

By using the Groovy script engine, we can execute Groovy scripts from the command line. For experimenting and debugging our scripts, we can run them interactively in the Groovy shell or the Groovy console. To build our Groovy programs into larger apps that require more than one class, or to integrate our Groovy programs into existing Java applications, we need to be able to compile Groovy.

The **groovyc** command works exactly the same way as **javac** does. It takes a Groovy source file and compiles it into a corresponding class file that can be run on the JVM. Our script examples up to now have not defined a class. The Groovy compiler will wrap our Groovy scripts into an executable class file, which can be invoked with the **java** command as if they had a `public static void main()` method.

Let's take the JMX monitor.groovy script that we wrote earlier, and compile and run it.

```
$ groovyc monitor.groovy
$ java -cp $GROOVY_HOME/embeddable/groovy-all-1.6.6.jar:. monitor
HEAP USAGE
Memory usage : 1118880
Memory usage after GC: 607128
```

Groovy IDE and editor integration

If you are going to do any amount of serious Groovy coding, you will want to work with Groovy in your favorite IDE.

Netbeans

Of the popular IDE environments, Netbeans was the first to provide built-in Groovy support. From Netbeans 6.5 onwards, Groovy support is available from within any of the Java bundles without any additional plug-ins being required. By default, you have excellent Groovy source editing with syntax highlighting, source folding, and code completion. You can mix and match Groovy with Java in your projects, or build a full Groovy on Grails-based project from scratch.

Eclipse

Eclipse was the first Java IDE to have Groovy support integrated through the GroovyEclipse plug-in. You can download an archive of the GroovyEclipse plug-in from <http://dist.codehaus.org/groovy/distributions/update/GroovyEclipse.zip>. Or better still, use the update site at <http://dist.codehaus.org/groovy/distributions/updateDev/>. The GroovyEclipse plug-in has full support for source-level Groovy editing with syntax highlighting, auto completion, and refactoring.

IntelliJ IDEA

Users of IntelliJ IDEA can use JetBrains' own JetGroovy plug-in. This plug-in can be downloaded from the IntelliJ plug-ins site at <http://plugins.intellij.net>.

Other IDEs and editors

Other IDEs with Groovy support are JDeveloper and JEdit. In addition, many of the popular program editors, such as TextMate and UltraEdit, also now have Groovy support. There is even a plug-in available to download for emacs. Check out <http://groovy.codehaus.org/IDE+Support> for a full list of available plug-ins, and extensive instructions on setting up and running Groovy in your preferred environment.

Introducing the Groovy Language

In the following sections, we will take a whistle-stop tour of the Groovy language. A working knowledge of Java is assumed, so we will focus on what is different between the Groovy and Java languages.

Module structure

Groovy programs and scripts are generally stored in Groovy source files with the .groovy extension. The exception to this are the Unix "shebang" scripts described above. Unlike Java source files, which must always contain a class definition, Groovy source files can contain both class definitions and inline scripting. Groovy generates a class object for each Groovy class that it encounters in the source. If the source file contains some scripting elements, it also generates a class object for these.

To see how this works, let's take an example script and compile it with `$groovyc AccountTest.groovy`. The example below contains two class definitions and some script that uses these classes.

```
class Customer {  
    int id  
    String name  
}  
  
class Account {  
    int id  
    double balance  
    Customer owner  
    void credit (double deposit) {  
        balance += deposit  
    }  
    String toString() {  
        "Account id ${id} owner ${owner.name} balance is ${balance}"  
    }  
}  
customer = new Customer(id:1,name:"Aaron Anderson")  
savings = new Account(id:2, balance:0.00, owner:customer)  
  
savings.credit 20.00  
println savings
```

Compiling the above code with `groovyc` will result in the generation of three class files: `Customer.class`, `Account.class`, and `AccountTest.class`. If we were to name our script `Customer.groovy` or `Account.groovy`, the Groovy compiler will politely suggest that we change either our script name or our class name because it cannot generate all of the required class files.



As a rule of thumb, I suggest only using this mixed mode of coding when you are writing standalone scripts. When writing Groovy code that will be integrated into a larger application, it is better to stick with the Java way of doing things so that the source file names correspond to the class names that you deploy as a part of your application.

One important difference to remember when writing Groovy classes versus Groovy scripts is that Groovy scripts have a special binding for variable references. In scripts we can immediately start using a variable from the point that we initialize it without having to declare it first. Script variables are stored in this Binding scope. So when we initialize the `savings` variable above with:

```
savings = new Account(id:2, balance:0.00, owner:customer)
```

the `savings` variable is automatically added to the Binding scope. At the point we make use of `savings`.

```
println savings
```

At this point, `savings` must be in the Binding scope or we will get an error.

If we rewrite the script portion of our above example to include a class definition and `main`, then `savings` and `customer` must be explicitly defined by using the `def` keyword, as follows:

```
class AccountSample {  
    public static void main (args) {  
        def customer = new Customer(id:1,name:"Aaron Anderson")  
        def savings = new Account(id:2, balance:0.00, owner:customer)  
  
        savings.credit 20.00  
        println savings  
    }  
}
```

Groovy shorthand

We have seen already that Groovy is source compatible with Java. To be more script-like, Groovy has some syntax elements that are optional and other syntax shortcuts that make code easier to read and write. By examining the `AccountTest` example, we can see some of these shorthand features in action.

Assumed imports

Java automatically imports the `java.lang` package for you. Groovy goes a step further and automatically imports some of the more commonly-used Java packages, as follows:

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

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

Two additional packages from the **Groovy JDK (GDK)** are also imported:

- `groovy.lang.*`
- `groovy.util.*`

Default visibility, optional semicolon

The majority of classes that I have written in Java have been declared public. Java requires us to always explicitly express the public visibility of a class. This is because the default visibility of classes is "package private", which, to be honest, is a visibility that is seldom used and is often misunderstood. "Package private" visibility means classes are accessible by other classes in the package but not by classes in other packages. Groovy makes the more sensible decision that public visibility is the default, so it does not need to be stated in the class definition.

Java uses the semicolon to separate statements even when they end on the same line. In Groovy, semicolons are optional as long as we limit ourselves to a single statement per line. This small change makes for much cleaner looking code.

```
def customer = new Customer(id:1,name:"Aaron Anderson")
def savings = new Account(id:2, balance:0.00, owner:customer)
```

The previous snippet from our Account example would have been more syntactically verbose in Java, without adding to the clarity of the code.

```
Customer customer = new Customer(id:1,name:"Aaron Anderson");
Account savings = new Account(id:2, balance:0.00, owner:customer);
```

If we have multiple statements on a line, a semicolon is required. The semicolon can still be left off the last statement in the line.

```
class Account {
    def id; double balance; Customer owner
...
}
```

Optional parentheses and types

The parentheses around method call parameters are optional for top-level statements. We have been looking at this language feature since the start of the chapter. Our Hello World program:

```
println "Hello, World!"
```

is in fact a call to the built-in Groovy `println` method and can be expressed with parenthesis if we want as follows:

```
println ("Hello, World!")
```

Similarly, the call to the `Account.credit` method in our `Account` example could have been written with parentheses.

```
savings.credit( 20.00 )
```

When a method call or closure call takes no arguments then we need to supply the parentheses. The compiler will interpret any reference to a method without parameters as a property lookup for the same name. A reference to a closure will return the closure itself:

```
getHello = { return "Hello, World" }

// Prints the closure reference
hello = getHello
println hello

// Parens required because
// println
// on its own is a reference to a property called println
println ()

// calls the closure
hello = getHello()
println hello
```

When method calls are nested, the parentheses are also needed to let the compiler distinguish between the calls.

```
greeting = { name -> return "Hello, " + name }

// Parens are optional for println but required for nested
// greeting call
println greeting ( "Fergal" )
```

Groovy also has the concept of dynamic types. We can define variables and member fields with static types if we wish. In our `Customer` and `Account` class, we declare our `id` fields with the `def` keyword. This allows us to decide at runtime what actual type we want our `ids` to be.



Note that the `def` keyword is required in a class context but is optional when we write Groovy scripts. Another way to look at the `def` keyword is that it declares a variable of type `Object`; so the following two lines are analogous:

```
def number = 1
Object number = 1
```

Optional return keyword

Every statement in Groovy has a resulting object value. We can see this clearly by running the Groovy shell `groovysh` and typing in statements. The shell outputs the resulting value of each statement as it is executed. Sometimes this value is `null`, as in the case of our Hello World script.

```
groovy:000> println message
Hello, World!
====> null
```

In Groovy, the `return` keyword is optional because the result value of the last statement is always returned from a method call. If the method is a `void` method, then the value returned will be `null`.

```
String toString() {
    "Account id " + id + " owner " + owner.name + " balance is " +
balance
}
```

The above result is a `String` object, which is automatically returned from the `toString` method. Other styles, which work equally well, are:

```
// Returned is the value contained in result
String greeting(name) {
    result = "Hello, " + name
    result
}
// Return type is dynamic, but the concrete return type is String
def greeting() {
    "Hello, " + name
}
```

Properties and GroovyBeans

We know from Java that a JavaBean is a class that implements getters and setters for some or all of its instance fields. Groovy automatically generates getters and setters for instance fields in a class that have the default visibility of public. It also generates the default constructor. This means that a Groovy class is automatically accessible as a JavaBean without any additional coding. Instance fields that have automatically generated getters and setters are known in Groovy as **properties**, and we refer to these classes as **GroovyBeans**, or by the colloquial **POGO** (plain old Groovy object).

```
Customer = new Customer()
Customer.setName("Brian Beausang")
println customer.getName()
customer.name = "Carol Coolidge"
println customer.name
```

This code snippet shows how we can optionally use getter/setter methods to manipulate the name field of our `Customer` class, or we can use field access syntax. It's important to note that when we use the field access syntax `customer.name`, we are not accessing the field directly. The appropriate getter or setter method is called instead.



If you want to directly access the field without going through a getter or setter, you can use the field dereference operator `@`. To access `customer.name` directly, you would use `customer@name`.



A JavaBean version of our `Customer` class would require getters, setters, and a default constructor, as follows:

```
public class Customer implements java.io.Serializable {
    private int id;
    private String name;

    public Customer () {
    }
    public int getId() {
        return this.id;
    }
    public void setId(final int id) {
        this.id = id;
    }
    public String getName() {
        return this.name;
    }
    public void setName(final String name) {
        this.name = name;
    }
}
```

GroovyBeans have a very useful initialization feature. We can pass a Map to the constructor of a bean that contains the names of the properties, along with an associated initialization value:

```
map = [id: 1, name: "Barney, Rubble"]
customer1 = new Customer( map )
customer2 = new Customer( id: 2, name: "Fred, Flintstone")
```

When passing the `map` directly to the `Customer` constructor, we can omit the `map` parentheses, as seen here when initializing `customer2`. However, this is another case where the method's parentheses cannot be omitted.

Every GroovyBean has this default built-in Map constructor. This constructor works by iterating the `map` object and calling the corresponding property setter for each entry in the `map`. Any entry in `map` that doesn't correspond to an actual property of the bean will cause an exception to be thrown. The beauty and simplicity of this approach is that it allows us to have absolute flexibility when initializing beans. We can name properties in any order that we want and we can omit properties if we see fit.



This feature is often referred to as **named parameters** as it gives the impression that we are providing a flexible parameter list where we name those parameters, even though we are just passing a Map object.



Assertions

Groovy has a built-in assertion keyword `assert`. The `assert` keyword can be used in conjunction with any Boolean conditional statement. Groovy assertions work just the same as Java assertions. If the asserted statement is not true, the `assert` keyword will cause a `java.lang.AssertionException` exception to be thrown. Assertions have two forms:

```
assert 1 == 1
assert 1 == 2 : "One is not two"
```

which gives the output:

```
java.lang.AssertionError: One is not two. Expression: (1 == 2)
at ConsoleScript154.run(ConsoleScript154:2)
```

The first assertion passes silently, whereas the second throws the `AssertionException` and the test inserted after the colon is injected into the exception log for more clarity.

From time to time in the text, we will use assertions as shorthand means of validating and illustrating the code. For instance, in the Account examples, we could have illustrated setting and getting property values using assertions as follows:

```
Customer customer = new Customer()
customer.setName("Carol Coolidge")
assert customer.name == customer.getName()
```

Autoboxing

Java has two ways of handling numeric values. We can either use the numeric primitive types, such as `int`, `float`, `double`, and so on, or their equivalent classes, such as `Integer`, `Float`, `Double`, and so on. Unfortunately, you can't put an `int` or any other primitive type into a collection; so you must box the `int` into an `Integer` object to put it into the collection, and unbox it if you need to use the primitive `int` type again.

From Java 1.5 onwards, Java introduced the concept of **autoboxing**, whereby primitive types are automatically promoted to their object-based equivalent when the need arises. Groovy goes a step further with the autoboxing concept. In essence, Groovy behaves as if primitives don't exist. Numeric fields of classes are stored as the declared primitive type; but as soon as we make use of that variable and assign it to a local variable, it is automatically converted to the equivalent wrapper type object. Even numeric literals behave like objects:

```
$ groovy -e "println 2.0.class.name"
java.math.BigDecimal
$
```

For all intents and purposes, you can treat any numeric value as if it is both an object-based numeric value and a primitive. You can pass a Groovy numeric object to a Java method that requires a primitive, and Groovy automatically unboxes it to the equivalent primitive.

If we need to explicitly coerce a numeric type to its equivalent primitive, for example, to call a Java method that takes a primitive parameter, we can do this unboxing with the `(as)` operator.

```
javaMethodCall(3.0 as double)
```

In most cases, this is redundant as Groovy will automatically unbox to the correct type. However, it can be useful as a hint about which method call to select when multiple methods signatures exist.

Strings

Regular Groovy strings can be defined with either the single quote ' or double quotes " characters. This makes it easy to include either type of quotes in a string literal.

```
String singleQuote = "A 'single' quoted String"
String doubleQuote = 'A "double" quoted String'
```

Strings declared with double quotes can also include arbitrary expressions by using the \${expression} syntax. Any valid Groovy expression can be included in the \${...}.

```
Customer = new Customer(name:"Daniel Dewdney")
println "Customer name is ${customer.name}"
```

Normal strings in Groovy are instances of the `java.lang.String` class. Strings that contain the \${...} syntax are instantiated as Groovy `GString` objects.

Multiline strings can be defined by surrounding them in a triple quote, which can use single quote ''' or double quote """ characters as follows:

```
String multiLine = '''Line one
Line two
"We don't need to escape quotes in a multi-line string"
Multi-line strings can be GStrings too.
'''
```

Multiline strings are useful for embedding XML, HTML, and SQL. Combined with `GStrings` they are ideal for building templates in our code.

```
name = "Daniel Dewdney"

customerSelectSQL = """
    select * from customer where name = ('${name}');
"""

```

Regular expressions

Groovy supports regular expressions natively within the language. There are three built-in convenience operators specifically for this purpose:

- The regex match operator ==~
- The regex find operator =~
- The regex pattern operator ~String

The match operator is a simple Boolean operator that takes the operands "String ==~ regex string". Match returns `true` if the string operand is a match to the regex string. Regex strings are a sequence of characters that define a pattern to apply when searching for a match within a string. They can consist of specific characters or character classes such as `\d` (any digit), and `\w` (any word character). See below for a table of some of the more commonly-used character classes.

In the simplest case, a regular expression can just consist of a sequence of regular characters. So a Regex string "Match Me" will match only strings containing exactly the characters "Match Me". In other words, it can be used to just test equality. We can use this feature to show how the different ways of expressing strings in Groovy result in the same string.

```
def matchMe = "Match Me"  
assert matchMe ==~ 'Match Me'  
assert matchMe ==~ """Match Me"""  
assert matchMe ==~ /Match Me/
```

The final assertion shown above introduces yet another Groovy string syntax, which goes under the cute name "slashy" strings. Slashy strings are most commonly used when defining regex strings, but they can be used anywhere you wish to define a string object.

The biggest advantage of the slashy string is the fact that the backslash character `\` does not need to be escaped. In a literal string, `\\"` is a single backslash. So to place the character class `\d` in a string literal, we need to write `\\\d`. To match the backslash character itself we need to write `\\\\\`. In slashy string format, these become `/\\d/` and `/\\\\/` respectively.

Groovy adds some neat usability features to Java regular expression handling but under the covers it still uses the `java.util.regex` classes. Groovy regex pattern strings are identical to their Java equivalents. The most comprehensive documentation for all of the pattern options available can be found in your Java SE Java Doc under the class documentation for `java.util.regex.Pattern`. Below is a truncated list of some of the more commonly-used patterns.

Construct	Matches
.	Any character
^	Start of a line
\$	End of a line
X	The character x
\d	A digit character
\D	Any character except a digit

Construct	Matches
\s	A whitespace character
\S	Any character except whitespace
\w	A word character
\W	Any character except word characters
\b	A word boundary
(x y z)	x or y or z. i.e. (apple orange pear)
[a-f]	Character class containing any character between a and f
[abc]	a, b, or c

The Groovy find operator (`=~`) is similar to match but returns a `java.util.regex.Matcher` object. Below we use `find` to return a matcher for all three-letter words with a middle letter of "o". The pattern we use is `/\b.o.\b/`. Groovy allows us to use the collection convenience method "each" to iterate over the resulting matches and invoke a closure on each match to output the result. (More on collections and closures shortly.)

```
def quickBrownFox = """
The quick brown fox
jumps over the lazy
dog.
"""
matcher = quickBrownFox =~ /\b.o.\b/
matcher.each { match -> println match }
```

which outputs:

```
fox
dog
```

Every time we use the `match` and `find` operators, behind the scenes, Groovy transforms the regex string into a `java.util.regex.Pattern` object and compiles it. The pattern operator does the same thing, and transforms the string it operates on into a compiled `Pattern` object. For most applications, using `find` and `match` directly on a pattern string is fine, because the overhead of transformation and compilation to a `Pattern` object is not significant. The rationale behind the pattern operator is that complex patterns are often expensive to compile on demand, so the precompiled pattern object will be faster to use.

A simple change to the previous code is all that is required to use a precompiled pattern instead:

```
def quickBrownFox = """
The quick brown fox
jumps over the lazy
dog.
"""

pattern = ~/b.o.b/
matcher = pattern.matcher(quickBrownFox)

matcher.each { match -> println match }
```

Methods and closures

Closures will be dealt with in detail in the next chapter, so we won't go into them in depth here. In order to do justice to the Groovy control structures and the special built-in support Groovy has for collections, we need to take just a brief excursion into closures for now.

Closures are snippets of Groovy program code enclosed in curly braces. They can be assigned to an instance property, or a local variable, or even passed as parameters to a method. In Java, program logic can only be found in class methods. The inclusion of static member functions in classes gives some flexibility to Java in allowing methods to be invoked outside of the context of an object instance.

In Groovy, methods can exist both inside classes and outside of classes. We know already that Groovy scripts get compiled to classes that have the same name as the script. Groovy methods within scripts just get compiled into member methods of the script class. Groovy has a slightly different syntax from Java to support the concept of a dynamic return type.

Groovy methods look very similar to Java methods except that public visibility is the default, so the `public` keyword can be left out. In addition, Groovy methods support optional arguments, as do closures. As with dynamic variables, we need to use the `def` keyword when defining a method that has a dynamic return type:

```
// Java method declaration
public String myMethod() {
    ...
}

// Groovy method declaration
String myMethod() {
    ...
}
```

```

}
// And with dynamic return type
def myMethod() {
}
```

Groovy script methods, which are declared in the same script, can be called directly by name:

```

def greet(greeting) {
    println greeting + ", World!"
}

greet ("Hello")
greet ("Goodbye")
```

The previous code gives the output:

```
Hello, World!
Goodbye, World!
```

Class methods are called by object reference, similar to Java.

```

class Greeting {
    def greet(greeting) {
        println greeting + ", World!"
    }
}
greeting = new Greeting()
greeting.greet ("Hello")
greeting.greet ("Goodbye")
```

Closures can look deceptively similar to method calls in their usage. In the next code snippet, we create a variable called `greet` and assign a closure to it. This closure is just a snippet of code enclosed in braces, which prints a greeting. Regular methods have their own local scope and can only access variables defined within that scope or member fields in the containing class. Closures, on the other hand, can reference variables from outside their own scope, as illustrated below. Closures can be invoked by applying the method call syntax to the variable containing the closure.

```

greeting = "Hello"
def greet = {
    println "${greeting}, World!"
}

greet()
greeting = "Goodbye"
greet()
```

Closures can also accept parameters but they have their own particular syntax for doing so. The next closure accepts a parameter, `greeting`. Multiple parameters can be defined by separating them with commas. Parameters can also have an optional type annotation.

```
def greet = { greeting ->
    println "${greeting}, World!"
}

def greetPerson = { String greeting, name ->
    println "${greeting}, ${name}!"
}

greet("Hello")
greetPerson("Goodbye", "Fred")
```

We can pass a closure as a method parameter. Many useful collection methods take a closure as a parameter. The list `each()` method takes a closure as its parameters. The `each()` method iterates over a list and applies the closure to each element in the list.

```
def fruit = ["apple", "orange" , "pear"]
def likeIt = { String fruit -> println "I like " + fruit + "s"}
fruit.each(likeIt)
```

The above code outputs:

```
I like apples
I like oranges
I like pears
```

Now, if we look back at our matcher example, at first glance it seems to be using some specialized collection iteration syntax.

```
matcher.each { match -> println match }
```

But if we remember that `matcher` is a collection of matches, and that parentheses in Groovy are optional, we can see that all that is happening here is that a closure is passed to the `each` method of the `matcher` collection. We could have written the same statement as:

```
matcher.each ({ match -> println match })
```

Groovy also has a neat shorthand for closures, which have just one parameter. We don't need to explicitly name this parameter and can just refer to it as `it`. So our `matcher` statement can be even more succinct:

```
matcher.each { println it }
```

Control structures

Groovy supports the same logical branching structures as Java. The Groovy versions of the common branching are identical in structure to those of Java:

```
// Simple if syntax
if (condition) {
...
}

// If else syntax
if (condition) {
...
}

// Nested if then else syntax
if(condition) {
...
} else if (condition) {
} else {
}
```

Groovy truth

The only difference is in how Groovy interprets `if` conditions. Groovy can promote a number of non-Boolean conditions to `true` or `false`. So, for instance, a non-zero number is always `true`. This wider, more all-encompassing notion of what can be `true` is often referred to as "**Groovy Truth**".

```
// Java non zero test
int n = 1;
if ( n != 0) {
...
}
// Groovy equivalent does not need to form a boolean expression
def n = 1;
if (n) {
...
}
```

Other "**Groovy Truths**" are as shown below. In other words, when taken in the context of a predicate, these values will all equate to a Boolean `true` value.

- Any non-null value
- Non-empty strings
- An initialized collection
- A matcher with valid matches

Now let's look at some "**Groovy Falsehoods**". Things which when used as a predicate will equate to a Boolean `false` value are:

- A zero value
- A null object value
- An empty string
- An empty collection
- An array of zero length

As with many Groovy language features, Groovy's loose interpretation of what can be "true" allows for much more succinct and understandable branching conditions.

Groovy Truth applies to Groovy assertions, too. So we can use any of the above non-Boolean conditions as an assertion, even though they do not necessarily return a Boolean `true` or `false`.

```
// ensure string is not empty
String empty = ''
assert !empty
```

Ternary and Elvis operators

The standard Java ternary operator (`a ? b : c`) is supported. Groovy also has another similar operator, the bizarrely-named Elvis operator (`a ?: b`). We can express the ternary operation as a traditional `if - else` branch as follows:

```
// Ternary operator
x > 0 ? y = 1 : y = 2

// Is same as
if (x > 0)
    y = 1
else
    y = 2
```

The Elvis operator's behavior is best illustrated as a version of the ternary operator. So `(a ?: b)` is equivalent to `(a ? a : b)`. The use of the Elvis operator makes more sense in the light of our previous discussion on Groovy Truth where the Boolean condition used can be something other than a regular expression.

```
// Elvis operator
y = x ?: 1

// Is the same as
y = x?x:1
```

```
// Which with regular if - else branching and conditions would be
if (x != 0)
    y = x
else
    y = 1
```

The Elvis operator has the added benefit of avoiding a second evaluation of the initial predicate. This may be important if `x` is either expensive to evaluate, has unwanted side effects, or results in an operation that we don't necessarily want to repeat (such as a database retrieval). The Elvis operator works by maintaining a hidden local variable, which stores the initial result. If that result is true according to the rules of **Groovy Truth**, then that value is returned, otherwise the alternative value is used.

Suppose that we want to retrieve shopping cart items in a Map so that we can display a list of selected items. If the check database and the cart contain entries, then that is the Map that we want to display. If there are no items in the cart, then we want to return a Map, which contains a dummy entry to display that just says that the cart is empty. If we use regular conditional logic, we can't use a ternary operator because we don't really want to check the cart twice. We would have to write something like the following to manage a temporary map while we decide what to do with it:

```
cartItemsMap = Cart.getItems()

if ( cartItemsMap ) // Groovy true if map has entries in it
    return cartItemsMap
else
    return [-1: "empty"]
```

`Cart.getItems` returns a Map, which in Groovy Truth is `true` if it has elements and `false` if it is empty. Knowing this we can rewrite the same code as a succinct one liner.

```
return Cart.getItems() ?: [-1: "empty"]
```

Switch statement

Groovy adds some neat features to the `switch` statement by adding some extra options that can be tested in the `case` expression.

```
switch (x) {
    case 1:
        // if x is number 1 we end up here
        break;
```

```
case "mymatch":  
// if x equals string "mymatch" we end up here  
break;  
case /.o./:  
// if x is a string and matches regex /.o./ we end up here  
break;  
  
case ["apple", "orange", "pear", 1, 2, 3]:  
// if x is found in the list we end up here  
break;  
  
case 1..5:  
// if x is one of the values 1, 2, 3, 4 or 5 we end up here  
break;  
}
```

Loops

Groovy does not support the traditional Java `for(initializer: condition: increment)` style of looping or the `do { } while (condition)` style of looping. It does support traditional `while` loops, as follows:

```
int n = 0;  
while (n++ < 10) {  
    println n  
}
```

Groovy makes up for this lack of looping options with its own styles of looping. Groovy loops are simpler and in many ways more powerful than the Java equivalent. In Groovy, we can iterate over any range of values, as follows:

```
for (n in 0..10)  
    println n
```

We can iterate all the values of a list without any funky `Iterator` objects:

```
for (x in ["apple", "orange", "pear"])  
    println x
```

In fact, as we will see in the following section on collections, we can use the "`in`" expression to iterate over any collection type. We can even iterate over the characters in a string, as follows:

```
def hello = "Hello, World!"  
for (c in hello)  
    println c
```

Collections

Groovy enhanced the Java collection classes by adding to and improving on the declaration syntax and additional convenience methods. It also adds the new collection type range, which provides a special purpose syntax for managing ranges of values.

Ranges

Groovy supports the concept of a **range of values**. We define a range of values with the range operator "...". So a range of integers from 1 to 10 is defined with `1..10`. A range value can be any object that belongs to a class that defines the `previous()`, `next()` methods and implements the `Comparable` interface. We saw previously how we can use the `for (variable in range)` style loop to iterate through a range. We can define ranges that are inclusive or exclusive as follows:

```
// Iterate over an inclusive range of integer values
// (all integers 1 to 10)
for (n in 1..10)
    println n
// Iterate an exclusive range of characters
// (all chars 'a', 'b', 'c' and 'd'
for(c in 'a' .. < 'e')
    println c
```

Range objects have two properties, `to` and `from`, that define their limits as shown below:

```
def numbers = 1..10
def letters = 'a'..<'e'
assert numbers.from == 1
assert numbers.to == 10
// For an exclusive range "to" is the last actual value
// that will be listed
assert letters.to == 'd'
```

Ranges are implemented under the covers by the `java.util.List` class. This means that all of the Java APIs that are available on a `List` object can also be applied to a range:

```
// Use java.util.List.contains() on range
def numbers = 1..10
assert numbers.contains 5
```

We can also use the `in` keyword as part of a predicate to test if a particular value is contained within the range.

```
assert 5 in numbers
```

Lists

List declarations look like array declarations in Java. Lists declared in this way are in fact `java.util.List` objects. Let's prove the last statement that ranges are equivalent to lists:

```
// Ranges are just lists with values in sequence
def numberList = [1,2,3,4,5,6,7,8,9,10]
def numberRange = 1..10
assert numberList == numberRange
// We can declare lists within lists and contents can be heterogeneous
def multidimensional = [1,3,5,[{"apple","orange","pear"}]]
```

There are some useful list operators that we can use as shortcuts.

```
// Add to lists together
assert [1,3,5] + [{"apple","orange","pear"}] == multidimensional
assert [1,3,5] << {"apple","orange","pear"} == multidimensional
// Subtract elements
assert multidimensional - [{"apple","orange","pear"}] == [1,3,5]
```

There are also some convenience functions that make list management easier, as follows:

```
// flatten that multi dimensional list
assert multidimensional.flatten() == [1,3,5,"apple","orange","pear"]
// reverse list elements
assert [1,3,5].reverse() == [5,3,1]
// build a new list having applied a closure to each element to
increment all values
assert [1,3,5].collect { it + 1 } == [2,4,6]
// apply a regex pattern match to all elements of the list
def animals = ["cat", "dog", "fox", "cow"]
assert animals.grep( ~/o./ ) == ["dog", "fox", "cow"]
// sort list elements
assert [5,3,1].sort() == [1,3,5]
// find list element matching an expression
assert animals.find { it == "dog" } == "dog"
```

We can iterate over a list in both directions by applying a closure to each item.

```
def list = [1,3,5]
def number = ''
// Iterate forwards
list.each { number += it }
assert number == '135'
number = ''
// Now iterate forwards
list.reverseEach { number += it }
assert number == '531'
```

Groovy adds two new methods to lists: `any` and `every`. These return a Boolean if any or every member of the list, respectively, satisfies the given closure.

```
def list = [1,2,3,5,7,9]

// Are any members even
assert list.any { it % 2 == 0 }
// Is every member even ... not true
assert ! list.every { it % 2 == 0 }
```

Maps

The declaration syntax for maps is very similar to that of lists. We declare a map as a list of key or value pairs delimited by colons. Groovy is flexible in what type of objects can be used as keys or values. In principle, any object that has a `hashCode` function that returns consistent values can be used as either a key or value in a map. By consistent I mean that any specific value that we define for the object will always return the same `hashCode` value. Let's start by looking at maps by using strings as keys.

```
// declare a simple map
def fruitPrices = ["apple":20, "orange":25, "pear":30]

// we can subscript a map with any key value
assert fruitPrices["apple"] == 20
// or use the key like it was a property
assert fruitPrices.apple == 20

// we can explicitly declare a variable that is empty but is a map
def empty = [:]
assert empty.size() == 0

// when declaring keys of type String we can leave out the quotes
assert fruitPrices = [apple:20,orange:25,pear:30]
```

```
// we can retrieve a value using the get method
assert fruitPrices.get("apple") == 20
// and supply a default value for items that are not found
assert fruitPrices.get("grape", 5) == 5

// assignment can be done via superscript or property syntax
fruitPrices['apple'] = 21
assert fruitPrices['apple'] == 21
fruitPrices.apple = 22
assert fruitPrices['apple'] == 22

// assignment automatically adds an item if it does not exist
fruitPrices.grape = 6
assert fruitPrices == [apple:22,orange:25,pear:30, grape:6]
```

Maps support the plus operator for adding maps together, but not the minus operator for taking away.

```
def fruit = [apple:20, orange:25 ]
def veg = [pea:1, carrot:15]

assert fruit + veg == [apple:20, orange:25, pea:1, carrot:15]

// map equality is agnostic to order
assert fruit + veg == [pea:1, carrot:15, apple:20, orange:25]
```

The most common use of maps in Groovy is with string keys. When we use a String as key, we can interchangeably use the key with or without quotes. We can also look up the value by using the subscript operator, or by using the property reference semantics. Because any object can be a key, this allows us to define some unusual looking maps:

```
def squares = [ 1:1, 2:4, 3.0:9]

assert squares[1] == 1
assert squares[2] == 4
assert squares[3.0] == 9
```

Here we see how we can use what seem to be primitive numeric values as keys. Because Groovy autoboxes these primitives into their equivalent wrapper object and these wrappers implement consistent hashCode methods, we can use them as keys. Not only that; we can mix the type of object that we use as a key.

We also can use object values as keys, but in order to do so we need to add parentheses around them to assist the compiler in determining our intention. Below, we add two keys to the map. The first key is a string, "apple", but the second is the value contained in the apple local variable, which is 1.

```
def apple = 1
def map = [ apple:"Red", (apple):"Green"]
assert map[1] == "Green"
assert map["apple"] == "Red"
```

Operators

Groovy implements all of the usual operators that we expect from Java, and adds a number of unique operators of its own. We have already encountered some of these in the preceding sections, such as the Elvis operator (? :), the manual type coercion operator (as) and the regex match (==~), find (=~), and pattern operators (~).

Spread and spread-dot

Collections also support some useful operators such as the spread-dot operator (*.). Spread-dot is used when we need to apply a method call or make a field or property accessible across all members of a collection. This is best illustrated with some examples as shown below:

```
def map = [a:"apple", o:"orange", p:"pear"]
def keys = ["a", "o", "p"]
def values = ["apple", "orange", "pear"]
// use spread dot to access all values
assert map*.key == keys
assert map*.value == values
// which is equivalent to using the collect method
assert map.collect { it.key } == keys
assert map.collect { it.value } == values
```

We can use spread-dot to invoke a method across all members of a list.

```
class Name {
    def name
    def greet(greeting) {
        println greeting + " " + name
    }
}

def names = [ new Name(name:"Aaron"),
             new Name(name:"Bruce"),
             new Name(name:"Carol")]

names*.greet("Hello")
```

This gives the following output:

```
Hello Aaron  
Hello Bruce  
Hello Carol
```

A close relative of spread-dot is the spread operator. Spread has the effect of tearing a list apart into its constituent elements.

```
def greetAll ( a, b, c) {  
    println "Hello " + a + "," + b + "," + c  
}  
  
// Spread the names  
greetAll(*names.name)
```

which outputs:

```
Hello Aaron,Bruce,Carol
```

Null safe dereference

One of my favorite operators in Groovy is the null safe dereference operator. How many times in your programming career with Java have you needed to write the following:

```
Customer customer = getCustomerFromSomewhere();  
if (customer != null) {  
    String name = customer.name;  
}
```

Groovy provides a neat `(?.)` operator that automatically does the null check for you before dereferencing, so the above can be written as follows:

```
Customer customer = getCustomerFromSomewhere();  
String name = customer?.name;
```

All told, these syntactical shortcuts make for much more concise and readable code. Freed from the syntactical sugar of Java, we can write cleaner code more quickly. Once you've gotten used to this shorthand, going back to Java is like wearing a pair of lead boots.

Operator overloading

Java inherited many features from the C++ language, with one notable exception – operator overloading. Groovy implements operator overloading as a language feature, which means that any class can implement its own set of operators. In the simplest case, we can overload the arithmetic operators and make any object of the class behave as if it is a numeric value. Operators are overloaded by implementing the corresponding operator method in the class, for example, the `plus()` method to implement addition.

The Groovy version of the `Date` class implements some operators including the `plus()` and `minus()` operators. Operator overloading is a fundamental feature in implementing DSLs, so we will go into this feature in significant detail later in the book.

```
def today = new Date()
def tomorrow = today + 1
def yesterday = today - 1
println yesterday
println
println today
println tomorrow

assert today.plus(1) == tomorrow
assert tomorrow.minus(1) == today
```

Summary

In this chapter, we have conducted a whistle-stop tour of the Groovy language. We have touched on most of the significant features of the language as a part of this tour. In the subsequent chapters, we will delve deeper into some of these features, such as operator overloading. We will also cover some of the more advanced features that have not been touched on here such as Builders and Meta programming. However, this book is not intended to be a complete tutorial on the Groovy language and I recommend you delve further into the language by reading the Groovy User guide, which is available at: <http://groovy.codehaus.org/User+Guide>.

3

Groovy Closures

In this chapter, we will focus exclusively on closures. We have touched upon closures already in the previous chapter. Now we will take a close look at them from every angle. Why devote a whole chapter of the book to one aspect of the language? The reason is that closures are the single most important feature of the Groovy language. Closures are the special seasoning that helps Groovy stand out from Java. They are also the single most powerful feature that we will use when implementing DSLs.

- We will start by explaining just what a closure is and how we can define some simple closures in our Groovy code.
- We will look at how many of the built-in collection methods make use of closures for applying iteration logic, and will see how this is implemented by passing a closure as a method parameter.
- We will look at the various mechanisms for calling closures, and we take a look under the covers at how groovy implements its various `doCall()` methods for different parameter types.
- We will go into some depth on how parameters are passed to closures, including a discussion on optional type annotations, default parameter values, and how to curry parameters.
- We will take a look at how return values are handled in closures, and finally we will look into how scope affects closures, particularly the field variables that are visible in surrounding scopes.

A handy reference that you might want to consider having at hand while you read this chapter is GDK Javadocs, which will give you full class descriptions of all of the Groovy built-in classes, but of particular interest here is `groovy.lang.Closure`.

What is a closure

Closures are such an unfamiliar concept to begin with that it can be hard to grasp initially. Closures have characteristics that make them look like a method in so far as we can pass parameters to them and they can return a value. However, unlike methods, closures are **anonymous**. A closure is just a snippet of code that can be assigned to a variable and executed later.

```
def flintstones = ["Fred", "Barney"]
def greeter = { println "Hello, ${it}" }
flintstones.each( greeter )
greeter "Wilma"
greeter = {}
flintstones.each( greeter )
greeter "Wilma"
```

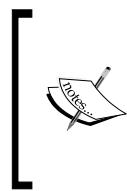
Because closures are anonymous, they can easily be lost or overwritten. In the above example, we defined a variable `greeter` to contain a closure that prints a greeting. After `greeter` is overwritten with a null closure, the original closure is lost.



It's important to remember that `greeter` is not the closure. It is a variable that contains a closure, so it can be supplanted at any time.

Because `greeter` has dynamic type, we could have assigned any other object to it. All closures are a sub class of type `groovy.lang.Closure`. Because `groovy.lang` is automatically imported, we can refer to `Closure` as a type within our code. By declaring our closures explicitly as `Closure`, we cannot accidentally assign a non-closure to them.

```
Closure greeter = { println it }
```



For each closure that is declared in our code, Groovy generates a `Closure` class for us, which is a subclass of `groovy.lang.Closure`. Our closure object is an instance of this class. Although we cannot predict what exact type of closure is generated, we can rely on it being a subtype of `groovy.lang.Closure`.

Closures and collection methods

In the last chapter, we encountered Groovy lists and saw some of the iteration functions, such as the `each` method.

```
def flintstones = ["Fred", "Barney"]  
  
flintstones.each {  
    println "Hello, ${it}"  
}
```

This looks like it could be a specialized control loop similar to a `while` loop. In fact, it is a call to the `each` method of `Object`. The `each` method takes a closure as one of its parameters, and everything between the curly braces `{ }` defines another anonymous closure.

Closures defined in this way can look quite similar to code blocks, but they are not the same. Code defined in a regular Java or Groovy style code block is executed as soon as it is encountered. With closures, the block of code defined in the curly braces is not executed until the `call()` method of the closure is made.

```
println "one"  
def two =  
{  
    println "two"  
}  
println "three"  
two.call()  
println "four"
```

Will print the following:

```
one  
three  
two  
four
```

Let's dig a bit deeper into the structure of the `each` of the calls shown above. I refer to `each` as a call because that's what it is – a **method call**. Groovy augments the standard JDK with numerous helper methods. This new and improved JDK is referred to as the Groovy JDK, or GDK for short. In the GDK, Groovy adds the `each` method to the `java.lang.Object` class. We will discover later in the book how to inject methods into existing classes ourselves. The signature of the `each` method is:

```
Object each(Closure closure)
```

The `java.lang.Object` class has a number of similar methods such as `each`, `find`, `every`, `any`, and so on. Because these methods are defined as part of `Object`, you can call them on any Groovy or Java object. They make little sense on most objects, but they do something sensible if not very useful.

```
def number = 1
number.each { println it }
```

will print out:

1

These methods all have specific implementations for all of the collection types, including arrays, lists, ranges, and maps. So what is actually happening when we see the call to `flintstones.each` is that we are calling the List's implementation of the `each` method. Because `each` takes a `Closure` as its only parameter, the following code block is interpreted by Groovy as an anonymous `Closure` object to be passed to the method.

The actual call to the closure passed to `each` is deferred until the body of the `each` method itself is called. The closure may be called multiple times—once for every element in the collection.

Closures as method parameters

We know already that parentheses around method parameters are optional, so the previous call to `each` can also be considered equivalent to:

```
flintstones.each ({ println "Hello, ${it}" })
```

Groovy has a special handling for methods whose last parameter is a closure. When invoking these methods, the closure can be defined anonymously after the method call parenthesis. So yet another legitimate way to call the above is:

```
flintstones.each() { println "Hello, ${it}" }
```

The general convention is not to use parentheses unless there are parameters in addition to the closure:

```
def flintstones = ["Fred", "Barney", "Wilma"]

println flintstones.findIndexOf(0) { it == "Wilma" }
```

which starts at index 0 in the list and applies the comparison closure to each element until it finds the first match. In this case, it outputs:

2

We can define our own methods that accept closures as parameters. The simplest case is a method that accepts only a single closure as a parameter.

```
def closureMethod(Closure c) {  
    c.call()  
}  
  
closureMethod {  
    println "Closure called"  
}
```

Method parameters as DSL

This is an extremely useful construct when we want to wrap a closure in some other code. Suppose we have some locking and unlocking that needs to occur around the execution of a closure. Rather than require the writer of the code to do this locking via a locking API call, we can implement the locking within a locker method that accepts the closure:

```
def locked(Closure c) {  
    callToLockingMethod()  
    c.call()  
    callToUnLockingMethod()  
}
```

The effect of this is that whenever we need to execute a locked segment of code we simply wrap the segment in a locked closure block as follows:

```
locked {  
    println "Closure called"  
}
```

In a small way, we are already writing a mini DSL when we use these types of constructs. This call to the locked method looks, to all intents and purposes, like a new language construct. We will be using this again and again in our DSL examples later in the book.

When writing methods that take other parameters in addition to a closure, we generally leave the `Closure` argument to last. Groovy has a special syntax handling for these methods, and allows the closure to be defined as a block after the parameter list when calling the method.

```
def closureMethodString(String s,Closure c) {  
    println s  
    c.call()  
}
```

```
closureMethodString("Line one") {  
    println "Line two"  
}
```

will print the following:

```
Line one  
Line two
```

Forwarding parameters

Parameters passed to the method may have no impact on the closure itself, or they may be passed to the closure as a parameter. Methods can accept multiple parameters in addition to the closure. Some may be passed to the closure, while others may not.

```
def closureMethodString(String s, Closure c) {  
    println "Greet someone"  
    c.call(s)  
}  
  
closureMethodString("Dolly") { name ->  
    println "Hello, ${it}"  
}
```

will print the following:

```
Greet someone  
Hello, Dolly
```

This construct can be used in circumstances where we have look-up code that needs to be executed before we have access to an object. Say we have customer records that need to be retrieved from a database before we can use them:

```
def withCustomer (id, c ) {  
    def cust = getCustomerRecord(id)  
    c.call(cust)  
}  
  
withCustomer(12345) { customer ->  
    println "Found customer ${customer.name}"  
}
```

We could write an `updateCustomer` method that saves the customer record after the closure is invoked and amend our `locked` method to implement transaction isolation on the database, as follows:

```
class Customer {  
    String name  
}  
def locked (Closure c) {  
    println "Transaction lock"  
    transactionLock()  
    c.call()  
    transactionRelease()  
    println "Transaction release"  
}  
  
def updateCustomer (id, c) {  
    // We have only one customer anyway  
    println "get customer record"  
    Customer cust = getCustomerRecord(id)  
    println "Customer name was ${cust.name}"  
    c.call(cust)  
    println "Customer name is now ${cust.name}"  
    println "save customer record"  
    saveCustomerRecord(cust)  
}
```

At this point we can write code that nests the two method calls by calling `updateCustomer` as follows:

```
locked {  
    updateCustomer(12345) { customer ->  
        customer.name = "Barney"  
    }  
}
```

This outputs the following result, showing how the update code is wrapped by `updateCustomer`, which retrieves the `customer` object and subsequently saves it. The whole operation is wrapped by `locked`, which includes everything within a transaction:

```
Transaction lock  
get customer record  
Customer name was Fred  
Customer name is now Barney  
save customer record  
Transaction release
```

Calling closures

In our previous examples, we were passing closures to the built-in collection methods. In the examples to date, we have deferred to the collection method to do the closure invocations for us. Let's now look at how we can make a call to the closure ourselves. For the sake of this example, we will ignore the fact that the GDK provides versions of the `Thread.start` method that achieves the same thing.

```
class CThread extends Thread {
    Closure closure

    CThread( Closure c ) {
        this.closure = c
        this.start()
    }
    public void run() {
        if (closure)
            closure() // invoke the closure
    }
}

CThread up = new CThread(
{
    [1..9]* each {
        sleep(10 * it)
        println it
    }
} )

CThread down = new CThread(
{
    ["three", "two", "one", "liftoff"] each {
        sleep(100)
        println it
    }
} )
```

Here we define a subclass of the Java `Thread` class, which can be constructed with a closure. The `run` method of the `Thread` invokes the closure using an unnamed `()` invocation on the closure field. The `CThread` constructor automatically starts the thread. We can invoke a closure in three different ways, as follows:

- Using the unnamed () invocation syntax as described above:

```
public void run() {  
    closure()  
}
```

- By calling the call() method of groovy.lang.Closure:

```
public void run() {  
    closure.call()  
}
```

- By calling the doCall() method of the closure itself:

```
public void run() {  
    closure.doCall()  
}
```

The call and doCall methods might seem redundant, but there is an important distinction. The call method is part of the groovy.lang.Closure class and can accept any number of dynamic arguments. The doCall method is generated dynamically for each closure that we define in our code, and has a signature that is specific to the individual closure.

The doCall method for the following closure will only accept a single string as its parameter list.

```
def closure = { String s -> println s }
```

The general convention is to use either the unnamed () syntax or groovy.lang.Closure.call() when invoking closures.

This example was useful as an illustration of how to call a closure that you have saved in a member field or variable. However, I think you will agree that the built-in Thread.start method taking a closure is far more elegant.

```
Thread.start  
{  
    [1..9]*.each {  
        sleep(10 * i)  
        println i  
    }  
}  
  
Thread.start  
{  
    ["three", "two", "one", "liftoff"] .each {  
        sleep(100)  
        println i  
    }  
}
```

Finding a named closure field

All of the above techniques for calling a closure rely on us having prior knowledge of a field or variable that contains a closure. This is fine for most straightforward applications of closures, but this book is about developing DSLs, so let's dig a little deeper.

Take the Grails application framework as an example. You can download Grails from <http://grails.org/Download>. Grails uses closures as a neat way of defining actions for its user interface controllers. No further configuration is required for the Grails runtime to be able to dispatch requests to an action.

```
class UserController {  
    ...  
    def login = {  
        ... Login closure code  
    }  
}
```

We can implement a login action for our user controller in Grails- simply by declaring a closure in the controller class and assigning it to a field called `login`. In the UI, Grails provides tags to automatically create a link that will dispatch to our login action.

```
<g:link controller="user" action="login">Login</g:link>
```

The Grails runtime can find the appropriate action, given the closure field name as a string, and call that closure when the user clicks on the link. We can achieve the same effect by simply using Java reflection.

```
class MyController {  
    def public myAction = {  
        println "I'm an action"  
    }  
}  
  
void callPublicClosureField(Class clazz, String closure ) {  
    def controller = clazz.newInstance()  
    controller.getClass()  
        .getDeclaredField(closure).get(controller).call()  
}  
  
callPublicClosureField(MyController.class, "myAction")
```

Here we are using Java reflection to access a public field in our controller class. We then invoke this closure field with the `call` method. Java reflection honors class visibility, so we need to make the field `public` in order to be able to access it. Later in the book, we will explore other methods that allow us to access static and private fields in both classes and scripts.

The ability to write code such as:

```
def something = {  
    snippet of code  
}
```

and have a separate runtime that makes sense of this will be the key to writing some of our DSLs later in the book.

Closure parameters

In our previous examples, we have made use of the `it` keyword. When a closure accepts only a single parameter, we are able to refer to this parameter as `it` and are free from having to explicitly define the parameter. The possible syntax definitions for a closure are:

```
// Default case. Will allow any parameters to be passed to the closure  
{ statement-list }  
// Closure does not accept any parameters  
{ -> statement-list }  
// Closure can accept one to many parameters with  
// optional type annotations.  
{ [type] param (,[type] param)* -> statement-list }
```

The parameter list is a comma-separated list of parameter names with optional type definitions. Closures behave slightly different depending on whether we supply the optional type.

```
def list = [1,3,5,7]  
def defaultParams = { println it; }  
def dynamicParams = { something -> println something; }  
def intParams = { int something -> println something; }  
def stringParams = { String something -> println something; }  
  
list.each defaultParams  
list.each dynamicParams  
list.each intParams  
list.each stringParams // Fails
```

The above examples illustrate the use of dynamic versus static typing for closure parameters. The first three closures work fine with a list of integers. The last will fail because an exception will get thrown when it attempts to invoke the closure with an int value.

Parameters and the doCall method

The exception that is thrown relates to the generated `doCall()` method of the closure. We know that Groovy generates closure classes for each closure that we define in our code. Therefore, we can imagine that Groovy is generating the following closure classes on our behalf for the above examples.

- For a closure with no explicit parameter defined, we can expect a `doCall` that accepts varargs to be generated. So `doCall` for this closure will accept any parameter that we pass to it.

```
def defaultParams = { println it; }

class Closure1 extends groovy.lang.Closure{
    def doCall(Object [] params ) {
    }
}
closure1 = new Closure1()

closure1.doCall("hello")
closure1.doCall("hello",1,0.1)
```

- For a closure accepting only one dynamically-typed parameter, we would expect our `doCall` to also accept a single parameter. We can pass any value to this `doCall`, but should expect an exception if we pass more than one parameter.

```
def dynamicParams = { something -> println something; }
class Closure2 extends groovy.lang.Closure{
    def doCall(something) {
    }
}

closure2 = new Closure2()

closure2.doCall("hello")
closure2.doCall(1)
closure2.doCall("hello",1,0.1) // exception
```

- A closure that accepts typed parameters will have a `doCall` method that accepts only the same specific types as the closure parameters to be generated.

```
def stringParams = { String something -> println something; }

class Closure3 extends groovy.lang.Closure{
    def doCall(String s) {
    }
}

closure3 = new Closure3()

closure3.doCall("hello")
closure3.doCall(1) // exception

class Closure4 extends groovy.lang.Closure{
    def doCall(int s) {
    }
}

closure3 = new Closure3()

closure3.doCall("hello") // exception
closure3.doCall(1)
```

For this reason, closures are often best used with dynamically-typed parameters, as it is difficult to guard against the side effects. Consider the code shown below. We would probably prefer if the closure were to be applied to the whole of the list and not to only a part of it. In this case, the `each` method will process all of the elements in the list up until it encounters the string "nine" element, which causes an exception to be thrown.

```
def list = [1,3,5,7, "nine"]

def intParams = { int something -> println something; }

list.each intParams // Fails when we hit list[4]
```

This will output the first four elements of the list and will fail when it reaches "nine". As we have seen earlier, the calling of the closure happens within the body of the each method. Therefore, we don't have an opportunity to wrap the closure calls within an appropriate try catch block. Even if we wrap the whole call to each with a try catch, we still don't know how to undo any unwanted actions.

```
def list = [1,3,5,7, "nine"]

def intParams = { int something -> println something; }

try {
    list.each intParams // Fails when we hit list[4]
} catch (e) {
    // We cannot easily undo actions applied to part of a list
}
```

Passing multiple parameters

Closures can accept multiple parameters. The call method and unnamed () closure invocation both allow us to pass as many parameters as we wish. If a closure accepts a parameter and we fail to pass a value for it, a null value is passed instead.

```
def greet = { println "Hello, ${it}" }

greet()
```

will output:

Hello, null

If we pass more parameters than the closure expects, then the extra parameters are simply ignored, except where we explicitly define the closure to have no parameters, as in the next section below.

```
def greet = { name -> println "Hello, ${name}" }

greet "Hello", "Dolly"
```

will output:

Hello, Hello

To accept multiple parameters, we list the parameters in order before the -> symbol.

```
def greet = { greeting, name -> println greeting + ", " + name; }

greet("Hello", "Dolly")
```

which outputs:

Hello, Dolly

Enforcing zero parameters

Sometimes we would like to explicitly define a closure as having no parameters. Passing a parameter to a closure like this will result in an exception being thrown.

```
def greet = { -> println "Hello,World!" }

def exception = false
try {
    greet "Hello"
} catch (e) {
    exception = true
}
assert exception
```

Default parameter values

We can define default parameters by supplying a value in the parameter list, as follows:

```
def greetString = {greeting, name = "World" ->
    return "${greeting}, ${name}!"
}

assert greetString("Hello") == "Hello, World!"
assert greetString("Hello", "Dolly") == "Hello, Dolly!"
```

However, beware, when providing default parameters, of how Groovy will scan the parameter values provided and apply them to the parameter list. When we invoke a closure without some parameters, Groovy will make the best effort to apply the supplied parameters by skipping the ones that have defaults. Take a note below of how providing type annotations changes the way, Groovy applies parameter values.

```
def defaultParams1 = {
    one = "one" ,two = 2, three -> return "${one} ${two} ${three}"
}

// Defaults first two params and sets third
assert defaultParams1( "trois" ) == "one 2 trois"
// defaults provide enough type hints to allow second param
// to be defaulted
```

```
assert defaultParams1( "un" , "trois" ) == "un 2 trois"
// int parameter is shunted to cover third parameter
assert defaultParams1( "un", 2 ) == "un 2 2"

def defaultParams2 = { String one = "one",
                      int two = 2,
                      three = "three" ->
    return "${one} ${two} ${three}"
}

// Defaults exist for everything so the supplied values are applied
// left to right
assert defaultParams2( "trois" ) == "trois 2 three"
assert defaultParams2( "un", 3 ) == "un 3 three"

def defaultParams3 = { String one = "one",
                      int two = 2,
                      String three ->
    return "${one} ${two} ${three}"
}
// Default values are redundant. By providing type annotations we can
// only invoke the closure when we pass all three parameters
assert defaultParams3( "un" , 3, "trois" ) == "un 3 trois"

def defaultParams4 = { String one = "one",
                      int two = 2,
                      String three = "three" ->
    return "${one} ${two} ${three}"
}

// by providing type annotations and defaults for all parameters
// we have the most flexibility
assert defaultParams4( "un" ) == "un 2 three"
assert defaultParams4( "un", 3 ) == "un 3 three"
assert defaultParams4( "un" , 3, "trois" ) == "un 3 trois"
```

Without type annotations, the parameter values are just applied in sequence and with the defaults being skipped. When we have type annotations, they serve as a hint as to which parameter to set.

Curried parameters

Curried parameters does not mean that we are including our parameters as ingredients in an Indian dish. **Currying** is a term borrowed from functional programming that is named after its inventor, the logician Haskell Curry. Currying involves transforming a function or method that takes multiple arguments in such a way that it can be called as a chain of functions or methods taking a single argument.

In practice with Groovy closures, this means we can "curry" a closure by pre-packing one or more of its parameters. This is best illustrated with an example.

```
def indian = { style, meat, rice ->
    return "${meat} ${style} with ${rice} rice."
}

def vindaloo = indian.curry("Vindaloo")
def korma = indian.curry("Korma")

assert vindaloo("Chicken", "Fried") ==
       "Chicken Vindaloo with Fried rice."
assert korma("Lamb", "Boiled") == "Lamb Korma with Boiled rice."
```

The `indian` closure above accepts three parameters. We can pre-pack its first parameter by calling the `curry` method of the closure. The `curry` method returns a new instance of the closure except with one or more of its parameters set. The variables `vindaloo` and `korma` contain instances of the `indian` closure with the first parameter `style` set. We refer to these as **curried closures**.

We can curry multiple parameters in one go. Parameters will always be curried in their order of declaration, so in this case `chickiTikka` will cause the `style` and `meat` parameters to be set.

```
def chickiTikka = indian.curry("Tikka", "Chicken")
assert chickiTikka("Boiled") == "Chicken Tikka with Boiled rice."
```

If we take a curried closure such as `korma` and curry it again, we now curry the subsequent parameters from the original `indian` closure. The `style` and `meat` parameters are now curried into the variable `lambKorma`.

```
def lambKorma = korma.curry("Lamb")
assert lambKorma("Fried") == "Lamb Korma with Fried rice."
```

We can continue currying parameters until we run out of parameters (or curry powder). At this point we have a curried closure `lambKormaBoiled` that can be invoked without passing any parameter.

```
def lambKormaBoiled = lambKorma.curry("Boiled")
assert lambKormaBoiled() == "Lamb Korma with Boiled rice."
```

Curried closures can be used very effectively in circumstances where contextual data needs to be gathered on the fly and then acted upon. We can write an appropriate closure that acts on all parameters as if they were available. We curry the parameters of the closure as we discover them, and eventually invoke the closure to act on the data. The only limitation that we have is that our closure parameters need to be defined in the correct order.

Closure return values

Closure declarations syntax provides no means of defining a return value. Every closure does, however, return a value with each invocation. A closure can have explicit `return` statements. If a `return` statement is encountered, then the value defined in the `return` statement is returned; otherwise execution continues until the last statement in the closure block.

```
def closure = { param ->
    if (param == 1)
        return 1
    2
}

assert closure(1) == 1 // return statement reached
assert closure(-1) == 2 // ending statement evaluates to 2
```

If no `return` statement is encountered, then the value returned by the closure is the result of evaluating the last statement encountered in the closure block. If the last statement has no value, the closure will return `null`.

```
void voidMethod() {
}

def nullReturn = { voidMethod() }
// voidMethod returns void so return is null
assert nullReturn() == null
```

Closure scope

Closures have access to variables in their surrounding scope. These can be local variables or parameters passed to a method inside which the closure is defined. Here, we can access the `name` parameter and the local variable `salutation` in our closure.

```
def greeting ( name ) {
    def salutation = "Hello"

    def greeter = { println salutation + ", " + name }

    greeter()
}

greeting("Dolly")
```

If the closure is defined within a class method, then the object instance fields are also available to the closure. The field member `separator` shown below is also accessible within the closure.

```
class Greeter {  
    def separator = ", "  
    def greeting ( name ) {  
        def salutation = "Hello"  
  
        def greeter = { println salutation + separator + name }  
  
        greeter()  
    }  
}  
  
Greeter greeter = new Greeter()  
greeter.greeting("Dolly")
```

In addition to directly accessing variables, we can also use `GStrings` to paste variables from the local scope into a string.

```
class Greeter {  
    def separator = ", "  
    def greeting ( name ) {  
        def salutation = "Hello"  
  
        def greeter = { println "${salutation}${separator}${name}" }  
  
        greeter()  
    }  
}  
  
Greeter greeter = new Greeter()  
greeter.greeting("Dolly")
```

In essence, the closure simply inherits all of the visible variables and fields from the surrounding scope in which it is defined. The closure can update any of these fields or variables, and the class or local method scope will see these changes. Likewise, any changes that occur in the class or method scope will also be seen by the closure.

Unlike regular method or class scope, we are able to pass a closure back from a method. At the time that a closure is defined, Groovy binds all of the variables that it accesses to the closure object. When this happens, Groovy converts any stack-based variables such as method parameters and local variables into heap-based duplicates of these objects. The values of these objects are now bound to the individual closure because the original values were lost once the method returned. Take the following example, which illustrates this:

```
class MyClass {  
    def member = "original "  
    def method (String param ) {  
        def local = member  
  
        return {  
            println "Member: " + member +  
                " Local: " + local +  
                " Parameter: " + param  
        }  
    }  
}  
  
MyClass myClazz = new MyClass()  
def clos1 = myClazz.method("First")  
clos1()  
myClazz.member = "modified "  
def clos2 = myClazz.method("Second")  
clos2()  
clos1()
```

Here we are examining the effect on scoped variables when we make multiple calls to a method that returns a closure. To keep things simple in this example, we are not modifying any variables within the closure. The first time that we call `method`, we return a closure which, when invoked, outputs the following.

```
Member: original  Local: original  Parameter: First
```

Both the field `member` and the `local` variable reflect the "original" state of the `member` field, because the `local` variable is just a copy of the field variable. The parameter variable reflects the parameter value that we just passed to `method`.

Before calling the method for the second time, we change the `member` variable to `modified`. When we invoke the closure that is returned from this call, we see:

```
Member: modified  Local: modified  Parameter: Second
```

The field `member` and `local` variable reflect the "modified" state of the `member` field, and the parameter value is what we just passed to the `method`. We then call the first closure for a second time, and see:

```
Member: modified  Local: original  Parameter: First
```

The `member` field shows the latest state of the field `member`, but the `local` variable and parameter are preserved in the same state as when the first call to `method` was made. If we were to have modified one or more local variables, then eventually all bets are off in terms of the state of these variables in relation to any particular closure instance. I recommend caution whenever returning closures from methods. Ensure that you fully understand the impact of the variables that you are acting upon and the state they will have when called is truly the state that you expect.

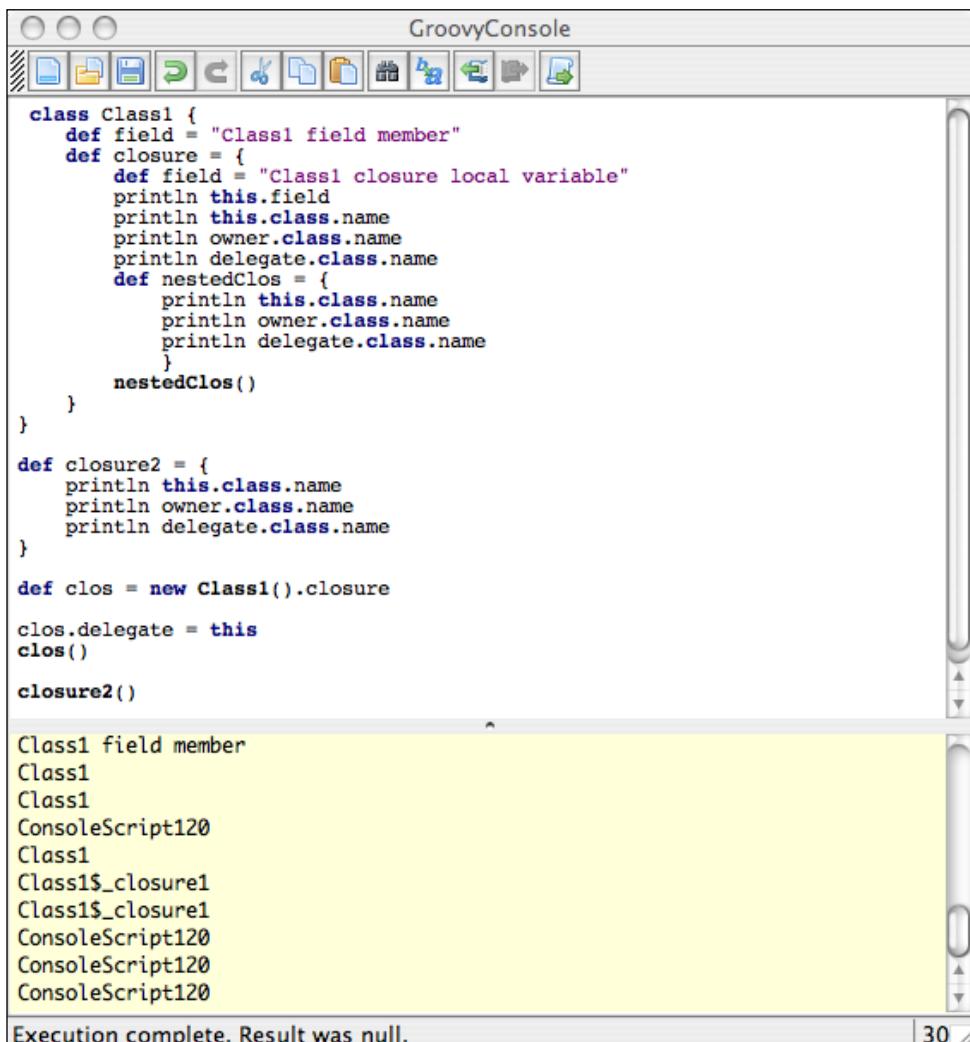
This note of caution also extends to accessing field members from a returned closure. While we can at least be sure that the state of a field member is the same whether it is accessed from the class or the closure, there is also the impact on the encapsulation of the class to be considered. When we pass a closure back from a class method, we are potentially giving insecure access to the inner workings of the class.

this, owner, and delegate

Groovy has three **implicit variables** in scope inside each closure. They are: `this`, `owner`, and `delegate`. The `this` variable refers to the enclosing class, provided that one exists. If the closure is defined within the scope of a script, the enclosing class is the script class. This will be auto-generated if we are running in the `GroovyConsole` or the `Groovy` shell.

The `owner` is the enclosing object of the closure. This is generally analogous to `this`, except in the case of nested closures where the enclosing object is another closure. The `delegate` is also usually the same as the `owner`, except that `delegate` can be changed.

In most closures, we only need to care about this as a means of accessing field variables in an outer scope. The owner and delegate variables will only become relevant later in the book when we deal with implementing our own builders. If we run the following script within the **GroovyConsole**, the console generates several Java classes under the covers. `ConsoleScript120` is the script class and `Class1$_closure1` is the closure class for closure.



The screenshot shows the GroovyConsole interface. The top bar has icons for file operations like Open, Save, and Run. The title bar says "GroovyConsole". The main area contains Groovy script code. The bottom pane shows the execution results.

```
class Class1 {  
    def field = "Class1 field member"  
    def closure = {  
        def field = "Class1 closure local variable"  
        println this.field  
        println this.class.name  
        println owner.class.name  
        println delegate.class.name  
        def nestedClos = {  
            println this.class.name  
            println owner.class.name  
            println delegate.class.name  
        }  
        nestedClos()  
    }  
}  
  
def closure2 = {  
    println this.class.name  
    println owner.class.name  
    println delegate.class.name  
}  
  
def clos = new Class1().closure  
  
clos.delegate = this  
clos()  
  
closure2()  
  
Class1 field member  
Class1  
Class1  
ConsoleScript120  
Class1  
Class1$_closure1  
Class1$_closure1  
ConsoleScript120  
ConsoleScript120  
ConsoleScript120
```

Execution complete. Result was null. | 30

Summary

In this chapter, we have covered closures in some depth. We have covered all of the important aspects of working with closures. We have explored the various ways to call a closure and the means of passing parameters. We have seen how we can pass closures as parameters to methods, and how this construct can allow us to appear to add mini DSL syntax to our code.

Closures are the real "power" feature of Groovy and they form the basis of most of the DSLs that we will develop later in the book. In the next chapter, we will build on this knowledge of closures and take a look at some more of the power features of the Groovy language, including Builders, and metaprogramming with the `ExpandoMetaClass` classes.

4

Example DSL: GeeTwitter

Before we dive any deeper into Groovy's more advanced features, let's take some time out to build a simple Groovy DSL, using some of the knowledge that we acquired in the previous chapters. In this chapter, we will use closures to build a simple and useful DSL that allows us to automate simple scripts that interact with Twitter.

We will take a stepwise approach to building our DSL. Starting with some vanilla Java APIs that require Groovy or Java programming skills, we will progressively apply some cool Groovy features to evolve a simple DSL that anybody can use.

Twitter

Twitter is the newest craze out there in the social networking and Web 2.0 world. Twitter has been variously described as a micro-blogging or social networking service. Twitter is a synergy between instant messaging, SMS, e-mail, and the Web, and allows users to make comments – "Tweets" – and have them instantly sent to multiple recipients – "followers".

Using Twitter is the essence of simplicity. Once you have set up an account, you can log onto the service and set a status message. Status messages are text messages of up to 140 characters in length. Twitter keeps a log of your status messages so that you or any other Twitter user can view them. If you follow another user or they follow you, then you will see their status messages in your updates page and they will see yours.

With Twitter, you can tweet from the Web or from your mobile phone. If you register your mobile phone with your account, you can send your tweets via SMS, and in certain countries you can get the tweets of the folks that you follow sent directly to your mobile phone as SMS messages.

To begin with, like most people, I was very skeptical about Twitter. My initial Twitter experiences mostly involved listening to friends' tweets about what they were having for breakfast. The experience was not unlike reading early bloggers blogging about the mundane aspects of their day. Since then Twitter has evolved, and many eminently sensible people are out there tweeting about stuff that really does have value.

Try searching Twitter for keywords such as "Groovy DSL" and you'll find tweets from folks like Guillame LaForge, who is a senior figure in the Groovy community. It's probably worth following the user **glaforge** on Twitter for that reason. In fact, it's safe to say that it is worth following anybody who is tweeting about "Groovy DSL".

Like most Web 2.0 applications, Twitter has a self-contained API that allows us to interface with it. By providing an open API to developers, Twitter has fostered a community of developers who have developed numerous different client applications. As a result, there are several third party and open source client applications for Twitter, including **gTwitter** for open source client Linux platforms and **Spaz**, a portable client built on Adobe AIR.

Suppose we were able to use Twitter APIs to develop a Groovy-based scripting interface, what would it look like? Imagine a script that allows us to follow every Twitter user who has recently tweeted about "Groovy DSL". Based on our knowledge of Groovy, something similar to the following would probably make sense:

```
"Groovy DSL".search { fromUser, message ->
    GTwitter.follow fromUser
}
```

For the rest of this chapter, we will walk through the steps it takes to turn this script into a working reality.

Working with the Twitter APIs

Twitter provides APIs that cover the whole gamut of operations that we might like to invoke for interacting with the service. Through the APIs, we can update our status, send direct messages to other users, list our friends and followers, and search for tweets by keyword, among many other useful features.

Twitter APIs are all pure HTTP-based requests. Any method that just retrieves data such as a search operation is implemented by using an HTTP GET request, whereas any method that updates, deletes, or creates an object is implemented by using an HTTP POST request. Most of the APIs conform to REST design principals and support XML and JSON data formats, and RSS and Atom syndication formats.

We can interact with the Twitter APIs by using any tool or programming language that allows us to interact with a web server using HTTP requests. Try out the following cURL command at the command line:

```
curl http://search.twitter.com/search.json?q=Groovy
```

This will invoke the search API, searching for tweets containing the keyword Groovy. The resulting list of tweets is formatted as JSON. cURL is a command line tool for fetching files using URL syntax. It's available by default on Mac OS X and most Linux distributions, or can be downloaded from <http://curl.haxx.se/download.html>. There is extensive documentation of Twitter APIs that can be found at <http://apiwiki.twitter.com>.

As an alternative to cURL, open the link <http://search.twitter.com/search.json?q=Groovy> in your browser. This will prompt you to download the file `search.json`, which contains your search results formatted as JSON. The JSON returned is not formatted in a user-readable manner, so it can be hard to read. A favorite site of mine is <http://www.javascriptbeautifier.com>. Pasting the `search.json` file into the **JavaScript** input box will let you produce nicely-formatted JSON, like this snippet from `search.json`:

```
{
  "results": [
    {
      "profile_image_url": "http://a3.twimg.com/profile_images/504587573/careto_normal.JPG",
      "created_at": "Wed, 09 Dec 2009 15:34:07 +0000",
      "from_user": "emedinam",
      "to_user_id": null,
      "text": "RT @glaforge: We've just released #Groovy 1.7-RC-2 live at #ggx http://bit.ly/58vfy :-)",
      "id": 6499346936,
      "from_user_id": 37077530,
      "geo": null,
      "iso_language_code": "en",
      "source": "<a href=\"http://www\(tweetdeck.com/" rel="uot;nofollow">TweetDeck</a>\""
    },
    ...
  ]}
```

The tweet above in JSON is the re-tweet of an announcement by Guillaume Laforge that 1.7 Release Candidate 2 was available. (announced live at the **Groovy&Grails eXchange Conference** in London(GGX).)

Reading raw JSON tweets is obviously not user friendly. Fortunately, thanks to the vibrant developer community that has sprung up around Twitter APIs, there are high-level client libraries for most languages, including Java, C++, PHP, and Ruby. There are three Java-based client libraries to be found – Twitter4J, jTwitter, and java-twitter. So far there is no Groovy library listed, but because Groovy is fully Java compatible, we can work with any of the Java libraries.

Using Twitter4J Java APIs

In the following examples, we will use Twitter4J by Yusuke Yamamoto. Twitter4J is an excellent open source, mavenized and Google App Engine-safe Java library for Twitter APIs, released under the BSD license. Using Twitter4J greatly simplifies the code that we need to write in order to build a simple DSL scripting interface for Twitter. Twitter4J can be downloaded from <http://yusuke.homeip.net/twitter4j/en/index.html>, and using it is as simple as adding the twitter4j JAR to your class path.

To begin with, let's try out the Twitter4J APIs and see what we can do with them. Although Twitter4J is a Java API, all of the examples here will run as Groovy scripts. If we were to write the same code as Java, we would need to build the examples into full-blown classes with the main methods before we could see any results. With Groovy, we can be up and running almost immediately.

 To run the scripts easily in groovyConsole, use the "Add Jar to classpath" menu option to add the Twitter4J JAR to the console class path. You can then run these examples unchanged in the console application.

Our examples dip into Groovy features at times, but we could simply cut and paste the sample code from the Twitter4J site and run it unchanged. Used in this way, Groovy can be a sandbox for exploring the Twitter4J APIs even without writing any Groovy code.

Tweeting

We'll start by trying out the APIs to get and set our current Twitter status. We will use the Twitter class from Twitter4J to log in to our Twitter account and access the APIs to update and get our status.

```
// Ensure twitter-4j-2.0.3.jar is in classpath
import twitter4j.*

def twitterId = "MyTwitterUserName"
def password = "MyTwitterUserPassword"
```

```
// Get a twitter connection
def twitter = new Twitter(twitterId,password)

// Update twitter status
twitter.updateStatus("Updating my status via the Twitter4J APIS")

// getUserDetail returns ExtendedUser which has the
// current status Tweet
println twitter.getUserDetail(twitterId).getStatusText()
```

Once we've got a connection to the service with the Twitter object, we can start to play with the APIs. The Twitter.updateStatus method sets a new status message (tweet) for us on our Twitter account. Try out this script with your own Twitter ID and password. Check your Twitter status through your favorite Twitter client or on the Web, and you will see it has been updated.

Retrieving our status is just as simple. We can use the Twitter.getUserDetail method to get an ExtendedUser object for any user including ourselves. Calling getStatusText on this object retrieves the current status message for any user.

Direct messages

Twitter has a direct message feature that allows you to send messages directly to another Twitter user. Direct messages don't show up in your general Twitter profile and are private to the sender and the recipient. We can use the APIs to send direct messages to a user and to check our own current messages from other users.

```
def twitter = new Twitter(twitterId,password)

// Send a direct messsage to twitter user GroovyDSL
twitter.sendDirectMessage(
    "GroovyDSL",
    "Hi Fergal read Groovy for DSL and loved it")

// Retrieve our latest direct messages
// same as visiting http://twitter.com/#inbox
messages = twitter.directMessages

messages.each { message ->
    println "Message from : ${message.senderScreenName}"
    println "      ${message.text}"
}
```

In the code snippet that we've just seen, we used the `Twitter.sendDirectMessage` method to send a message to the Twitter user `GroovyDSL`. We then use the `Twitter.directMessages` method to retrieve our latest direct messages from our inbox. This in fact is a shortcut to the `Twitter.getDirectMessages` API, which returns a list of `DirectMessage` objects. Groovy conveniently allows us to take shortcuts to any getter methods as a property access, even though the `Twitter` class is not in fact a POJO. We then list the sending user's screen name and the text of the message. We can also retrieve the time when the message was sent, and the user objects for the sender and recipient of the message.

Searching

The Twitter APIs have a powerful search capability that allows us to search for what people are commenting on at any given time. Twitter4J implements searching by passing a `Query` object to the `Twitter.search` API. Below, we search for tweets containing the keywords `Groovy` and `DSL`.

```
import twitter4j.*  
  
// For searching an anonymous connection is fine  
def twitter = new Twitter()  
// Create a query for tweets containing the terms "Groovy" and "DSL"  
def query = new Query("Groovy DSL")  
// Search and iterate the results  
twitter.search(query).tweets.each { tweet ->  
    println "${tweet.fromUser} : ${tweet.text}"  
}
```

Calling the `Twitter.search` API returns a `QueryResult` object. `QueryResult.getTweets()` will return a list of `tweet` objects. We can use the list of tweets matching our search criteria as a Groovy collection, and iterate it through the use of the built-in `each` method. At the time of writing this book, I got the following result from running this script:

```

import twitter4j.*

// For searching an anonymous connection is fine
def twitter = new Twitter()
// Create a query for tweets containing the terms "Groovy" and "DSL"
def query = new Query("Groovy DSL")
// Search and iterate the results
twitter.search(query).tweets.each { tweet -
    println "${tweet.fromUser} : ${tweet.text}"
}

glaforge : Funny, @venkat_s removed his shoes before starting his #groovy DSL
epragt : @venkat_s A shame Venkat, that you didn't use my Groovy Powerpoint DSL
mittie : 4: command-expression based DSL. #ggx (via @wmacgyver) #groovy
cgug : Thanks to @looselytyped for a great presentation tonight on "Creat
looselytyped : Just bumped into @ryanbriones in Chicago when walking back after
neodevelop : 50% off all MEAPs today only! with promo code dotd1207 at manning
groovytweets : RT @aalmiray: 50% off all MEAPs today only! with promo code dotd1207 at manning.co
grailsmx : 50% off all MEAPs today only! with promo code dotd1207 at manning.co
jconfino : RT @debasishg: RT @aalmiray: 50% off all MEAPs today only! with pro

```

Execution complete. | 11

Amid this cacophony of tweets, I see comments from **glaforge** the current Official Groovy Project manager. We might also see comments from Graeme Rocher in this list depending on when we run the script. Graeme is the creator of the Grails project. We might be interested in seeing what Graeme is saying about Grails right now. We can then modify our query to search for messages just from Graeme about Groovy and Grails as follows:

```

// Create a query for tweets containing the terms
// "Groovy" and "Grails"
Query query = new Query("from:graemerocher Groovy Grails")

```

Running the search script with this query would have resulted in an output that contained the following comment among the tweets:

```

graemerocher : RT @springsource: @glaforge , @graemerocher and @cdupuis
are presenting at this week's #Groovy and #Grails eXchange http://bit.
ly/60tsf ...

```

The @glaforge means graemerocher is referencing Guillame in his comments. It's always interesting to hear what these guys are talking about, especially when they reference each other. We can search for tweets from Guillame referencing Graeme by using the query string "from:glaforge @graemerocher", and so on.

I've just touched upon some of the search capabilities of Twitter APIs. The APIs allow you to use all of the same search operators that you can use in the Twitter **Search** box on the web application at <http://search.twitter.com/>. The full list of search operators that can be used are listed here: <http://search.twitter.com/operators>.

Following

The fundamental feature of Twitter is the concept of *following* and *followers*. You can tweet away on Twitter to your heart's content, but if no one is following your tweets you will not be heard. When you follow someone, your Twitter client will pick up their latest tweets and display them for you. If they follow you, then your tweets will be listed in their Twitter client when they view it. The Twitter web application has two lists: for those "following" and your "followers". Confused?

The Twitter4J APIs refer to the users you follow as *friends* and those following you as *followers*. This certainly clears up the confusion a little bit. However, users on Twitter don't necessarily know or seldom care about who is following them. So calling them as friends who eavesdrop on all their conversations is a little like stalking. Anyway we can use the APIs to list all of our friends or followers, or to create a new friendship (in other words follow another user).

```
import twitter4j.*  
  
twitterId = "MyTwitterUserName"  
password = "MyTwitterUserPassword"  
  
def twitter = new Twitter(twitterId,password)  
  
// Get a list of my followers  
friends = twitter.friends  
friends.each { friend ->  
    // Print each screen name  
    println friend.screenName  
}  
  
// "Follow" the Twitter user GroovyDSL  
twitter.createFriendship("groovydsl")
```

Here we use the `Twitter.getFriends` method to retrieve the list of users that we are currently following. In Groovy, we simply need to reference `twitter.friends`. Iterating this list, we can print the screen names of all of these users. Finally, we use the `Twitter.createFriendship` method to start stalking the user `GroovyDSL`. This is a user that I set up while writing this book and testing out these scripts. Feel free to run this script and start following me yourself. I just can't guarantee it won't be a Twitter Bot written in Groovy DSL that updates this user!

Groovy improvements

So far we have been using Twitter4J as a vanilla API, with a smattering of Groovy, so we have not been bringing any of the Groovier features to bear. Now that we know a little bit about the API, let's try to improve our usage by using some Groovy features. In this section, we are going to progressively improve our usage of the Twitter4J APIs by selectively using the features that Groovy provides. One of the most obvious features to use is **Closures**.

A Groovier way to find friends

In the previous examples, we iterated over our friends and printed out their details. What if we were to provide a method that takes a closure to apply to each friend or follower? In this example, we add these methods to a script, along with a follow method, to follow another Twitter user. We can use the `eachFollower` or `eachFriend` methods to list our current connections.

```
import twitter4j.*  
  
def twitterId = "MyTwitterUserName"  
def password = "MyTwitterPassword"  
  
// Get a twitter connection  
def twitter = new Twitter(twitterId,password)  
  
// Method to apply a closure to each friend  
static void eachFriend(Closure c) {  
    twitter.friends.each {  
        c.call(it.screenName)  
    }  
}  
  
// Method to apply a closure to each follower  
static void eachFollower(Closure c) {  
    twitter.followers.each {  
        c.call(it.screenName)  
    }  
}  
  
// Method to follow another twitter user  
void follow(twitter, user) {  
    twitter.createFriendship("${twitter.getUserDetail(user).getId()}")  
}
```

With these methods defined, we can start writing some powerful Groovy code to act on our friends and followers. How about printing the screen names of all the users that we are following:

```
println "I'm Following"
eachFriend (twitter) {
    println it
}
```

Or those who are following us:

```
println "Following me"
eachFollower (twitter) {
    println it
}
```

We can write a neat auto-follow script. In the following example, we use eachFollower to apply a closure to each of our followers. The closure method determines if we are already a friend of this follower simply by using the Groovy collections any, and follows the follower if we are not.

```
println "I'm Following"
eachFriend (twitter) {
    println it
}

// Auto follow
eachFollower (twitter) { follower ->
    // If any of my friends is this follower
    if (twitter.friends.any { friend ->
        friend.screenName == follower
    })
        return;
    // Otherwise follow him
    println "Following ${follower}"
    follow(twitter, follower)
}

println "Now I'm following!"
eachFriend (twitter) {
    println it
}
```

Twitter throws an exception if we try to follow a user that we are already following. In the `Auto follow` closure shown in the previous code snippet, we first checked to see if any of our friends is a follower before trying to follow him or her. If code brevity is important, we can of course rewrite the closure as follows:

```
// Auto follow
eachFollower (twitter) { follower ->
    try {
        follow(twitter, follower)
        println "Following ${follower}"
    } catch (e) { /* Ignore */ }
}
```

Groovy searching

In the same vein, we can also add a search method taking a closure.

```
import twitter4j.*

def twitterId = "MyTwitterUserName"
def password = "MyTwitterPassword"

// Get a twitter connection
def twitter = new Twitter(twitterId,password)

// Method to follow another twitter user
static void follow(user) {
    try {
        twitter.createFriendship("${twitter.
getUserDetail(user).getId()}")
    } catch (e) {}
}

static void search(terms, Closure c) {
    if (!twitter)
        twitter = new Twitter()
    def query = new Query(terms)
    twitter.search(query).tweets.each {
        c.call(it.fromUser,tweet.text)
    }
}
```

We can pass a closure to the `search` method in order to print out the details of the tweets that we find.

```
// Print all recent tweets about Groovy and DSL
search (twitter, "Groovy DSL") { from, tweet ->
    println from + " : " + tweet
}
```

Or we can use the `follow` method to follow the tweets of any user who posts about the search terms that we are interested in.

```
// Follow all users that have tweeted recently about Groovy and DSL
search (twitter, "Groovy DSL") { from, tweet ->
    try {
        follow(twitter, from)
        println "Following ${from}"
    } catch (e) { /* Ignore */ }
}
```

Adding all of these methods together is the first step towards writing a useful and simple DSL for Twitter, but it suffers from a number of problems, which we need to address.

Removing boilerplate

Any DSL that we develop with Groovy is referred to as an embedded DSL. In other words, it uses language features from the host language in order to build a new mini dialect that achieves a particular goal. As programmers, we can appreciate the elegance of how a closure can define a mini dialect that is embedded within our code. We are used to all of the boilerplate that goes with using a Java library.

By boilerplate we mean all of the setup code that is needed to establish the context in which our code is running. This could be connecting to a database, establishing a connection to a remote EJB object via a JNDI lookup, and so on. It also includes all of the other code, which is superfluous to the problem at hand but is imposed by the languages and environments that we use. The requirement in Java to write all of our code within a class is a case in point. Comparing the Groovy "Hello, World" program with its Java equivalents, we can see that all but a single line of the code is boilerplate, imposed by the language.

```
// Groovy
println "Hello, World!"

// Java
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Even the one useful line in the Java version is burdened with boilerplate. We have to explicitly write `System.out.println` to say that we are using the `System` class to print to the standard output. In Groovy, this is all just assumed. When we write Groovy, we embed mini DSLs within our code all the time, and surround it with these types of regular Groovy and Java-like structures. The fact that we are using mini dialect of Groovy is hidden because the cool stuff gets hidden among other not-so-cool boilerplate code.

What if we would like a non-Groovy or non-Java programmer to use our DSL? Ideally, we just want to document how to use the DSL features, and not the boilerplate that goes with it. Otherwise we would find ourselves saying to our users, "Trust me, just write this line of code and it will work... don't worry about what it does." Unfortunately, that's not what happens in practice. Boilerplate will always be a source of confusion and mistakes; so the less of it we have the better.

When we write something like the following in isolation of the boilerplate, we are actually getting towards a dialect that could be written by a non-Groovy programmer.

```
eachFriend (twitter) {  
    println it  
}
```

The goal now should be to remove as much of the boilerplate code as possible from our scripts.

Refactoring

The next steps we take with our DSL are refactorings to remove boilerplate. Our previous examples have all implemented methods locally within the script. This clearly needs to change, so our first step will be to refactor these methods into a standalone class. This class will become the main class for our DSL. We'll call the class `GeeTwitter` so as not to confuse it with the `gTwitter` client for Linux.

We need to consider how we would like our users to access the methods in our class. By default, the methods that we add to a class are instance methods and are only accessible through an instance of the class. If we define `login` and `search` methods in our class as shown below, the user of the DSL must first create a new instance of the `GeeTwitter` class before he can use them.

```
import twitter4j.*  
  
class GeeTwitter {  
    def twitter = null
```

```
void search(terms, Closure c) {
    if (!twitter)
        twitter = new Twitter()
    def query = new Query(terms)
    twitter.search(query).tweets.each {
        c.call(it.fromUser,it.text)
    }
}
```

We can write a script that uses this search method, as follows:

```
GeeTwitter gTwitter = new GeeTwitter()

gTwitter.search ("Groovy DSL") { from, tweet ->
    println "${from} : ${tweet}"
}
```

Although this is fine for most circumstances, we would like to make the ending DSL scripts as clear and to the point as possible, so that a non-programmer might be able to write them. The need to create a `GeeTwitter` object before we can use the method is more unnecessary boilerplate. If, instead, we make the method static, the usage of the method is much clearer to the average user.

```
import twitter4j.*

class GeeTwitter {
    static twitter = null

    static void search(terms, Closure c) {
        if (!twitter)
            twitter = new Twitter()
        def query = new Query(terms)
        twitter.search(query).tweets.each {
            c.call(it.fromUser,it.text)
        }
    }
}
```

To use this method, we can write the following script:

```
GeeTwitter.search ("Groovy DSL") { from, tweet ->
    println "${from} : ${tweet}"
}
```

By default, the classes generated by Groovy scripts are in the default package. In the previous examples, we don't define any package for the GeeTwitter class, so it also resides in the default package. When we run the search script, Groovy will automatically look for any class that we use and compile it, as long as it is in the same directory as the script that we launch with. For instance, if we launch the Groovy shell from a directory in which the GeeTwitter class resides, we can immediately start typing Groovy search DSL and see the results.

```
$ groovysh -cp /Path/to/twitter4j.jar
Groovy Shell (1.6.6 JVM: 1.5.0_16)
Type 'help' or '\h' for help.

-----
-----
-----

groovy:000> GeeTwitter.search("Packt Publishing") { from, tweet ->
groovy:001> println from + " : " + tweet
groovy:002> }

linobertrand : drupal New Book: Flash with Drupal | drupal.org: Packt
Publishing is pleased to announce the release of a .. http://tinyurl.com/
nhyzxxh

1BizAngel : New Book: Flash with Drupal | drupal.org: Packt Publishing
is pleased to announce the release of a new Drupal bo.. http://u.mavrev.
com/6su3

bigallan : reviewing a new chapter from the upcoming RichFaces book by
Packt Publishing

====> null
groovy:000>
```

Here we see how we can type Groovy search script directly into the Groovy shell and immediately see some tweets discussing upcoming books by Packt Publishing. Already we have something we could call a mini-Groovy DSL for Twitter searching. The Groovy shell is clearly not the best tool for our users to run this DSL in. So next we will look at some more improvements to allow us launch our DSL scripts from the command line. First, let's fully flesh out the GeeTwitter class with more methods.

Fleshing out GeeTwitter

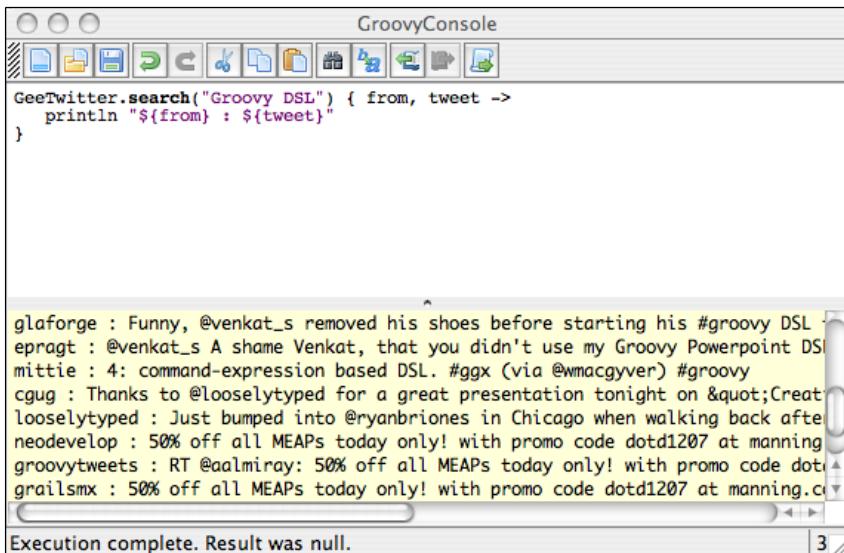
Searching with Twitter can be done with an anonymous connection to the service. If we want to add any methods that require an authenticated connection to Twitter, then we need to add a static instance of the Twitter class to GeeTwitter, and a login method to instantiate this object. Below is the fully fleshed out GeeTwitter class, with methods for logging into Twitter, along with following and searching methods.

```
import twitter4j.*  
  
class GeeTwitter {  
    static twitter = null  
  
    // establish connection with Twitter service  
    static login(id, password) {  
        twitter = new Twitter(id, password)  
    }  
  
    static sendMessage(id, message) {  
        // Send a direct messsage to twitter user GroovyDSL  
        twitter.sendDirectMessage(id, message)  
    }  
    // Method to apply a closure to each friend  
    static eachMessage(Closure c) {  
        twitter.directMessages.each {  
            c.call(it.senderScreenName, it.text)  
        }  
    }  
  
    // Method to apply a closure to each friend  
    static void eachFriend(Closure c) {  
        twitter.friends.each {  
            c.call(it.screenName)  
        }  
    }  
  
    // Method to apply a closure to each follower  
    static void eachFollower(Closure c) {  
        twitter.followers.each {  
            c.call(it.screenName)  
        }  
    }  
}
```

```
// Method to follow another twitter user
static void follow(user) {
    try {
        twitter.createFriendship(
            "${twitter.getUserDetail(user).getId()}")
    } catch (e) {}
}

static void search(terms, Closure c) {
    if (!twitter)
        twitter = new Twitter()
    def query = new Query(terms)
    twitter.search(query).tweets.each {
        c.call(it.fromUser,tweet.text)
    }
}
}
```

Now if we launch the **GroovyConsole** from the same directory as this class and add the Twitter4J jar from the class path, we can start experimenting with our fully-fledged Twitter DSL interactively. Here we can issue a search for the terms "Groovy DSL" and see the result directly within the console output pane.



If we need to use a method that requires authentication, we can call the GeeTwitter.login method first. In the following snippet, we see how the auto follow script has been reduced to one line. By removing all boilerplate and handling the follow method within our own DSL method, we can eliminate the need for the user to care about any exceptions that might be thrown. Auto follow becomes one elegant line of script.

```
GeeTwitter.eachFollower { GeeTwitter.follow it }
```

```
GroovyConsole
GeeTwitter.login("myId", "myPassword")
GeeTwitter.eachFriend { println it }
println "-- Now Follow your followers"
GeeTwitter.eachFollower { GeeTwitter.follow it }
GeeTwitter.eachFriend { println it }

groovy> GeeTwitter.eachFollower { GeeTwitter.follow it }
groovy> GeeTwitter.eachFriend { println it }

kodeblogger
GroovyDSL
-- Now Follow your followers
kodeblogger
fdearle
GroovyDSL

Execution complete. Result was null.
```

Improving search

When we started, we set the ambition of being able to write the following DSL code for searching Twitter:

```
"Groovy DSL".search { fromUser, message ->
    GTwitter.follow fromUser
}
```

This is not quite what we have achieved, but one small change is all we need to realize our ambition. To do so we need to digress slightly into another cool Groovy feature. We will deal with the whole subject of metaprogramming and the ExpandoMetaClass later in the book. For now, we just want to look at one aspect of metaprogramming – the ability to inject behavior into a class on the fly.

All objects that we use in Groovy implement the interface `GroovyObject`. Associated with every `GroovyObject` is a `MetaClass` object. The `MetaClass` defines the behavior of the `GroovyObject`, and one of its responsibilities is dispatching method invocations onto class methods. Class methods can be accessed through the `MetaClass` as properties, and with a little bit of Groovy magic we can add a method to the `MetaClass` by setting a named property with a `Closure` as its value.

All objects in the runtime, including Java objects, are wrapped as a `GroovyObject`, which means we can add a `search` method to `String` that does Twitter searching, as follows:

```
String.metaClass.search = { Closure c ->
    GeeTwitter.search(delegate, c)
}

String searchTerms = "Groovy DSL"
searchTerms.search { fromUser, message ->
    println fromUser + " : " + message
}
```

Here we assign a closure to a new property of the `String` class called `search`. This is just Groovy shorthand for calling `String.getMetaClass().setProperty(closure)`. When we refer to `delegate` in the `Closure` object, what we are in fact referring to is the original `String` object. In the previous example, `delegate` will be set to the `searchTerms` instance. The closure now acts as if it is an extra method of the `String` class, where `delegate` equates to this.

The effect of all this is to add a new method to the `String` class called `search`, which takes a `Closure` argument. This closure that we are assigning also takes a closure as its main argument. So when we write:

```
searchTerms.search { from, tweet ->
    ...
}
```

Groovy allows us to use methods and closures interchangeably. So what we are in fact doing is invoking the `call` method of `search`, and passing a closure to it. We could write this in a long-winded form as follows. The reference `.&someMethod` syntax used below allows us to assign the closure code of a method to a variable. The variables `searchClosure` and `searchMethod` wrap the same closure code.

```
Closure searchClosure = { Closure c ->
    GeeTwitter.search(delegate, c)
}

String.metaClass.search = searchClosure
```

```
String searchTerms = "Groovy DSL"

closure = { from, tweet ->
    println from + " : " + tweet
}

searchMethod = searchTerms.&search

searchMethod.call(closure)
```

However, when we define the search method, we end up with a new dynamic method added to the String class. Because Groovy has syntax that allows us to invoke a method directly on a literal value, we can cut our invocations down to:

```
"Groovy DSL".search { fromUser, message ->
    GTwitter.follow fromUser
}
```

The only challenge that now remains is how to instantiate this search method in such a way that the DSL user is unaware of it. Up to now we have been allowing the DSL script to be invoked directly. Running our DSL scripts in this way was fine when all we needed to make use of was the GeeTwitter class. GeeTwitter was automatically loaded when we ran the main script; so one possible solution is to add the method in a static initializer in the GeeTwitter class.

```
class GeeTwitter {
    static {
        String.metaClass.search = { Closure c ->
            search(delegate, c)
        }
    }
}
```

This is fine when the DSL class resides in the same directory as the script that we are running. This is no longer practical as we evolve our DSL. We will eventually want to pre-compile our DSL into a JAR file. There is no guarantee, then, that the GeeTwitter class will be loaded and the static initializer invoked before we make use of the search method.

Adding a command-line interface

One more step in making our DSL roadworthy is to add a command-line interface to it. In doing so, we move from invoking the DSL directly to allowing it to be loaded by a DSL command. This gives us more control over the environment in which the DSL will run, and allows us to take care of the housekeeping, such as adding the search method to the String class.

Groovy being Groovy, adding a command line is surprisingly easy:

```
#!/usr/bin/env groovy
String.metaClass.search = { Closure c ->
    GeeTwitter.search(delegate, c)
}

if (args)
    evaluate(new File(args[0]))
else
    println "Usage: GeeTwitter <script>"
```

The above shell script is all that we need in order to launch and run our GeeTwitter DSL. Being a shell script, we can run this directly on most Linux environments and Mac OS X, and on Windows if you have the **Gygwin** shell installed. We start by adding a search method to String. We then test to see if we have any arguments passed, and evaluate the first argument as the name of a file containing the DSL script.

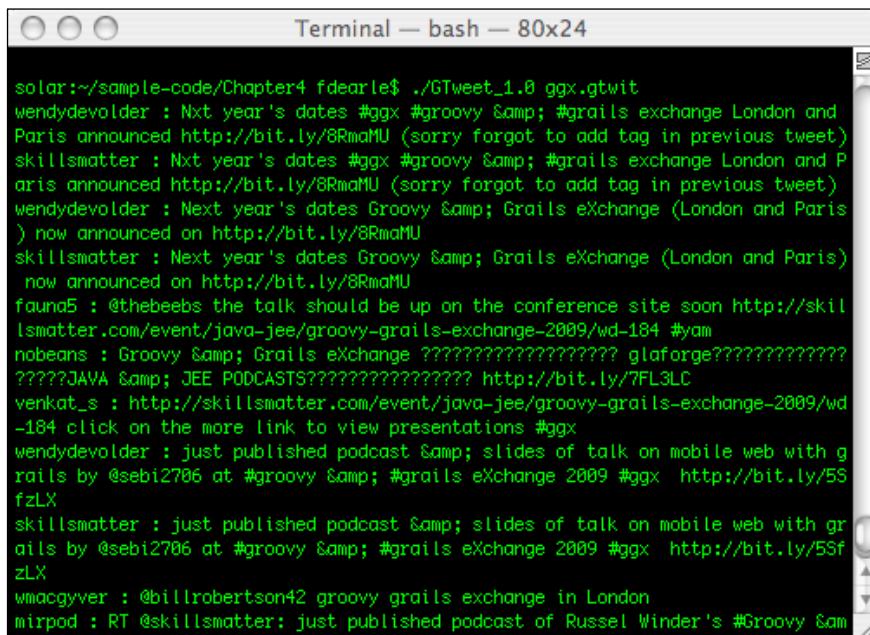
The evaluate method in Groovy allows us to pass a file containing Groovy code to the Groovy interpreter. Any code contained within the file is compiled and executed within the same virtual machine as the one where we are running the loading script. When the above script is executed from the command, the `groovy` command is invoked by the shell, which in turn launches a JVM. The `evaluate` method causes the target script to be loaded and executed in the same environment, and everything else works as if by magic.

Using the `evaluate` method means that any file name or extension can be used to contain the Twitter DSL scripts. Instead of naming all of our GeeTwitter scripts with the `.groovy` extension, we could decide that by convention our Twitter DSLs should have the extension `gtwit`.

By saving the command-line script as `GTweet_1.0` in a file accessible on the command path, and setting up our classpath to include the `twitter4j.jar`, we now have a bona fide Twitter DSL, implemented in Groovy, and that we can invoke through a shell command.

We can now try out our DSL from the command line. To run a script to search for tweets about the Groovy Grails Exchange event, we can save the following in a file called ggx.gtwit and invoke the GTweet_1.0 shell script:

```
"Groovy/Grails eXchange".search { from, tweet ->
    println from + " : " + tweet
}
```



A screenshot of a Mac OS X Terminal window titled "Terminal — bash — 80x24". The window shows the command "solar:~/sample-code/Chapter4 fdearie\$./GTweet_1.0 ggx.gtwit" followed by a series of tweets from various users (@wendydevalder, @skillsmatter, @found5, @nobean5, @venkat_s, @sebi2706, @wmacgyver, @mirpod) about the Groovy Grails Exchange event. The tweets include hashtags like #ggx, #groovy, and #grails, along with URLs and event details.

```
solar:~/sample-code/Chapter4 fdearie$ ./GTweet_1.0 ggx.gtwit
wendydevalder : Nxt year's dates #ggx #groovy & #grails exchange London and Paris announced http://bit.ly/8RmaMU (sorry forgot to add tag in previous tweet)
skillsmatter : Nxt year's dates #ggx #groovy & #grails exchange London and Paris announced http://bit.ly/8RmaMU (sorry forgot to add tag in previous tweet)
wendydevalder : Next year's dates Groovy & Grails eXchange (London and Paris) now announced on http://bit.ly/8RmaMU
skillsmatter : Next year's dates Groovy & Grails eXchange (London and Paris) now announced on http://bit.ly/8RmaMU
found5 : @thebeebes the talk should be up on the conference site soon http://skillsmatter.com/event/java-jee/groovy-grails-exchange-2009/wd-184 #yam
nobean5 : Groovy & Grails eXchange ?????????????????? glaforge?????????????
????JAVA & JEE PODCASTS????????????????? http://bit.ly/7FL3LC
venkat_s : http://skillsmatter.com/event/java-jee/groovy-grails-exchange-2009/wd-184 click on the more link to view presentations #gx
wendydevalder : just published podcast & slides of talk on mobile web with grails by @sebi2706 at #groovy & #grails eXchange 2009 #ggx http://bit.ly/5SfzLX
skillsmatter : just published podcast & slides of talk on mobile web with grails by @sebi2706 at #groovy & #grails eXchange 2009 #ggx http://bit.ly/5SfzLX
wmacgyver : @billrobertson42 groovy grails exchange in London
mirpod : RT @skillsmatter: just published podcast of Russel Winder's #Groovy & Sam
```

Adding built-in methods

However simple our DSL might now look, the need to preface our method calls with GeeTwitter is one final piece of boilerplate code that it would be nice to remove. Because we are evaluating the DSL script ourselves, rather than allowing Groovy to do it, we still have some scope to do this.

It would certainly be nicer to be able to write:

```
login myId, myPassword

eachFollower {
    sendMessage it, "Thanks for taking the time to follow me!"
}
```

which assumes that the `login` and `eachFollower` methods are built in to the DSL, rather than the more verbose:

```
GeeTwitter.login myId, myPassword
GeeTwitter.eachFollower {
    GeeTwitter.sendMessage (it,
        "Thanks for taking the time to follow me!")
}
```

Groovy provides two mechanisms that allow us achieve just this. Later in the book we will look at how we can manipulate the binding to achieve this. For this chapter, we will look at a more straightforward mechanism, which exploits the compilation model for Groovy scripts.

Whenever we run a Groovy script, the script gets compiled behind the scenes into a class derived from the `Groovy Script` class. Most of the time we are unaware of the fact that `Script` has just a few methods of interest to us. For instance, to access the binding of a script, we reference the `binding` property. When we do so, what we are in fact doing is calling the `Script.getBinding()` method. Similarly, when we used `evaluate` in the previous section to load and run our DSL from the launcher script, we were actually calling the `Script.evaluate()` method.

Instead of calling `Script.evaluate()` in the launch script, we can use the `GroovyScript` class to do the same thing. However, now we have the option of initializing the `GroovyScript` object that evaluates our DSL with a `CompilationConfiguration` object. This now gives us control over how the compilation of the script will be handled.

`CompilationConfiguration` gives us the ability to set a number of compilation attributes, including the `classpath` to be used and the `PrintWriter` object to be used as a standard output. It also provides a method `CompilationConfiguration.setScriptBaseClass()`, which allows us to provide an alternative subclass of `Script` to be used for the base class of our script instance.

```
abstract class MyBaseScript extends Script {
    def built-in ( a ) { println a }
}
```

If we provide the above MyBaseScript as the alternate Script class, any script that we evaluate by using this class will have the built-in method available by default. There is nothing magical here; it's now just a method of the `script` class that gets compiled into our environment. We can now rewrite our `GeeTwitter` class to make it a subclass of `Script`. Let's call it `GeeTwitterScript`, in order to distinguish it from the original.

```
import twitter4j.*

abstract class GeeTwitterScript extends Script {
    static twitter = null

    // establish connection with Twitter service
    def login(id, password) {
        twitter = new Twitter(id,password)
    }

    def sendMessage(id, message) {
        // Send a direct message to twitter user GroovyDSL
        twitter.sendDirectMessage(id, message)
    }
    // Method to apply a closure to each friend
    def eachMessage(Closure c) {
        twitter.directMessages.each {
            c.call(it.senderScreenName,it.text)
        }
    }

    // Method to apply a closure to each friend
    def eachFriend(Closure c) {
        twitter.friends.each {
            c.call(it.screenName)
        }
    }

    // Method to apply a closure to each follower
    def eachFollower(Closure c) {
        twitter.followers.each {
            c.call(it.screenName)
        }
    }

    // Method to follow another twitter user
    def follow(user) {
        try {
            twitter.createFriendship(
                "${twitter.getUserDetail(user).getId()}")
        }
    }
}
```

```
        } catch (e) {}
    }

def search(terms, Closure c) {
    if (!twitter)
        twitter = new Twitter()
    def query = new Query(terms)
    twitter.search(query).tweets.each {
        c.call(it.fromUser,it.text)
    }
}

def block(user) {
    try {
        twitter.createBlock(user)
    } catch (e) {}
}

static void search(terms, Closure c) {
    if (!twitter)
        twitter = new Twitter()
    def query = new Query(terms)
    twitter.search(query).tweets.each {
        c.call(it.fromUser,it.text)
    }
}
}
```

With this version, we supply the name of the new subclass to the `CompilerConfiguration` object in the launch script. The new launch script appears as follows:

```
#!/usr/bin/env groovy

import org.codehaus.groovy.control.*
String.metaClass.search = { Closure c ->
    GeeTwitterScript.search(delegate,c)
}
if(args) {
    def conf = new CompilerConfiguration()
    conf.setScriptBaseClass("GeeTwitterScript")
    def shell = new GroovyShell(conf)
    shell.evaluate (new File(args[0]))
}
else
    println "Usage: GeeTwitter <script>"
```

With this final version of our DSL, we have managed to distil the code down to the barest outline that is needed to express what we want to do. Using only the launcher shell script above, and the `GeeTwitterScript.groovy` class, we can start to automate our Twitter experience from the command line. Here are some of sample GeeTwitter scripts we can try out:

- Send a direct message to a user:

```
login "id", "password"  
sendMessage "GroovyDSL", "Using GeeTwitter to send you a message."
```

- Send a direct message to all of my followers:

```
login myId, myPassword  
eachFollower {  
    sendMessage it, "Thanks for taking the time to follow me!"  
}
```

- List all of the users that I'm following:

```
login "id", "password"  
eachFriend { println it }
```

- Follow all of my followers:

```
login myId, mypassword  
eachFollower {  
    follow it  
}
```

I think you'll agree that we've built quite an elegant DSL for Twitter, and we've been able to do it with surprisingly little code. This version has limited functionality to choose from, as I did not want to clutter the text with a fully-functional Twitter DSL. With a little bit more coding, we could extend this DSL to do a lot more. For example, if you've got a follower on Twitter who is following hundreds of users, but who has no followers, and who has issued only one tweet containing a link, then chances are that the follower is a spammer. Wouldn't it be nice to be able to block such a user automatically with a DSL script?

Summary

We have now built our first fully-fledged, albeit simple, Groovy DSL. We've seen how we can start with an existing Java-based API and evolve it into a simple user-friendly DSL that can be used by almost anybody. We've learned the importance of removing boilerplate code and how we can structure our DSL in such a way that the boilerplate is invisible to our DSL users.

The resulting DSL, being written in Groovy, is still an embedded DSL, but by sufficiently isolating the user scripts from its runtime and boilerplate, we have developed a DSL that could be documented in such a way that non-programming users could readily grasp how to use it. In the next chapter, we will extend our knowledge of the language further, by using some of Groovy's more advanced features such as builders and metaprogramming.

5

Power Groovy DSL features

In this chapter, we will cover some more advanced Groovy features. Coincidentally, these are also the features that, along with closures, allow us to extend and manipulate the language in order to create DSLs. We will cover a lot of ground in this chapter, including the following important features:

- Named parameters

To begin, we will look at this simple but effective feature, and see how Maps passed as parameters act as named parameters to a method.

- Builders

We will cover how to use Groovy builders to rapidly construct anything from web pages and XML to Swing UIs. While looking at Groovy builders, we will also introduce the native Groovy support for tree-based DOM structures, by looking at the built-in `GPath` operators in the Groovy language.

- `SwingBuilder`

We will add a quick and simple UI to our Twitter DSL, by using the `SwingBuilder` class.

- Method pointers

We will cover method pointers as a useful way to create aliases.

- Meta Object Protocol

We will cover the inner workings of Groovy's **Meta Object Protocol (MOP)**.

- How Builders work

Once we have covered the concepts behind the MOP, we will revisit Groovy Builders to understand how they are implemented using features from the MOP.

- `ExpandoMetaClass`

Finally, we will take a look at the `ExpandoMetaClass`, which is one of the most interesting Groovy classes as it provides the keys to dynamically change the behavior of any existing class, including Java classes, on the fly.

Named parameters

We have touched upon the concept of named parameters already. In a previous chapter, we looked at how Groovy allows us to construct a POGO by using a default built-in constructor that accepts a Map argument. We can construct a POGO by passing an inline Map object to the constructor. Groovy uses the map object to initialize each property of the POGO in turn. The map is iterated and the corresponding setter is invoked for each map element that is encountered.

```
class POGO {  
    def a = 0  
    def b = 0  
    def c = 0  
}  
  
def pogo1 = new POGO(a:1, b:2, c:3)  
  
assert pogo1.a == 1  
assert pogo1.b == 2  
assert pogo1.c == 3  
  
def pogo2 = new POGO( b:2, c:3)  
  
assert pogo2.a == 0  
assert pogo2.b == 2  
assert pogo2.c == 3  
  
def pogo3 = new POGO(b:2, a:3)  
  
assert pogo3.a == 3  
assert pogo3.b == 2  
assert pogo3.c == 0
```

When we pass a Map object to a constructor, the parentheses [] can be left out. We can also list the property values in any order we like. If a property is excluded, the corresponding setter will not be called, so its default value will be preserved.

Groovy also allows the same parameter-passing scheme to be used with method calls. If we invoke a method and pass a list of map elements in the same fashion as above, Groovy will collect the map elements into a Map object and pass this to the method as the first parameter. Parameters passed in this way are generally known as **named parameters**. The key that we use for each parameter provides a name for the parameter, which otherwise is anonymous.

```
def namedParamsMethod(params) {  
    assert params.a == 1
```

```
assert params.b == 2
assert params.c == 3
}

namedParamsMethod(a:1, b:2, c:3)
```

If the method has other parameters, Groovy allows the map entries to be placed before or after the other parameters. The map entries will still get collected and passed as the first argument.

```
def namedParamsMethod(params, param2, param3) {
    assert params.a == 1
    assert params.b == 2
    assert params.c == 3
    assert param2 == "param1"
    assert param3 == "param2"
}

namedParamsMethod("param1", "param2", a:1, b:2, c:3)
```

In fact, the map entries can be interspersed among the other parameters in any order we like. Groovy will collect the map entries and pass them as the first parameter. It will then scan the rest of the parameters from left to right and assign them to the subsequent parameters of the method. We can also drop the method call parentheses, which allows us to invoke the method call as follows:

```
namedParamsMethod a:1, "param1", b:2, "param2", c:3
```

These features combine neatly together for use in a DSL. Consider a method call to transfer funds from one account to another for a customer. The conventional way to layout parameters to a method is in the order of their importance from a programming logic point of view. So we declare the `customer` as the first parameter, as this is the primary object that we are operating on. We follow this with the accounts we are operating on, and finish up with the amount to transfer.

```
def transfer( customer, from_account, to_account, amount) {
    println """debiting ${amount} from ${from_account} account,
    crediting ${to_account} account for ${customer}"""
}

transfer("Joe Bloggs", "checking", "savings", 100.00)
```

Reading the method call does not provide an immediate clarity as to the function of all of the parameters. So we will only know for sure that `savings` is the receiving account by checking the method documentation to see that the third parameter is the receiving account. What if we make a small change to this method and have it accept named parameters instead?

```
def transfer( transaction, amount ) {  
    println """debiting ${amount} from ${transaction.from} account,  
    crediting ${transaction.to} for ${transaction.for}"""  
}  
  
transfer 100.00, from: "checking", to: "savings", for: "Joe Bloggs"  
transfer for: "Joe Bloggs", 200.00, from: "checking", to: "savings"
```

Now our method call even starts to look like English. We also have a good degree of flexibility in the order that we place the named parameters and where we place the `amount` parameter; so if we like we can turn the call into something that looks like English.

```
transfer 100.00, from: "checking", to: "savings", for: "Joe Bloggs"
```

Named parameters in DSLs

Being able to clarify exactly what a parameter means is a very useful technique to use in a DSL. Not only does it improve the readability of the DSL but it can also remove potential ambiguities. Looking back at our GeeTwitter DSL from the last chapter, we had a `sendMessage` call, which sends a text message to a Twitter user. Both the `message` parameter and the `user id` parameter were defined as strings, which of course could lead to ambiguity in the calling sequence.

```
def sendMessage(id, message) {  
    println "Sending (${message}) to ${id}"  
}  
  
// Correct  
sendMessage "GroovyDSL", "Hi from GeeTwitter"  
// Incorrect  
sendMessage "Hi from GeeTwitter", "GroovyDSL"
```

The second invocation here would of course cause an exception in the real GeeTwitter as we try to send a message to a user called "Hi from GeeTwitter" instead of to `GroovyDSL`. A small change removes this ambiguity and improves the readability of the DSL.

```
def sendMessage(id, message) {  
    println "Sending (${message}) to ${id.to}"  
}
```

```
// Correct
sendMessage to: "GroovyDSL", "Hi fromGeeTwitter"
// Correct - order is no longer important
sendMessage "Hi fromGeeTwitter", to: "GroovyDSL"
```

It might seem a little redundant or inefficient, from a programming point of view, to package a single value in a map. However, even though we are only going to pass the single value to a parameter along with a message parameter, naming this `to` parameter adds significantly to the resulting DSL script in terms of legibility.

Builders

Much of what we do in writing software involves construction or assembly of some sort or other. It could be building a graphical user interface, constructing a file to be saved on disk, or structuring a response to be sent to another system through a web services request. A lot of coding effort is dedicated to getting the structure of what we are building correct. Web pages need to be structured correctly in order to be displayed in a browser. XML-based files and responses to service requests need to be well-formed or they will cause validation exceptions. Building rich client UIs is an art in itself with each client framework – such as Swing or SWT – having its own arcane API to work with.

Beyond the complexities of the structures that we build, the pattern of construction and the order of initialization imposed by different APIs bring their own constraints. This alone will often obfuscate the structure of what we are building by burying it deep within boilerplate code. In addition to this, the classes of object that we need to construct may be of similar nature and have different means of construction.

Builder design pattern

It would be useful to have a means of constructing objects such that the method of construction was hidden. Enter the **builder** design pattern. The concept of a design pattern comes originally from the architectural profession in the late 1970s. In building architecture, a design pattern refers to the reuse of design solutions for similar problems. In office complexes, the collocating of stairwells, elevators, and bathrooms around central service columns is a typical design pattern.

Using such a design pattern, architects designing large office buildings can quickly lay out floor after floor of the building by repeating the layouts around the service columns on each floor. This leaves more time and effort to be expended on developing the functional work areas and aesthetics of the building. This not only benefits the architect but also benefits the user of the building.

No matter where we travel, whether it is to Bangkok, San Francisco, Paris, or London, it's usually not too difficult to find a bathroom presuming we can remember how to find our way back to the elevator that we came up in. When we do, we are benefiting from the application of a design pattern.

Design patterns are seldom invented. Instead, they are usually observed in existing buildings and catalogued. Good design evolves over time and is repeated again and again. By exploiting existing design patterns, the architect can rely on the experience of generations of previous building projects and be sure that at least these elements of the building will work as expected.

Design patterns began to be observed and catalogued in software engineering in the late eighties. Ward Cunningham and Kent Beck wrote one of the first conference papers on the subject at OOPSLA 1987. By 1994, the seminal work by Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides – *Design Patterns: Elements of Reusable Object-Oriented Software* – was published, which catalogued over 20 reusable design patterns, including the **Model View Controller (MVC)** and Factory and Builder patterns.

These authors were not claiming to have invented all of these patterns. For each of the patterns, they listed source systems where the patterns could be observed in use. Some came from ET++, an object-oriented framework developed by Gamma and others at Taligent. Many of the patterns, including MVC and Builder, originated from the Smalltalk language and framework.

We will focus our attention on the builder design pattern. The builder pattern originates from the `Parser`, `ClassBuilder`, and `ByteCodeStream` classes in Smalltalk. Gamma and the others describe its intent in their book as to "separate the construction of a complex object from its representation so that the same construction process can create different representations". The builder pattern has four components:

- **Builder**

This is an abstract interface, which specifies the methods that are needed to create the parts of a final product (file, UI, and so on). In the following figure, the `buildPart` method is listed, but a Builder could implement any set of methods that describe the production process of an object or a class of objects.

- **ConcreteBuilder**

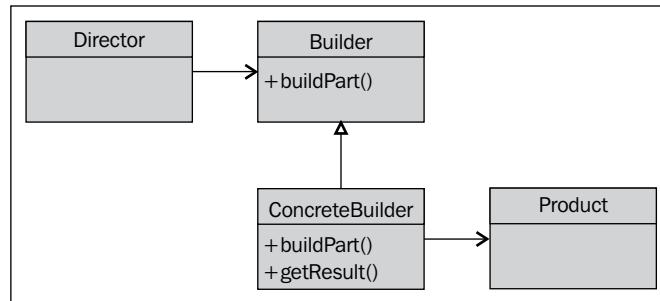
This is the concrete implementation of the builder interface, which is capable of building a specific product. The `ConcreteBuilder` implements the abstract builder methods and hides the complexities.

- Director

This is the code or class that constructs a particular product by using the builder interface methods.

- Product

This is the end result of the building process. It could be a file (XML, text, CSV, and so on), a UI (Swing, SWT), or markup (XML, HTML).



Using Groovy Builders

An important and powerful part of Groovy is its implementation of the Builder pattern. In the UML diagram of the Builder pattern above, we would expect that the Director component might be implemented through a `Director` class. The Groovy implementation goes beyond this by providing a mini DSL, which appears to embed the building process right into the language.

This style of markup is known as `GroovyMarkup`, and whose code looks more like a customized markup script than a regular Groovy script. This is due to the clever use of the **Groovy's Meta Object Protocol (MOP)** and closures.

At first glance, Groovy's builders defy our understanding of how things should work in an object-oriented language. We no longer seem to be creating objects and invoking methods as we would expect. For this section, let's suspend our disbelief and just enjoy the power that Groovy builders provide.

Here we will just try to understand how to use `GroovyMarkup` to build things. Later on in the chapter, we will cover how the MOP works, and it will become clear what tricks are being employed by the Groovy designers in order to give us a very neat way of constructing complex objects.

The best way to illustrate the GroovyMarkup feature is by way of an example. We'll start with something simple. Suppose that we need to export customer records from our e-commerce package in such a way that they can be used to initialize the customer database of a new CRM system that has been installed by us. The CRM system accepts customer records formatted in XML with customer IDs, names, and addresses. The XML required might look like this:

```
<customers>
    <customer id='1001'>
        <name firstName='Fred' surname='Flintstone' />
        <address street='1 Rock Road' city='Bedrock' />
    </customer>
    <customer id='1002'>
        <name firstName='Barney' surname='Rubble' />
        <address street='2 Rock Road' city='Bedrock' />
    </customer>
</customers>
```

Constructing this simple snippet of XML in Java requires numerous method calls to create XML elements and to set attributes of these elements. The nested structure of the document would need to be explicitly constructed by appending some elements as children of other elements. By the time that we are done coding, the procedural nature of the construction process means that the code doing the markup bears no resemblance to the end result XML.

MarkupBuilder

Consider the GroovyMarkup equivalent. The following is an actual self-contained script, which outputs the previous XML to the command line. Run this in the Groovy Console and you will see what it does.

```
def builder = new groovy.xml.MarkupBuilder()

def customers = builder.customers {
    customer(id:1001) {
        name(firstName:"Fred", surname:"Flintstone")
        address(street:"1 Rock Road", city:"Bedrock")
    }
    customer(id:1002) {
        name(firstName:"Barney", surname:"Rubble")
        address(street:"2 Rock Road", city:"Bedrock")
    }
}
```

The striking thing about the previous code snippet is that, unlike the Java code required to do the same, this snippet is remarkably similar in structure to the XML that is output. In this example, we are using the `MarkupBuilder` class from the `groovy.xml` package. `MarkupBuilder` is one of the several builder classes provided out of the box as part of the Groovy jars. `MarkupBuilder` can be used to effortlessly build XML- and HTML-formatted output. What we are in fact looking at is a series of nested closures, one within the other. The nesting of the closures exactly matches the tree-like structure of the desired XML output.

The same code can be modified to write the XML to a file, by constructing `MarkupBuilder` with a `FileWriter` object, as follows:

```
def builder = new groovy.xml.MarkupBuilder(
    new FileWriter("customers.xml"))

def customers = builder.customers {
    customer(id:1001) {
        name(firstName:"Fred", surname:"Flintstone")
        address(street:"1 Rock Road", city:"Bedrock")
    }
    customer(id:1002) {
        name(firstName:"Barney", surname:"Rubble")
        address(street:"2 Rock Road", city:"Bedrock")
    }
}
```

Namespaced XML

What if we would like to create namespaced XML? In GroovyMarkup, tags conform to the method call syntax. So how can we do that if `namespace:tag` is not a valid Groovy method name? Fortunately, there is a way around this. In order to insert the colon into a tag name, we simply surround the element name in quotes. Groovy allows us to invoke a method by using a string in place of the method name, so "`myMethod()`" is treated the same as `myMethod()`.

```
def xml = new groovy.xml.MarkupBuilder()

def params = [:]

params."xmlns:bk" = "urn:loc:gov:books"
params."xmlns:isbn" = "urn:ISBN:0-393-36341-6"

def bk_tag = "bk:book"
xml."bk:book"(params) {
```

```
"bk:title"("Cheaper by the Dozen")
"isbn:number"(1568491379)
}
```

Here we are using the strings' references to set the `xmlns` namespaces for `bk` and `isbn`. Then we use strings to declare the element names in our markup. All of this results in the following output:

```
<bk:book xmlns:bk='urn:loc.gov:books' xmlns:isbn='urn:ISBN:0-393-
36341-6'>
    <bk:title>Cheaper by the Dozen</bk:title>
    <isbn:number>1568491379</isbn:number>
</bk:book>
```

This technique is not limited to namespaces. We can use it anywhere that we need to output a character in a tag name, which would otherwise not be valid as a Groovy method name (for instance, hyphenated element names). Any Groovy string can be used as an element name, so the following is also valid, where we use `${book_title}` to paste the tag name into the markup from a local variable:

```
def xml = new groovy.xml.MarkupBuilder()

def book = "bk-book"
def book_title = "bk-title"

xml."${book}" {
    "${book_title}"("Cheaper by the Dozen")
    "isbn-number"(1568491379)
}
```

The `MarkupBuilder` class will slavishly emit whatever we ask it to. In the previous code snippet, we are creating namespaces by using standard markup with the `MarkupBuilder` class. A more elegant way of creating namespaced XML is by using the `StreamingMarkupBuilder` class, which has built-in support for namespaces.

`StreamingMarkupBuilder` decouples the output of the markup from the creation of the markup closure. We then bind the closure to the `StreamingMarkupBuilder` at the time at which we want the output to take place.

```
def xml = new groovy.xml.StreamingMarkupBuilder()

def markup = {
    customers {
        customer(id:1001) {
            name(firstName:"Fred",surname:"Flintstone")
            address(street:"1 Rock Road",city:"Bedrock")
```

```
        }
        customer(id:1002) {
            name(firstName:"Barney", surname:"Rubble")
            address(street:"2 Rock Road", city:"Bedrock")
        }
    }
}

println xml.bind( markup )
```

Within the closure, we can reference a variable, `mkp`, which allows us to give instructions to the builder in order to control XML generation. Two handy methods we can invoke are `xmlDeclaration()`, which causes the XML declaration header to be output, and `declareNamespace()`, which sets up a namespace.

```
def xml = new groovy.xml.StreamingMarkupBuilder().bind
{
    mkp.xmlDeclaration()
    mkp.declareNamespace('bk':'urn:loc:gov:books')
    mkp.declareNamespace('isbn':'urn:ISBN:0-393-36341-6')

    println mkp.class

    bk.book {
        bk.title("Cheaper by the Dozen")
        isbn.number(1568491379)
    }
}

println xml
```

Once we have made the builder aware of our namespaces, we can utilize them in the markup code by using suffix notation. So `namespace.tag` will be output in the XML as `namespace:tag`, as follows:

```
<?xml version="1.0"?>
<bk:book xmlns:bk='urn:loc:gov:books' xmlns:isbn='urn:ISBN:0-393-
36341-6'>
    <bk:title>Cheaper by the Dozen</bk:title>
    <isbn:number>1568491379</isbn:number>
</bk:book>
```

GroovyMarkup syntax

GroovyMarkup is nothing more than method call syntax combined with closures and named parameters. But in effect these Groovy syntactical features are combined to produce a new DSL syntax for GroovyMarkup with its own rules. Let's look in detail at the syntax from the previous examples.

```
def customers = builder.customers {  
    ...
```

To begin with, we define the root node of our XML by invoking `MarkupBuilder.customers()` on the `builder` object. This causes a root `customers` tag to be output into the XML stream and the code. The tag is not closed off until the following closure is executed. This looks and behaves like a `customer` method taking a closure as a parameter even though there is no such method.

Nested within the closure we come across more methods, such as calls to `customer`, `title`, `name`, and `address`.

```
customer(id:1001) {  
    ...
```

This method call will cause a new nested `customer` tag to be output into the XML stream with an `id` attribute set to `1001`. Once again, the tag is not closed off until the closure is executed, during which more method-like calls are encountered.

```
title("Mr")  
name(firstName:"Fred", surname:"Flintstone")  
address(street:"1 Rock Road", city:"Bedrock")
```

No methods exist for `customers`, `customer`, `title`, `name`, or `address`. The `MarkupBuilder`, in conjunction with its base `BuilderSupport` class, uses the Groovy MOP to make all of this work as if by magic. The beauty of this approach is how intuitive the resulting code is, because it closely reflects the resulting markup. All we need to remember is that pseudo method call syntax will create a tag, and named parameters will be inserted as attributes in the resulting output.

The parameters passed follow the same conventions that we discussed earlier in relation to named parameters. In this case, all named parameters are collected and become the attributes of the element. We should only pass one additional parameter, which is used as the body of the tag/element.

GroovyMarkup and the builder design pattern

We can try to understand GroovyMarkup in the context of the builder pattern described earlier.

- Builder

`MarkupBuilder` is derived from the `BuilderSupport` class, which is the `Builder` component of the pattern.

- `ConcreteBuilder`

In this case, `MarkupBuilder` class is the `ConcreteBuilder`, which is designed to handle all XML-style nested tag formats delineated by angled brackets.

- Director

The director is the `GroovyMarkup` code itself, as illustrated earlier.

- Product

The product is the XML streamed directly to standard output or to the `Writer` we specify when constructing the `MarkupBuilder` object.

With `MarkupBuilder`, it's also just as easy to build HTML pages. Here we generate a simple HTML page:

```
def html = new groovy.xml.MarkupBuilder()

html.html {
    head {
        title "Groovy Builders"
    }
    body {
        h1 "Groovy Builders are cool!"
    }
}
```

In the next example, we build a more complex HTML page containing **nested tables**. `MarkupBuilder` will close all tags correctly so that they are well formed. A classic mistake when working with nested tag formats is to misplace or unbalance the closing of tags. The HTML `<table>` tag and its nested `<tr>` and `<td>` tags are highly prone to error when hand-coded. Assume that we want to generate HTML to display the names of the various `Groovy Builder` and `ConcreteBuilder` classes in a nested table.

Builder class	Concrete Class
<code>groovy.util.BuilderSupport</code>	<code>groovy.util.AntBuilder</code> <code>groovy.xml.MarkupBuilder</code>
<code>groovy.util.FactoryBuilderSupport</code>	<code>groovy.util.NodeBuilder</code> <code>groovy.swing.SwingBuilder</code>

The HTML to produce this table would be something like the following:

```
<html>
<body>
<table border="1">
<tr>
    <th>Builder class</th>
    <th>Concrete Class</th>
</tr>
<tr>
    <td>groovy.util.BuilderSupport</td>
    <td>
        <table>
            <tr><td>groovy.util.AntBuilder</td></tr>
            <tr><td>groovy.xml.MarkupBuilder</td></tr>
        </table>
    </td>
</tr>
<tr>
    <td>groovy.util.FactoryBuilderSupport</td>
    <td>
        <table>
            <tr><td>groovy.util.NodeBuilder</td></tr>
            <tr><td>groovy.swing.SwingBuilder</td></tr>
        </table>
    </td>
</tr>
</table>
</body>
</html>
```

Although the HTML above looks correct, one of the TABLE tags is incorrectly terminated as </td>. Displaying the above code in a browser would show an extra layer of nesting in the TABLE that was not intended. The same HTML can be generated using MarkupBuilder as shown in the following listing:

```
def html = new groovy.xml.MarkupBuilder()
html.html {
    head {
        title "Groovy Builders"
    }
    body {
        table(border:1) {
            tr {
                th "Builder class"
                th "Concrete class"
            }
            tr {
                td "groovy.util.BuilderSupport"
                td {
                    table {
                        tr {
                            td "groovy.util.AntBuilder"
                        }
                        tr {
                            td "groovy.xml.MarkupBuilder"
                        }
                    }
                }
            }
            tr {
                td "groovy.util.FactoryBuilderSupport"
                td {
                    table {
                        tr {
                            td "groovy.util.NodeBuilder"
                        }
                        tr {
                            td "groovy.swing.SwingBuilder"
                        }
                    }
                }
            }
        }
    }
}
```

In the Groovy version, all of the tags that are produced are guaranteed to be correct with respect to nesting and balancing. The Groovy version also looks much less cluttered and readable. In fact, it is impossible for us to make the same type of errors with the Groovy version, as the compiler will insist that all parentheses are properly balanced.

Using program logic with builders

So far we have just used Groovy builders as straightforward markup scripts. In spite of the unusual syntax, GroovyMarkup scripts are still just plain Groovy scripts; so there is nothing stopping us from mixing the construction process with regular program logic if we please. Here we iterate over a list of customer data while generating XML from the customer records that we find.

```
def builder = new groovy.xml.MarkupBuilder()

class Customer {
    int id
    String firstName
    String surname
    String street
    String city
}
def fred = new Customer(id:1001,firstName:"Fred",
surname:"Flintstone",
street:"1 Rock Road",city:"Bedrock")
def barney = new Customer(id:1002,firstName:"Barney",
surname:"Rubble",
street:"2 Rock Road",city:"Bedrock")
def customerList = [ fred, barney]

def customers = builder.customers {
    for (cust in customerList) {
        customer(id:cust.id) {
            name(firstName:cust.firstName,surname:cust.surname)
            address(street:cust.street, city:cust.city)
        }
    }
}
```

Builders for every occasion

Out of the box, the Groovy libraries include a suite of builders for most of the common construction tasks that we might encounter. Here is a list of some of them:

- `MarkupBuilder`

This we have already seen. It can be used to generate any XML-style tagged output. Class: `groovy.xml.MarkupBuilder`

- `NodeBuilder`

This is a useful builder for building tree-like structures in memory `Node` instances in memory. Class: `groovy.util.NodeBuilder`

- `DOMBuilder`

This builder will construct a WC3 DOM tree in memory from the `GroovyMarkup` that we provide. Class: `groovy.xml.DOMBuilder`

- `SAXBuilder`

This is very similar to the `DOMBuilder` in so far as the end result is a WC3 DOM in memory. The difference is that it works with an existing SAX `ContentHandler` class and fires SAX events to it as the `GroovyMarkup` is executed. Class: `groovy.xml.SAXBuilder`

- `AntBuilder`

`AntBuilder` is a little bit of a conundrum in that it is not strictly speaking a builder in the "builder design pattern" sense. The `GroovyMarkup` closely matches the equivalent Ant XML but what actually happens is that the Ant tasks are fired directly by the script rather than producing an Ant XML script. Class: `groovy.util.AntBuilder`

- `JMXBuilder`

Also in the same vein as `AntBuilder` is the `JMXBuilder` class, which can be used to deploy JMX management beans by using simple markup-style syntax. `JMXBuilder` is a Groovy-based DSL for declaratively exposing services, POJOs, POGOs, and so on, via the **Java Management Extensions (JMX)**.

Class: `groovy.jmx.builder.JMXBuilder`

- `SwingBuilder`

Next we'll cover `SwingBuilder` in detail with an example. This builder constructs Swing-based UIs. Class: `groovy.swing.SwingBuilder`

NodeBuilder

NodeBuilder is used to build tree structures of Node instances in memory. We use exactly the same GroovyMarkup syntax as before. Here we build up a tree structure in memory from customer data, using the same structure as with MarkupBuilder. All that needs to change to construct a node-based tree in memory is to replace the builder instance created with an instance of NodeBuilder. Once the markup code has been executed, the customers field contains the tree structure, which can be accessed by using Groovy's XPath-like syntax, GPath.

```
def builder = new groovy.util.NodeBuilder()

class Customer {
    int id
    String firstName
    String surname
    String street
    String city
}
def fred = new Customer(id:1001,firstName:"Fred",
surname:"Flintstone",
street:"1 Rock Road",city:"Bedrock")
def barney = new Customer(id:1002,firstName:"Barney",
surname:"Rubble",
street:"2 Rock Road",city:"Bedrock")
def wilma = new Customer(id:1003,firstName:"Wilma",
surname:"Flintstone",
street:"1 Rock Road",city:"Bedrock")
def betty = new Customer(id:1004,firstName:"Betty", surname:"Rubble",
street:"2 Rock Road",city:"Bedrock")
def customerList = [ fred, barney,wilma,betty]

def customers = builder.customers {
    for (cust in customerList) {
        customer(id:cust.id) {
            name(firstName:cust.firstName,surname:cust.surname)
            address(street:cust.street, city:cust.city)
        }
    }
}

assert customers.customer[0].@id == 1001
assert customers.customer[1].@id == 1002
assert customers.customer[0].address[0].@street ==
customers.customer[2].address[0].@street'
```

```
assert customers.grep{
    it.name.any{it.'@surname' == "Rubble"}
}.size == 2
assert customers.grep{
    it.name.any{it.'@surname' == "Rubble"}
}.address.every{ it.'@street'[0] == "2 Rock Road"}
```

Using GPath to navigate a node tree

We've used GPath in the preceding code to access the node structure created from our markup. To make sense of how the GPath syntax works, we must visualize it as a tree structure where `customers` is the root node. Node attributes are accessible as map entries, so `element.'@attribute'` is used to access the attribute values.

```
assert customers.customer[0].'@id' == 1001
assert customers.customer[1].'@id' == 1002
```

Below the root `customers` node, each node can have 1 to n leaf nodes, so `customer` is always returned as a list object even if only one item is contained in the list. We access individual elements by using array syntax (`customer[1]`) but any list method can be used. The snippet below will list out the first names of all `customers` in the tree.

```
customers.customer.each {
    println it.name[0].'@firstName'
}
```

As we index deeper into the tree, we still need to use array syntax to access the lower nodes, even if the elements at these levels are singletons. Here we assert that Fred and Wilma live at the same address.

```
assert customers.customer[0].address[0].'@street'
== customers.customer[1].address[0].'@street'
```

Finally, we can use a more complex GPath query to assert that all the Rubbles live at "2 Rock Road". This is quite a complex query, so we will decompose it as shown below. First, we use `grep` on the root `customers` node to produce a tree of all `customers` whose surname is `Rubble`. This tree should have two nodes: one for `Barney` and one for `Betty`.

```
def rubbles = customers.grep{ it.name.any{it.'@surname' == "Rubble"}}
```

Now we can assert that every Rubble lives at "2 Rock Road":

```
assert rubbles.address.every{ it.'@street'[0] == "2 Rock Road"}
```

SwingBuilder

Most Java developers I know hate Swing UIs with a passion. This is a pity because Swing is still to this day the best way to build a native UI application that will genuinely work on any operating system. The reason why people hate Swing is because of the APIs. Let's face it, Swing UIs are a chore to build and maintain, due to the unwieldy nature of the Swing APIs.

Any Swing app I've ever worked on has been a mess of component initialization code, intermingled with anonymous inner classes for event handling. Each Swing component, however small or insignificant, has to be newed and given a name. Figuring out how all of the components nest together, when some such as button groups and panels may not even be visible, is an endless chore.

We have all worked with apps that started out with a neat and tidy code layout, but generations of code changes later ended up as Frankenstein monsters. A neat trick with apps like this that I learnt years ago is the **border hack**. This involves placing a different colored border around each important top-level component in your layout, and bingo! You can start to make sense of the structure of your UI and debug layouts.

So for me `SwingBuilder` is the single most joyful Groovy feature to use – a much needed markup language for Swing UIs. For years I've been doing my quick and nasty UI prototyping with Swing and the **Napkin Look & Feel**. By the time you've hacked together a UI and given a demo, you just want to throw it away and start coding again from scratch. But not with `SwingBuilder`!

[ The Napkin Look & Feel (<http://napkinlaf.sourceforge.net/>) is a Swing Look & Feel designed with one purpose in mind – to give the impression of something temporary so that your pointy-haired boss does not think the demo that you just gave him is of production-ready code that should be ready to ship.]

Below is a UI built with `SwingBuilder` that puts a simple UI onto the GeeTwitter searching DSL from the last chapter. You can see in the coming screenshot how the markup mirrors the actual layout in the UI. Closures are used in place of anonymous inner classes for events such as `actionPerformed` on the **Exit** menu. This took less than five minutes to throw together, and unlike a pure Swing API version, it is production-ready as soon as we remove the Napkin Look & Feel line from the code.

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*
import net.sourceforge.napkinlaf.*

data = []
```

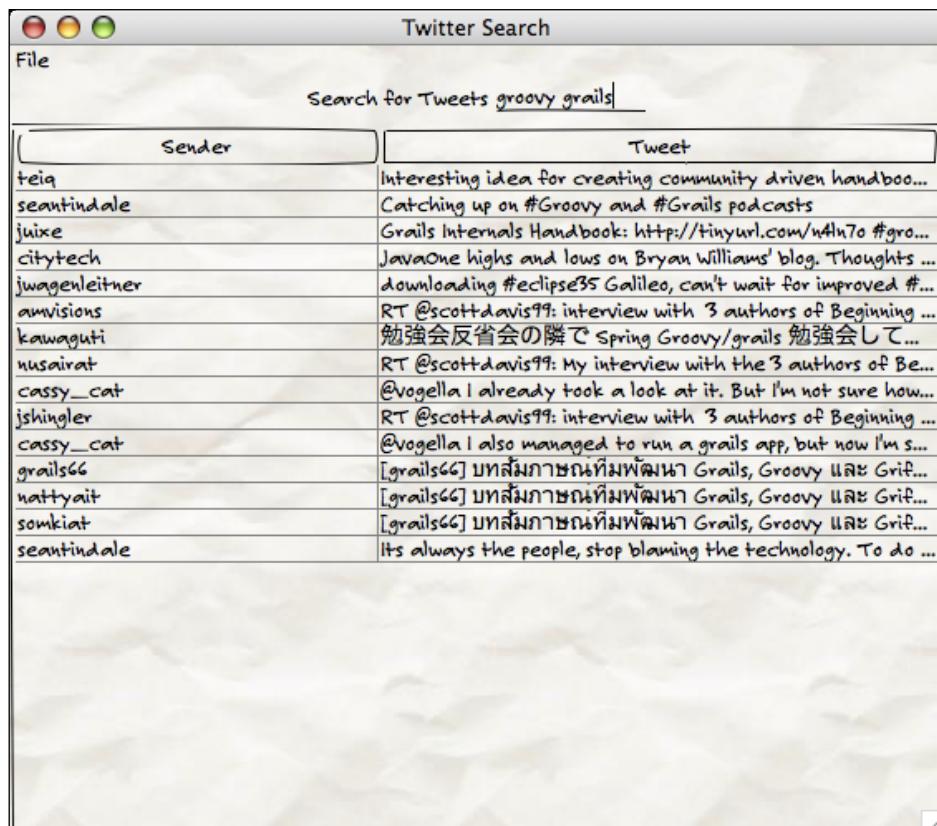
```
def results
swing = new SwingBuilder()
swing.lookAndFeel(new NapkinLookAndFeel())
frame = swing.frame(title:'Twitter Search') {
    menuBar {
        menu('File') {
            menuItem 'Exit', actionPerformed: { System.exit(0) }
        }
    }
    panel(layout: new BorderLayout()) {
        panel (constraints:BorderLayout.NORTH) {
            label 'Search for Tweets'
            textField(columns:10, actionPerformed: { event ->
                data = GeeTwitter.search(event.source.text)
                results.model = tableModel(list:data) {
                    propertyColumn(header:'Sender',
                        propertyName:'from',preferredWidth:20)
                    propertyColumn(header:'Tweet',
                        propertyName:'tweet',preferredWidth:140)
                }
            })
        }
        scrollPane (constraints:BorderLayout.SOUTH) {
            results = table() {
                tableModel(list:[]) {
                    propertyColumn(header:'Sender',
                        propertyName:'from',preferredWidth:20)
                    propertyColumn(header:'Tweet',
                        propertyName:'tweet',preferredWidth:140)
                }
            }
        }
    }
    frame.pack()
    frame.show()
```

The GroovyMarkup that we use for `SwingBuilder` is pretty much identical to what we've seen before, with a few differences. Unlike `MarkupBuilder` and `NodeBuilder`, we can't simply invent tags to insert into the GroovyMarkup, as this would not make sense. The tags must correspond to real UI widgets or controls that can be placed into the UI. Above, we use `frame`, `menuBar`, `panel`, and `textField`, among others. The full list of widgets can be found at <http://groovy.codehaus.org/Alphabetic+Widgets+List>. There are other non-widget tags, such as `tableModel`, that must be used in conjunction with a `table` tag and others.

In the above example, we start with a `frame` tag. The `SwingBuilder` class takes care of creating a `JFrame` widget for this frame, maintaining it. Any further widgets declared in the nested closure below this frame will be added to the frame. Take the preceding `scrollPane`, for example. Widgets that are nested below this will be added to the `scrollPane`, and so on. The nesting of the closure code that we use to declare the components dovetails exactly with how these components are nested in the UI. Declaring a widget returns the corresponding Swing widget; so the `frame` variable above contains a `JFrame` instance that allows us to call the regular swing `pack()` and `show()` methods to display the UI.

`SwingBuilder` handles the Swing event loop and dispatches any event that occurs. All we need to do is supply a closure for the `actionPerformed` attribute of any widget that we want to provide event handling for. This is far neater than the anonymous classes that regular Swing code is usually littered with.

The following result is a quick and nasty UI for Twitter searching:



Griffon: Builders as DSL

If you like the look of `SwingBuilder`, then take a look at the Griffon project. Griffon is a Grails-like application framework for desktop application development. Griffon provides an MVC paradigm where the view portion of the application can be coded in the Builder of your choice.

Griffon adds the framework and runtime support around the Builder code so that only the markup needs to be written to create a View. Here is the View code for a simple Griffon application:

```
application(title:'DemoConsole', pack:true, locationByPlatform:true) {  
    panel(border:emptyBorder(6)) {  
        borderLayout()  
  
        scrollPane(constraints:CENTER) {  
            textArea(text:bind(target:model,  
                targetProperty:'scriptSource'),  
                enabled: bind {model.enabled},  
                columns:40, rows:10)  
        }  
  
        hbox(constraints:SOUTH) {  
            button("Execute",  
                actionPerformed:controller.&executeScript,  
                enabled: bind {model.enabled})  
            hstrut(5)  
            label("Result:")  
            hstrut(5)  
            label(text:bind {model.scriptResult})  
        }  
    }  
}
```

The set-up code for the Builder objects is not necessary as Griffon takes care of this. Used in this context, Builder markup code is now more like a declarative DSL for the user interface. Later on in the book, we will take a look at how another Groovy project **Gant** uses `AntBuilder` in a similar fashion, to make a declarative Groovy DSL replacement for Ant scripts.

Method pointers

Groovy allows you to assign a method to a closure by using the `&` syntax. The closure returned is often referred to as a **method pointer**. Method pointers can be assigned by de-referencing the method name from any object instance, for example:

```
def list = ["A", "B", "C"]

def addit = list.&add

addit "D"

assert list == ["A", "B", "C", "D"]
```

The difficulty with method pointers to instance methods is being sure what instance the method pointer is referencing. In essence, an instance method pointer violates the encapsulation rules for the object by passing control to an object that is outside the direct control of a class. So I recommend caution when using them. However, method pointers when applied to static methods can be a very useful way to create DSL shortcut keywords.

Looking back at the GeeTwitter DSL from Chapter 4, we ended up with a neat DSL script that could search for tweets and automatically follow the Twitter user responsible for the tweet.

```
"Groovy DSL".search { fromUser, message ->
    GeeTwitter.follow fromUser
}
```

How about improving this by adding a method pointer to alias the `GeeTwitter.follow` method:

```
def follow = GeeTwitter.&follow

"Groovy DSL".search { fromUser, message ->
    follow fromUser
}
```

By defining a method pointer variable called `follow` in the current scope, we have a convenient alias to use instead of `GeeTwitter.follow()`. This is a much more natural way to do a Twitter follow than invoking a `static` method. This is especially useful when we are providing our DSL for use to a non-programming audience.

Metaprogramming and the Groovy MOP

In a nutshell, the term **metaprogramming** refers to writing code that can dynamically change its behavior at runtime. A **Meta-Object Protocol (MOP)** refers to the capabilities in a dynamic language that enable metaprogramming. In Groovy, the MOP consists of four distinct capabilities within the language: **reflection**, **metaclasses**, **categories**, and **expandos**.

The MOP is at the core of what makes Groovy so useful for defining DSLs. The MOP is what allows us to bend the language in different ways in order to meet our needs, by changing the behavior of classes on the fly. This section will guide you through the capabilities of MOP, and based on what we learn we will later dissect some builder code in order to understand how builders work under the covers.

Reflection

To use Java reflection, we first need to access the `Class` object for any Java object we are interested in through its `getClass()` method. Using the returned `Class` object, we can query everything from the list of methods or fields of the class to the modifiers that the class was declared with. Below, we see some of the ways that we can access a `Class` object in Java and the methods we can use to inspect the class at runtime.

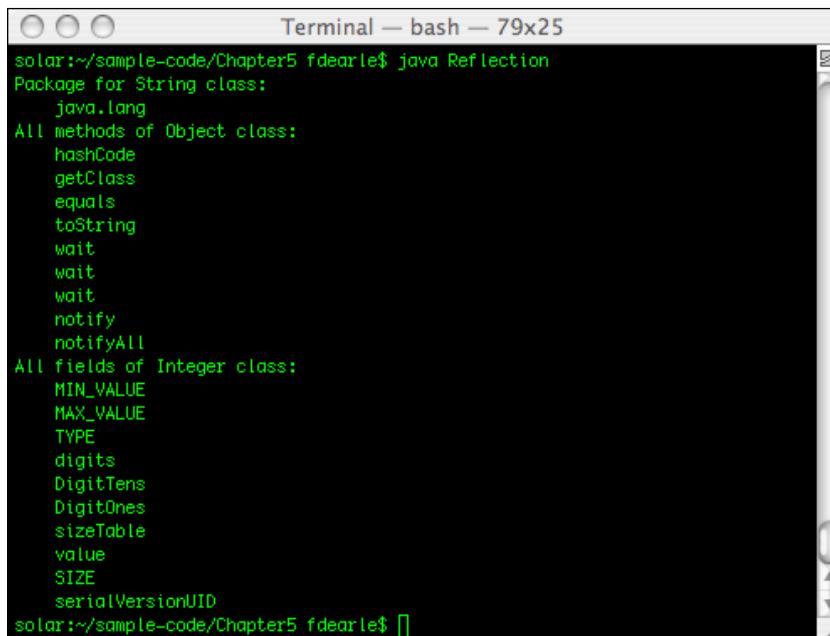
```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class Reflection {
    public static void main(String[] args) {
        String s = new String();
        Class sClazz = s.getClass();
        Package _package = sClazz.getPackage();
        System.out.println("Package for String class: ");
        System.out.println("    " + _package.getName());
        Class oClazz = Object.class;
        System.out.println("All methods of Object class:");
        Method[] methods = oClazz.getMethods();
        for(int i = 0;i < methods.length;i++)
            System.out.println("    " + methods[i].getName());
        try {
            Class iClazz = Class.forName("java.lang.Integer");
            Field[] fields = iClazz.getDeclaredFields();
            System.out.println("All fields of Integer class:");
            for(int i = 0; i < fields.length;i++)
```

```
        System.out.println("      " + fields[i].getName());
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
}
```

We can access the `Class` object from an instance by calling its `Object.getClass()` method. If we don't have an instance of the class to hand, we can get the `Class` object by using `.class` after the class name, for example, `String.class`. Alternatively, we can call the static `Class.forName`, passing to it a fully-qualified class name.

`Class` has numerous methods, such as `getPackage()`, `getMethods()`, and `getDeclaredFields()` that allow us to interrogate the `Class` object for details about the Java class under inspection. The preceding example will output various details about `String`, `Integer`, and `Double`.

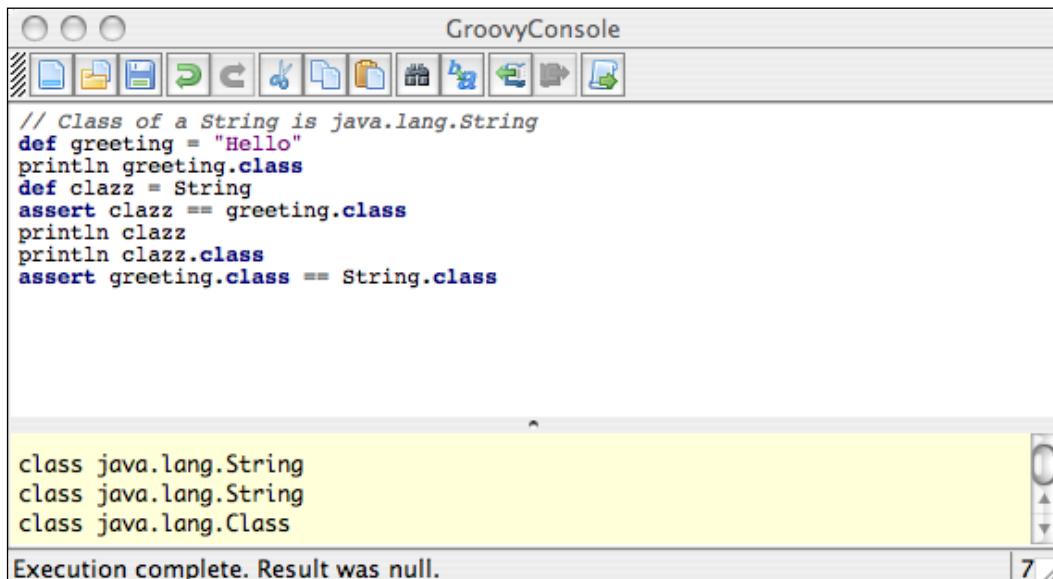


The screenshot shows a terminal window titled "Terminal - bash - 79x25". The window contains the following text output:

```
solar:~/sample-code/Chapter5 fdearle$ java Reflection
Package for String class:
  java.lang
All methods of Object class:
  hashCode
  getClass
  equals
  toString
  wait
  wait
  wait
  notify
  notifyAll
All fields of Integer class:
  MIN_VALUE
  MAX_VALUE
  TYPE
  digits
  DigitTens
  DigitOnes
  sizeTable
  value
  SIZE
  serialVersionUID
solar:~/sample-code/Chapter5 fdearle$
```

Groovy Reflection shortcuts

Groovy, as we would expect by now, provides shortcuts that let us reflect classes easily. In Groovy, we can shortcut the `getClass()` method as a property access `.class`, so we can access the class object in the same way whether we are using the class name or an instance. We can treat the `.class` as a `String`, and print it directly without calling `Class.getName()`, as follows:



The screenshot shows the GroovyConsole interface. The title bar says "GroovyConsole". Below the title bar is a toolbar with various icons. The main area contains Groovy code and its execution results.

```
// Class of a String is java.lang.String
def greeting = "Hello"
println greeting.class
def clazz = String
assert clazz == greeting.class
println clazz
println clazz.class
assert greeting.class == String.class
```

Execution results:

```
class java.lang.String
class java.lang.String
class java.lang.Class
```

Execution complete. Result was null.

The variable `greeting` is declared with a dynamic type, but has the type `java.lang.String` after the "Hello" String is assigned to it. Classes are first class objects in Groovy so we can assign `String` to a variable. When we do this, the object that is assigned is of type `java.lang.Class`. However, it describes the `String` class itself, so printing will report `java.lang.String`.

Groovy also provides shortcuts for accessing packages, methods, fields, and just about all other reflection details that we need from a class. We can access these straight off the class identifier, as follows:

```
println "Package for String class"
println "    " + String.package
println "All methods of Object class:"
Object.methods.each { println "    " + it }
println "All fields of Integer class:"
Integer.fields.each { println "    " + it }
```

Incredibly, these six lines of code do all of the same work as the 30 lines in our Java example. If we look at the preceding code, it contains nothing that is more complicated than it needs to be. Referencing `String.package` to get the Java package of a class is as succinct as you can make it. As usual, `String.methods` and `String.fields` return Groovy collections, so we can apply a closure to each element with the `each` method. What's more, the Groovy version outputs a lot more useful detail about the package, methods, and fields.

```
solar:~/sample-code/Chapter5 fdearle$ groovy Reflection.groovy
Package for String class
    package java.lang, Java Platform API Specification, version 1.5
All methods of Object class:
    public native int java.lang.Object.hashCode()
    public final native java.lang.Class java.lang.Object.getClass()
    public boolean java.lang.Object.equals(java.lang.Object)
    public java.lang.String java.lang.Object.toString()
    public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
    public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
    public final void java.lang.Object.wait() throws java.lang.InterruptedException
    public final native void java.lang.Object.notify()
    public final native void java.lang.Object.notifyAll()
All fields of Integer class:
    public static final int java.lang.Integer.MIN_VALUE
    public static final int java.lang.Integer.MAX_VALUE
    public static final java.lang.Class java.lang.Integer.TYPE
    public static final int java.lang.Integer.SIZE
solar:~/sample-code/Chapter5 fdearle$ []
```

When using an instance of an object, we can use the same shortcuts through the `class` field of the instance.

```
def greeting = "Hello"
assert greeting.class.package == String.package
```

Expando

An `Expando` is a dynamic representation of a typical Groovy bean. `Expando`s support typical `get` and `set` style bean access but in addition to this they will accept gets and sets to arbitrary properties. If we try to access a non-existing property, the `Expando` does not mind and instead of causing an exception it will return `null`. If we set a non-existent property, the `Expando` will add that property and set the value. In order to create an `Expando`, we instantiate an object of class `groovy.util.Expando`.

```
def customer = new Expando()

assert customer.properties == [:]

assert customer.id == null
```

```
assert customer.properties == [:]

customer.id = 1001
customer.firstName = "Fred"
customer.surname = "Flintstone"
customer.street = "1 Rock Road"

assert customer.id == 1001

assert customer.properties == [
    id:1001, firstName:'Fred',
    surname:'Flintstone', street:'1 Rock Road']
customer.properties.each { println it }
```

The `id` field of `customer` is accessible on the `Exando` shown in the preceding example even when it does not exist as a property of the bean. Once a property has been set, it can be accessed by using the normal field getter: for example, `customer.id`. `Expando`s are a useful extension to normal beans where we need to be able to dump arbitrary properties into a bag and we don't want to write a custom class to do so.

A neat trick with `Expando`s is what happens when we store a closure in a property. As we would expect, an `Exando` closure property is accessible in the same way as a normal property. However, because it is a closure we can apply function call syntax to it to invoke the closure. This has the effect of seeming to add a new method on the fly to the `Exando`.

```
customer.prettyPrint = {
    println "Customer has following properties"
    customer.properties.each {
        if (it.key != 'prettyPrint')
            println "    " + it.key + ": " + it.value
    }
}

customer.prettyPrint()
```

Here we appear to be able to add a `prettyPrint()` method to the `customer` object, which outputs to the console:

```
Customer has following properties

surname: Flintstone
street: 1 Rock Road
firstName: Fred
id: 1001
```

Categories

Adding a closure to an Expando to give a new method is a useful feature, but what if we need to add methods to an existing class on the fly? Groovy provides another useful feature—**Categories**—for this purpose. A Category can be added to any class at runtime, by using the `use` keyword.

We can create Category classes that add methods to an existing class. To create a Category for class, we define a class containing static methods that take an instance of the class that we want to extend as their first parameter. By convention, we name this parameter as `self`. When the method is invoked, `self` is set to the object instance that we are extending. The Category can then be applied to any closure by using the `use` keyword.

```
class Customer {  
    int id  
    String firstName  
    String surname  
    String street  
    String city  
}  
  
def fred = new Customer(id:1001,firstName:"Fred",  
surname:"Flintstone",  
street:"1 Rock Road",city:"Bedrock")  
def barney = new Customer(id:1002,firstName:"Barney",  
surname:"Rubble",  
street:"2 Rock Road",city:"Bedrock")  
  
def customerList = [ fred, barney]  
  
class CustomerPrinter {  
    static void prettyPrint(Customer self) {  
        println "Customer has following properties"  
        self.properties.each {  
            if (it.key != 'prettyPrint')  
                println "    " + it.key + ": " + it.value  
        }  
    }  
}  
  
use (CustomerPrinter) {  
    for (customer in customerList)  
        customer.prettyPrint()  
}
```

Java libraries are full of classes that have been declared `final`. The library designers in their wisdom have decided that the methods they have added are all that we will ever need. Unfortunately, that is almost never the case in practice. Take the Java `String` class, for example. There are plenty of useful string manipulation features that we might like to have in the `String` class. Java has added methods progressively to this class over time: for instance, `match` and `split` in Java 1.4, with `replace` and `format` being added in Java 1.5.

If we needed these style methods before Sun got around to adding them, we could not do it ourselves because of the `final` modifier. So the only option has been to use classes from add-on libraries such as Commons `StringUtils`. The Apache Commons Lang component class contains a slew of useful classes that augment the basic capabilities of Java classes, including `BooleanUtils`, `StringUtils`, `DateUtils`, and so on. All of the util class methods are implemented as static, taking `String` as the first parameter. This is the typical pattern used in Java when we need to mix in extra functionality to an existing class.

```
import org.apache.commons.lang.StringUtils;

public class StringSplitter {

    public static void main(String[] args) {
        String [] splits = StringUtils.split(args[0], args[1]);

        for (int i = 0; i < splits.length; i++) {
            System.out.println("token : " + splits[i]);
        }
    }
}
```

Conveniently, this pattern is the same as the one used Groovy categories, which means that the Apache Commons Lang Util classes can all be dropped straight into a `use` block. So all of these useful utility classes are ready to be used in your Groovy code as Categories.

```
import org.apache.commons.lang.StringUtils

use (StringUtils) {
    "org.apache.commons.lang".split(".").each { println it }
}
```

Metaclass

In addition to the regular Java `Class` object that we saw earlier when looking at reflection, each Groovy object also has an associated `MetaClass` Object. All Groovy classes secretly implement the `groovy.lang.GroovyObject` interface, which exposes a `getMetaClass()` method for each object.

```
public interface GroovyObject {  
    /**  
     * Invokes the given method.  
     */  
    Object invokeMethod(String name, Object args);  
  
    /**  
     * Retrieves a property value.  
     */  
    Object getProperty(String propertyName);  
  
    /**  
     * Sets the given property to the new value.  
     */  
    void setProperty(String propertyName, Object newValue);  
  
    /**  
     * Returns the metaclass for a given class.  
     */  
    MetaClass getMetaClass();  
  
    /**  
     * Allows the MetaClass to be replaced with a  
     * derived implementation.  
     */  
    void setMetaClass(MetaClass metaClass);  
}
```

Pure Java classes used in Groovy do not implement this interface, but they have a `MetaClass` assigned anyway. This `MetaClass` is stored in the `MetaClass` registry. Earlier versions of Groovy required a look-up in the registry to access the `MetaClass`. Since Groovy 1.5, the `MetaClass` of any class can be found by accessing its `.metaClass` property.

```
class Customer {  
    int id  
    String firstName  
    String surname  
    String street
```

```
        String city
    }

// Access Groovy meta class
def groovyMeta = Customer.metaClass

// Access Java meta class from 1.5
def javaMeta = String.metaClass

// Access Groovy meta class prior to 1.5
def javaMetaOld = GroovySystem.metaClassRegistry.getMetaClass(String)
```

Metaclasses are the secret ingredients that make the Groovy language dynamic. The `MetaClass` maintains all of the metadata about a Groovy class. This includes all of its available methods, fields, and properties. Unlike the `Java class` object, the Groovy `MetaClass` allows fields and methods to be added on the fly. So while the `Java class` can be considered as describing the compile time behavior of the class, the `MetaClass` describes its runtime behavior. We cannot change the `Class` behavior of an object but we can change its `MetaClass` behavior by adding properties or methods on the fly.

The Groovy runtime maintains a single `MetaClass` per Groovy class, and these operate in close quarter with the `GroovyObject` interface. `GroovyObject` implements a number of methods, which in their default implementations are just facades to the equivalent `MetaClass` methods. The most important of these to understand is the `invokeMethod()`.

Pretended methods (`MetaClass.invokeMethod`)

An important distinction between Java and Groovy is that in Groovy a method call never invokes a class method directly. A method invocation on an object is always dispatched in the first place to the `GroovyObject.invokeMethod()` of the object. In the default case, this is relayed onto the `MetaClass.invokeMethod()` for the class and the `MetaClass` is responsible for looking up the actual method. This indirect dispatching is the key to how a lot of Groovy power features work as it allows us to hook ourselves into the dispatching process in interesting ways.

```
class Customer {
    int id
    String firstName
    String surname
    String street
    String city
```

```
Object invokeMethod(String name, Object args) {
    if (name == "prettyPrint") {
        println "Customer has following properties"
        this.properties.each {
            println "    " + it.key + ": " + it.value
        }
    }
}

def fred = new Customer(id:1001,firstName:"Fred",
    surname:"Flintstone", street:"1 Rock Road",city:"Bedrock")
def barney = new Customer(id:1002,firstName:"Barney",
    surname:"Rubble", street:"2 Rock Road",city:"Bedrock")

def customerList = [ fred, barney]

customerList.each { it.prettyPrint() }
```

Above, we added a `Customer.invokeMethod()` to the `Customer` class. This allows us to intercept method invocations and respond to calls to `Customer.prettyPrint()` even though this method does not exist. Remember how in `GroovyMarkup` we appeared to be calling methods that did not exist? This is the core of how `GroovyMarkup` works. The `Customer.prettyPrint()` method in the previous code snippet is called a **pretended method**.

Understanding this, delegate, and owner

Like Java, Groovy has a `this` keyword that refers to the "current" or enclosing Java object. In Java, we don't have any other context that we can execute code in except a class method. In an instance method, `this` will always refer to the instance itself. In a static method, `this` has no meaning as the compiler won't allow us to reference `this` in a static context.

In addition to the instance methods, Groovy has three additional execution contexts to be aware of:

- Code running directly within a script where the enclosing object is the script.
- Closure code where the enclosing object is either a script or an instance object.
- Closure code where the enclosing object is another closure.

In addition to the `this` keyword, Groovy has two other keywords that are referred only in the context of a closure – `owner` and `delegate`.

- The `owner` keyword refers to the enclosing object, which in the majority of cases is the same as `this`, the only exception being when a closure is surrounded by another closure.
- The `delegate` keyword refers to the enclosing object and is usually the same as `owner` except that `delegate` is assignable to another object. Closures relay method invocations that they handle themselves back to their `delegate`. This is how the methods of an enclosing class become available to be called by the closure as if the closure was also an instance method. We will see later that one of the reasons builders work the way they do is because they are able to assign the `delegate` of a closure to themselves.

 The delegate will initially default to `owner`, except when we explicitly change the delegate to something else through the `Closure.setDelegate` method.

The following example illustrates this, `owner`, and `delegate` working under various different contexts. This example is necessarily complex, so take the time to read and understand it.

```
class Clazz {
    void method() {
        println "Class method this is : " + this.class
    }
    void methodClosure() {
        def methodClosure = {
            println "Method Closure this is : " + this.class
            assert owner == this
            assert delegate == this
        }
        methodClosure()
    }
}

def clazz = new Clazz()

clazz.method()

def closure = { self ->
    println "Closure this is : " + this.class
    assert this == owner
}
```

```
assert delegate == clazz
def closureClosure = {
    println "Closure Closure this is : " + this.class
    assert owner == self
    assert delegate == self
}
assert closureClosure.delegate == self

closureClosure()

}

closure.delegate = clazz
closure(closure)
clazz.methodClosure()

println this.class
```

Running the preceding code will output the following text:

```
Class method this is : class Clazz
Closure this is : class ConsoleScript1
Closure Closure this is : class ConsoleScript1
Method Closure this is : class Clazz
Script this is : class ConsoleScript1
```

So the rules for resolving `this`, `owner`, and `delegate` in the various contexts are:

- In a class instance method, `this` is always the instance object. `owner` and `delegate` are not applicable and will be disallowed by the compiler.
- In a class static method, `this`, `owner`, and `delegate` references will be disallowed by the compiler.
- In a closure defined within a script, `this`, `owner`, and `delegate` all refer to the `Script` object unless `delegate` has been reassigned.
- In a closure within a method, `this` and `owner` refer to the instance object of the enclosing class; as will `delegate`, unless it has been reassigned to another object.
- In a script, `this` is the `Script` object, and `owner` and `delegate` are not applicable.

How Builders work

Earlier, when we looked at the `MarkupBuilder` code, the unfamiliar syntax must have seemed strange. Now that we have an understanding of how the MOP and pretended methods work, let's take a quick look again at some builder code and, see if we can figure out what might be happening. `MarkupBuilder` is derived from the `BuilderSupport` class. When describing how `MarkupBuilder` works, I won't make a distinction between `BuilderSupport` and `MarkupBuilder`. Most of the mechanism described here is in fact implemented by `BuilderSupport` and is shared with other `Builder` classes.

```
def customers = builder.customers {  
    customer(id:1001) {  
        name(firstName:"Fred", surname:"Flintstone")  
        address(street:"1 Rock Road", city:"Bedrock")  
    }  
    customer(id:1002) {  
        name(firstName:"Barney", surname:"Rubble")  
        address(street:"2 Rock Road", city:"Bedrock")  
    }  
}
```

No matter how far you look in the documentation for `MarkupBuilder`, you won't find anything about it having a `customers` method. So what's happening when we write:

```
def customers = builder.customers {  
    ...
```

The answer is that `MarkupBuilder` is pretending to have a method with the signature: `MarkupBuilder.customers(Closure c)`. In the next line of code, things get a little more interesting. This line of code is defined within the body of the closure itself.

```
    customer(id:1001) {  
        ...
```

To explain this, we need to understand how closures handle method calls. When a closure encounters a method call that it cannot handle itself, it automatically relays the invocation to its owner object. If this fails, it relays the invocation to its delegate. Normally, the delegate would be the enclosing script or class, but the `MarkupBuilder` sets the delegate to itself. The closure relays the `customer` method invocation to `MarkupBuilder`, which has an `invokeMethod()` implementation that pretends to have a method `MarkupBuilder.customer(Map m, Closure c)`.

Method invocation and property lookup are governed by the **resolve strategy** of the Closure. The resolve strategy tells the Closure what objects it should look at when attempting to resolve a method or property reference. By default, the Resolve Strategy is set to `OWNER_FIRST`, which means that the first place we look is in the owner. If this lookup fails, then the search continues to the delegate object.

`MarkupBuilder` relies on the default resolve strategy, but we can change the resolve strategy as the need arises. The full list of resolve strategies is as follows:

- `OWNER_FIRST` (the default): Resolve methods and properties in the owner first followed by the delegate if not found.
- `DELEGATE_FIRST`: Resolve in the delegate first and then search the owner if not found.
- `OWNER_ONLY`: Resolve in the owner only and don't search the delegate.
- `DELEGATE_ONLY`: Resolve in the delegate only with no search of the owner.
- `TO_SELF`: This is a special case to allow `getProperty` of the Closure itself to be overridden. With this resolve strategy, the closure calls `getProperty` on itself first before continuing the lookup through the normal lookup process.

Coming back to our markup processing, the next line of code is in the context of a closure within a closure:

```
name(firstName: "Fred", surname: "Flintstone")
```

At this level of nesting, the delegate would normally refer to the enclosing script or instance object. Once again `MarkupBuilder` has reassigned the delegate to refer to itself. When this Closure relays the invocation up to its delegate, `MarkupBuilder.invokeMethod()` handles it and again pretends it has a method `MarkupBuilder.name(Map m, Closure c)`.

With each of these pretended methods, `MarkupBuilder` outputs a tag with the name of the method and attributes set according to the named parameters, and then calls the closure. As with most things in Groovy, building your own Builder is surprisingly easy when you know how to do it.

```
class PoorMansTagBuilder {  
    int indent = 0  
    Object invokeMethod(String name, Object args) {
```

```
    indent.times {print "      "}
    println "<${name}>"
    indent++
    args[0].delegate = this // Change delegate to the builder
    args[0].call()
    indent--
    indent.times {print "      "}
    println "</${name}>"
  }
}

def builder = new PoorMansTagBuilder ()

builder.root {
  level1{
    level2 {
    }
  }
}
```

In order to illustrate the builder mechanism shown in the previous code snippet, we are conveniently ignoring any parameter passing, and assuming that `args` just contains a first parameter of type `Closure`. However, this short example does illustrate method pretending through the `invokeMethod()`, and method relaying by assigning the `delegate`. The code will output a nice tagged representation of our `GroovyMarkup` code.

```
<root>
  <level1>
    <level2>
      </level2>
    </level1>
  </root>
```

ExpandoMetaClasses

We briefly touched on metaclasses when building our Twitter DSL in Chapter 4. In the coming example, we've used `String.metaClass` to dynamically add a method to the `String` class for Twitter searching. Let's look at what is happening here.

```
String.metaClass.search = { Closure c ->
  GeeTwitter.search(delegate,c)
}
```

From the earlier section on Expando, we understand how an Expando allows us to dynamically add a property to a class. That's all that is happening here. In the above code, we are dynamically adding a property to the `MetaClass` for `String` in the same way as we added properties to the Expando. This property happens to be a `Closure`, and the object happens to be the `MetaClass` of `String`, so it has the effect of adding a new method to the `String` class.

Adding a regular property to a `MetaClass` can be achieved in the same way as with Expando. There is only a single `MetaClass` per Groovy or Java class, so this is useful only if we have a new property that is relevant across all instances of a class. In practice, this will rarely happen. Apart from adding properties and methods, there are a whole bunch of other interesting things that we can do with the `ExpandoMetaClass`. We will go through a selection of these here.

Replacing methods

The technique that we use to add a method can also be used to replace an existing method. When doing so, we can subvert the existing logic of a class. Wouldn't it be nice if we could change all bank managers' minds as easily as this?

```
class BankManager {  
    def loan_approval_status = false  
  
    boolean approveLoan() {  
        return loan_approval_status  
    }  
}  
  
def myBankManager = new BankManager()  
  
assert myBankManager.approveLoan() == false  
  
BankManager.metaClass.approveLoan = { true }  
  
myBankManager = new BankManager()  
assert myBankManager.approveLoan() == true
```

Any method can be overridden or added. This includes any of the operator methods, such a `plus()`, `minus()`, `multiply()`, `divide()`, and so on. If need be, we can add operator semantics to any class, even if we have not written it ourselves.

Adding or overriding static methods

To add or override a static method of a class, we just insert the `static` keyword before the method name. In this example, we take the abstract `Calendar` class and supply a static `getInstance` method that instantiates a `GregorianCalendar` object. We then add a static `now` method that makes use of the new instance.

```
Calendar.metaClass.static.getInstance = { new GregorianCalendar() }

println Calendar.getInstance().getTime()

Calendar.metaClass.static.now = { Calendar.getInstance().getTime() }

println Calendar.now()
```

Dynamic method naming

We can use GStrings to name methods as we add or override them in a class. This means that we can dynamically generate method names on the fly. In the following example, we iterate all of the properties in the `Customer` class. We can exclude the `class` and `metaClass` properties with the `find` operator `it =~ /lass/` so that we just add methods for the properties that we want.

```
class Customer {
    def firstName
    def lastName
    def address1
    def address2
}

def c = new Customer()
// Find all properties except class and metaClass and add
// a new print(field) method
c.properties.keySet().findAll { !(it =~ /lass/) } .each {
    Customer.metaClass."print_${it}" = { -> println delegate."${it}" }
}
def cust = new Customer(firstName:"Fred",
                       lastName:"Flintstone",
                       address1:"Rock Road",
                       address2:"Bedrock")

cust.print(firstName)
cust.print(lastName)
```

Adding overloaded methods

Whenever we add a method to the `ExpandoMetaClass` that has the same signature as an existing method, the original method is overridden. In the following snippet, we can see that it is the last `String blanked` method that is in place after we override on subsequent occasions.

```
String.metaClass.blanked = { delegate.replaceAll(/./) {'%' } }
String.metaClass.blanked = { delegate.replaceAll(/./) {'@' } }
String.metaClass.blanked = { delegate.replaceAll(/./) {'*' } }

assert "A String".blanked() == "*****"
```

To add overloaded versions of methods, we can continue to add new methods. As long as the signatures are different from the last each method, it will be added as an overloaded method.

```
String.metaClass.static.valueAndType = { double d ->
    "${d.class.name}:\${valueOf(d)}"
}
String.metaClass.static.valueAndType = { float f ->
    "${f.class.name}:\${valueOf(f)}"
}
String.metaClass.static.valueAndType = { int i ->
    "${i.class.name}:\${valueOf(i)}"
}
String.metaClass.static.valueAndType = { long l ->
    "${l.class.name}:\${valueOf(l)}"
}

assert String.valueAndType(1.0) == "java.lang.Double:1.0"
assert String.valueAndType(3.333f) == "java.lang.Float:3.333"
assert String.valueAndType(101) == "java.lang.Integer:101"
assert String.valueAndType(1000000L) == "java.lang.Long:1000000"
```

When we are overloading subsequent methods with different signatures, we can make use of the append operator `<<`.

```
String.metaClass {
    static.valueAndType << { double d ->
        "${d.class.name}:\${valueOf(d)}"
    }
    static.valueAndType << { float f ->
        "${f.class.name}:\${valueOf(f)}"
    }
}
```

```
static.valueAndType << { int i ->
    "${i.class.name}:\${valueOf(i)}"
}
static.valueAndType << { long l ->
    "${l.class.name}:\${valueOf(l)}"
}
}

assert String.valueAndType(1.0) == "java.lang.Double:1.0"
assert String.valueAndType(3.333f) == "java.lang.Float:3.333"
assert String.valueAndType(101) == "java.lang.Integer:101"
assert String.valueAndType(1000000L) == "java.lang.Long:1000000"
```

Adding constructors

Constructors can be added to a class by using the constructor property of the metaclass. Just be wary when doing this so as not to call the default constructor. The mechanism used by the metaclass to call the constructor will cause a stack overflow, if you do.

```
class Customer {
    def firstName
    def lastName
    def address1
    def address2
}

Customer.metaClass.constructor = {
    String first, String last -> new Customer(
        firstName:first,
        lastName:last)
}

def c = new Customer("Fred", "Flintstone")
```

Summary

We have covered a lot of ground in this chapter. We have now covered all of the important features of the Groovy language, and looked in depth at how some of these features can be applied to developing DSLs. We now have an appreciation of what can be achieved by using features in the MOP, and how using the MOP enables other powerful features, such as GroovyMarkup.

Now that we have a better familiarity with the Groovy language, it's time to take a look at some of the readily-available DSLs and figure out how they are implemented within the Groovy language. In the next chapter, we will take a look at some Groovy-based DSLs, such as **Gant** and **GSpec**. This will give us a greater understanding of how to use Groovy power features in concert, in order to implement fully-functioning DSLs of our own.

6

Existing Groovy DSLs

By now we have covered a lot of ground in describing the DSL-enabling features of Groovy. In this chapter, we will look at some of the existing Groovy DSLs that are freely available for download. The purpose of this chapter is not to try to give a comprehensive tutorial on any of them. We will explore each in turn in order to understand how they work, but more importantly we will go through them in order to understand how they are implemented using the Groovy features and techniques that we have covered in the book so far.

- The **Grails Object Relational Mapping (GORM)** is a core component of the Grails web application framework and uses DSL- style techniques to apply persistence to regular Groovy classes. We will be looking at how GORM decorates regular POGO classes to add persistence semantics to them.
- Gant is an Ant-based build system that uses the `AntBuilder` class to add Groovy-scripting capabilities to Ant. We will be looking at how Gant uses the Builder paradigm to provide Groovy-scripted builds on top of Ant.
- GSpec, EasyB, and Spock are **Behavior Driven Development (BDD)** tools. BDD is all about enabling collaboration between developers, QA, and non-technical stakeholders in a software development environment. These tools aim to enable the development of specifications in a Groovy-based DSL that can be run as tests in order to validate the completeness of a system. To achieve this, specifications need to be written in a language that can be understood by all of the various stakeholders. We will see how these three tools use Groovy to try to achieve this aim.

The Grails Object Relational Mapping (GORM)

The Grails framework is an open source web application framework built for the Groovy language. Grails not only leverages Hibernate under the covers as its persistence layer, but also implements its own Object Relational Mapping layer for Groovy, known as GORM. With GORM, we can take a POGO class and decorate it with DSL-like settings in order to control how it is persisted. GORM can be considered a DSL as it uses many of the cool DSL features that we have already discussed, in order to add its own mini dialect to Groovy in order to implement persistence.

Grails programmers use GORM classes as a mini language for describing the persistent objects in their application. In this section, we will do a whistle-stop tour of the features of Grails. This won't be a tutorial on building Grails applications, as the subject is too big to be covered here. Our main focus will be on how GORM implements its Object model in the domain classes.

Grails quick start

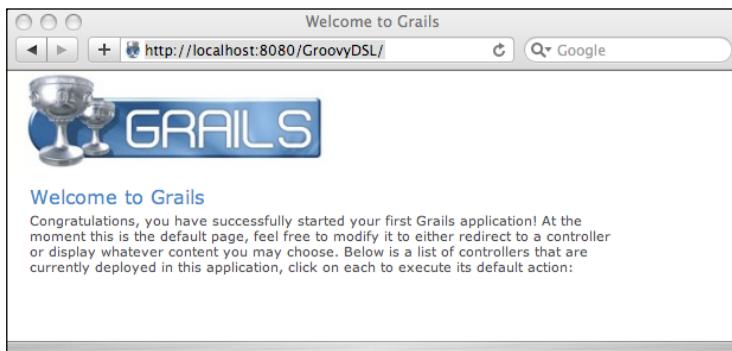
Before we proceed, we need to install Grails and get a basic app installation up and running. The Grails' download and installation instructions can be found at <http://www.grails.org/Installation>. Once it has been installed, and with the Grails binaries in your path, navigate to a workspace directory and issue the following command:

```
grails create-app GroovyDSL
```

This builds a Grails application tree called GroovyDSL under your current workspace directory. If we now navigate to this directory, we can launch the Grails app. By default, the app will display a welcome page at <http://localhost:8080/GroovyDSL/>.

```
cd GroovyDSL
```

```
grails run-app
```



The grails-app directory

The **GroovyDSL** application that we built earlier has a `grails-app` subdirectory, which is where the application source files for our application will reside. We only need to concern ourselves with the `grails-app/domain` directory for this discussion, but it's worth understanding a little about some of the other important directories.

- `grails-app/conf`: This is where the Grails configuration files reside.
- `grails-app/controllers`: Grails uses a **Model View Controller (MVC)** architecture. The controller directory will contain the Groovy controller code for our UIs.
- `grails-app/domain`: This is where Grails stores the GORM model classes of the application.
- `grails-app/view`: This is where the **Groovy Server Pages (GSPs)**, the Grails equivalent to **JSPs** are stored.

Grails has a number of shortcut commands that allow us to quickly build out the objects for our model. As we progress through this section, we will take a look back at these directories to see what files have been generated in these directories for us.



In this section, we will be taking a whistle-stop tour through GORM. You might like to dig deeper into both GORM and Grails yourself. You can find further online documentation for GORM at <http://www.grails.org/GORM>.

DataSource configuration

Out of the box, Grails is configured to use an embedded HSQL in-memory database. This is useful as a means of getting up and running quickly, and all of the example code will work perfectly well with the default configuration. Having an in-memory database is helpful for testing because we always start with a clean slate. However, for the purpose of this section, it's also useful for us to have a proper database instance to peek into, in order to see how GORM maps Groovy objects into tables. We will configure our Grails application to persist in a MySQL database instance.

Grails allows us to have separate configuration environments for development, testing, and production. We will configure our development environment to point to a MySQL instance, but we can leave the production and testing environments as they are.

First of all we need to create a database, by using the `mysqladmin` command. This command will create a database called `groovydsl`, which is owned by the MySQL root user.

```
mysqladmin -u root create groovydsl
```

Database configuration in Grails is done by editing the `DataSource.groovy` source file in `grails-app/conf`. We are interested in the environments section of this file.

```
environments {
    development {
        dataSource {
            dbCreate = "create-drop"
            url = "jdbc:mysql://localhost/groovydsl"
            driverClassName = "com.mysql.jdbc.Driver"
            username = "root"
            password = ""
        }
    }
    test {
        dataSource {
            dbCreate = "create-drop"
            url = "jdbc:hsqldb:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:mem:testDb"
        }
    }
}
```

The first interesting thing to note is that this is a mini Groovy DSL for describing data sources. In the previous version, we have edited the development `dataSource` entry to point to the MySQL `groovydsl` database that we created.

In early versions of Grails, there were three separate `DataSource` files that need to be configured for each environment, for example, `DevelopmentDataSource.groovy`. The equivalent `DevelopmentDataSource.groovy` file would be as follows:



```
class DevelopmentDataSource {
    boolean pooling = true
    String dbCreate = "create-drop"
    String url = "jdbc:mysql://localhost/groovydsl"
    String driverClassName = "com.mysql.jdbc.Driver"
    String username = "root"
    String password = ""
}
```

The `dbCreate` field tells GORM what it should do with tables in the database, on startup. Setting this to `create-drop` will tell GORM to drop a table if it exists already, and create a new table, each time it runs. This will keep the database tables in sync with our GORM objects. You can also set `dbCreate` to `update` or `create`.



`DataSource.groovy` is a handy little DSL for configuring the GORM database connections. Grails uses a utility class—`groovy.util.ConfigSlurper`—for this DSL. The `ConfigSlurper` class allows us to easily parse a structured configuration file and convert it into a `java.util.Properties` object if we wish. Alternatively, we can navigate the `ConfigObject` returned by using dot notation. We can use the `ConfigSlurper` to open and navigate `DataSource.groovy` as shown in the next code snippet. `ConfigSlurper` has a built-in ability to partition the configuration by environment. If we construct the `ConfigSlurper` for a particular environment, it will only load the settings appropriate to that environment.

```
def development =
    new ConfigSlurper("development").parse(new
File('DataSource.groovy').toURL())
def production =
    new ConfigSlurper("production").parse(new
File('DataSource.groovy').toURL())
assert development.dataSource.dbCreate == "create-drop"
assert production.dataSource.dbCreate == "update"
def props = development.toProperties()
assert props["dataSource.dbCreate"] == "create-drop"
```

Building a GORM model

The `grails` command can be used as a shortcut to carve out GORM domain classes. We can create a domain class for `Customer` by issuing the following Grails command from the `GroovyDSL` application directory:

```
grails create-domain-class Customer
```

This will create a stub `Customer.groovy` file in the `grails-app/domain` directory, as follows:

```
class Customer {

    static constraints = {
    }
}
```

If we add some fields to this class, we can peek into the MySQL database to see how GORM automatically creates a table for this class.

```
class Customer {
    String firstName
    String lastName

    static constraints = {
    }
}
```

Now if we restart Grails by issuing the `grails run-app` command, we can inspect the resulting table in MySQL:

```
solar:~ fdearie$ mysql -u root groovydsl
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

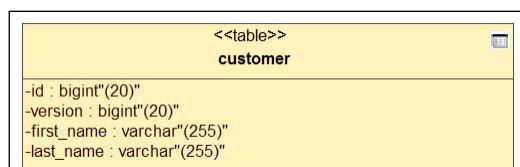
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.0.41 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> describe customer;
+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| id    | bigint(20) | NO  | PRI | NULL    | auto_increment |
| version | bigint(20) | NO  |      |          |          |
| first_name | varchar(255) | NO  |      |          |          |
| last_name | varchar(255) | NO  |      |          |          |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> [
```

GORM has automatically created a customer table for us in the `groovydsl` database. The table has two fields by default, for the row **`id`** and the object **`version`**. The two fields we added to `Customer` have been mapped to the `first_name` and `last_name` of Type `varchar(255)` columns in the database.

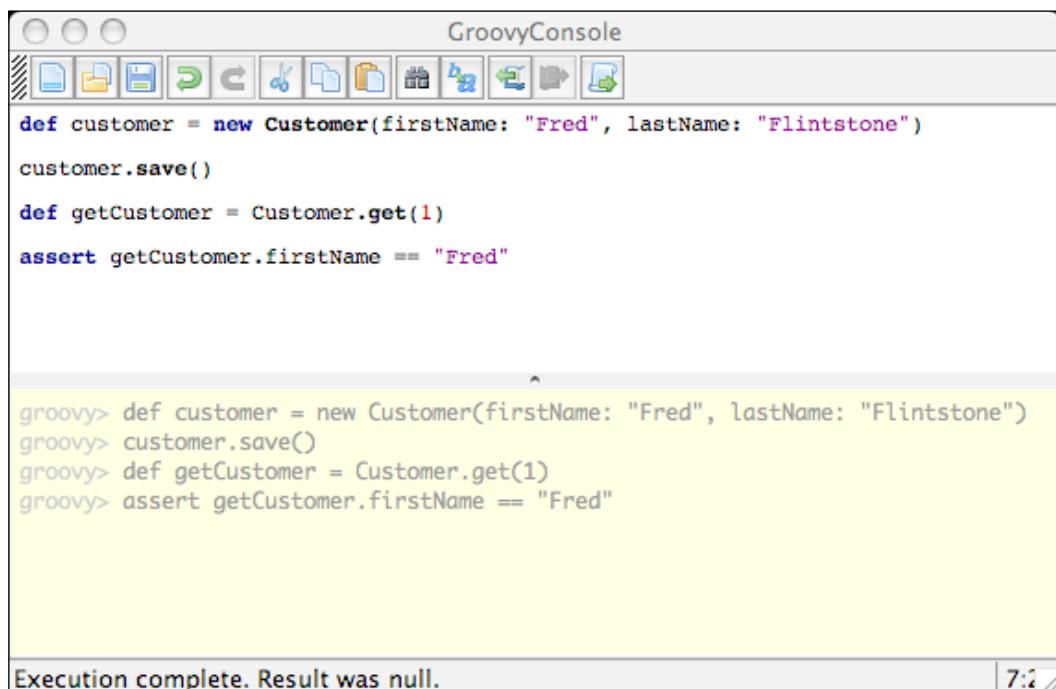


Using domain classes

Now that we have a domain class, how do we go about trying it out? We could dive in and write controllers and a view in order to build UI pages for the `Customer` object, which would be the norm if we were building a full-blown Grails application. We don't want to do this as all we want to do is look at how the GORM model is constructed.

Fortunately, Grails has some ways to let us interact directly with model classes without building any UI superstructure. The simplest of these is to use the `GroovyConsole` that we have used in our previous chapters. The `grails` command can be used to launch the console with the Grails environment running, so that model classes are immediately accessible.

```
$grails console
```



The screenshot shows the GroovyConsole interface. At the top, there's a toolbar with various icons. Below the toolbar, the code area contains the following Groovy script:

```
def customer = new Customer(firstName: "Fred", lastName: "Flintstone")
customer.save()
def getCustomer = Customer.get(1)
assert getCustomer.firstName == "Fred"
```

Below the code, the Groovy shell prompt shows the same script being run:

```
groovy> def customer = new Customer(firstName: "Fred", lastName: "Flintstone")
groovy> customer.save()
groovy> def getCustomer = Customer.get(1)
groovy> assert getCustomer.firstName == "Fred"
```

At the bottom of the interface, a message states "Execution complete. Result was null." and there's a status bar with the number "7:2".

A normal POGO class would only allow us to construct it and interact with its properties at this point. However, Grails has used the Groovy MOP to add some persistence methods to the `Customer` class. If we run the code shown in the previous example, we call the `save` method on a `Customer` object that causes it to be stored in the MySQL database. Using `mysql` we can confirm to our satisfaction that a row has actually been saved, by doing this:

```
mysql> select * from customer;
+----+-----+-----+
| id | version | first_name | last_name |
+----+-----+-----+
| 1 |      0 | Fred       | Flintstone |
+----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

We also have methods to `get()` an object from the database by ID, `update()` the object, and `list()` available objects.

```
def barney = new Customer(firstName: "Barney", lastName: "Rubble")
barney.save()

def fred = Customer.get(1)
fred.firstName = "Fred"
fred.save()

def customers = Customer.list()
customers.each { println c.id + ": " + c.firstName + " , " +
c.lastName }
```

The above example saves a new customer "Barney" to the database, gets the customer object for "Fred", and updates the `firstName` field. The `list` method returns all customers as a regular list collection so that we can apply a closure to all members to print them.

```
1: Fred, Flintstone
2: Barney, Rubble
```



Let's take a second to look at what Grails has done to our class. The `Customer` class that we declared had no base class to inherit methods from, and it did not define these methods. Grails has used the Groovy MOP to add these methods. When we run our Grails app in development mode, Grails iterates over all the domain classes in `grails-app/domain`. As it does so, it adds these persistence methods to the `MetaClass` of each domain class that it encounters. We have covered several different options as to how to add a method to a class on the fly. In this case, Grails has augmented the `MetaClass` for `Customer` with a combination of static and normal methods, as follows:

```
class Customer {
    ...
}
Customer.metaClass.static.get = { ... }
Customer.metaClass.static.list = { ... }
Customer.metaClass.save = { ... }
```

Modeling relationships

Storing and retrieving simple objects is all very well, but the real power of GORM is that it allows us to model the relationships between objects, as we will now see. The main types of relationships that we want to model are **associations**, where one object has an associated relationship with another, for example, `Customer` and `Account`, **composition relationships**, where we want to build an object from sub components, and **inheritance**, where we want to model similar objects by describing their common properties in a base class.

Associations

Every business system involves some sort of association between the main business objects. Relationships between objects can be one-to-one, one-to-many, or many-to-many. Relationships may also imply ownership, where one object only has relevance in relation to another parent object.

If we model our domain directly in the database, we need to build and manage tables, and make associations between the tables by using foreign keys. For complex relationships, including many-to-many relationships, we may need to build special tables whose sole function is to contain the foreign keys needed to track the relationships between objects. Using GORM, we can model all of the various associations that we need to establish between objects directly within the GORM class definitions. GORM takes care of all of the complex mappings to tables and foreign keys through a Hibernate persistence layer.

One-to-one

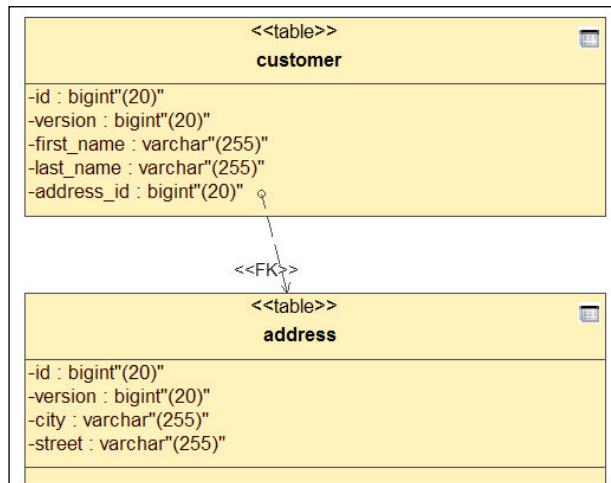
The simplest association that we need to model in GORM is a one-to-one association. Suppose our customer can have a single address; we would create a new `Address` domain class using the `grails create-domain-class` command, as before.

```
class Address {  
    String street  
    String city  
  
    static constraints = {  
    }  
}
```

To create the simplest one-to-one relationship with `Customer`, we just add an `Address` field to the `Customer` class.

```
class Customer {  
    String firstName  
    String lastName  
    Address address  
  
    static constraints = {  
    }  
}
```

When we rerun the Grails application, GORM will recreate a new address table. It will also recognize the `address` field of `Customer` as an association with the `Address` class, and create a foreign key relationship between the **customer** and **address** tables accordingly.



This is a one-directional relationship. We are saying that a Customer "has an" Address but an Address does not necessarily "have a" Customer.

We can model bi-directional associations by simply adding a Customer field to the Address. This will then be reflected in the relational model by GORM adding a `customer_id` field to the `address` table.

```
class Address {
    String street
    String city
    Customer customer

    static constraints = {
    }
}

mysql> describe address;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+
| id   | bigint(20) | NO   | PRI | NULL    | auto_increment |
| version | bigint(20) | NO   |     |          |                |
| city  | varchar(255) | NO   |     |          |                |
| customer_id | bigint(20) | YES  | MUL | NULL    |                |
| street | varchar(255) | NO   |     |          |                |
+-----+-----+-----+-----+-----+
5 rows in set (0.01 sec)
```

mysql>

These basic one-to-one associations can be inferred by GORM just by interrogating the fields in each domain class via reflection and the Groovy metaclasses. To denote ownership in a relationship, GORM uses an optional static field applied to a domain class, called `belongsTo`. Suppose we add an `Identity` class to retain the login identity of a customer in the application. We would then use

```
class Customer {
    String firstName
    String lastName
    Identity ident
}
```

```
class Address {  
    String street  
    String city  
}  
  
class Identity {  
    String email  
    String password  
  
    static belongsTo = Customer  
}
```



Classes are first-class citizens in the Groovy language. When we declare `static belongsTo = Customer`, what we are actually doing is storing a static instance of a `java.lang.Class` object for the `Customer` class in the `belongsTo` field. Grails can interrogate this static field at load time to infer the ownership relation between `Identity` and `Customer`.

Here we have three classes: `Customer`, `Address`, and `Identity`. `Customer` has a one-to-one association with both `Address` and `Identity` through the `address` and `ident` fields. However, the `ident` field is "owned" by `Customer` as indicated in the `belongsTo` setting. What this means is that saves, updates, and deletes will be cascaded to `identity` but not to `address`, as we can see below. The `addr` object needs to be saved and deleted independently of `Customer` but `id` is automatically saved and deleted in sync with `Customer`.

```
def addr = new Address(street:"1 Rock Road", city:"Bedrock")  
def id = new Identity(email:"email", password:"password")  
def fred = new Customer(firstName:"Fred",  
    lastName:"Flintstone",  
    address:addr, ident:id)  
  
addr.save(flush:true)  
  
assert Customer.list().size == 0  
assert Address.list().size == 1  
assert Identity.list().size == 0  
  
fred.save(flush:true)  
  
assert Customer.list().size == 1  
assert Address.list().size == 1  
assert Identity.list().size == 1  
  
fred.delete(flush:true)  
  
assert Customer.list().size == 0
```

```
assert Address.list().size == 1
assert Identity.list().size == 0

addr.delete(flush:true)

assert Customer.list().size == 0
assert Address.list().size == 0
assert Identity.list().size == 0
```

Constraints

You will have noticed that every domain class produced by the grails create-domain-class command contains an empty static closure, constraints. We can use this closure to set the constraints on any field in our model. Here we apply constraints to the e-mail and password fields of Identity. We want an e-mail field to be unique, not blank, and not nullable. The password field should be 6 to 200 characters long, not blank, and not nullable.

```
class Identity {
    String email
    String password

    static constraints = {
        email(unique: true, blank: false, nullable: false)
        password(blank: false, nullable:false, size:6..200)
    }
}
```

From our knowledge of builders and the markup pattern, we can see that GORM could be using a similar strategy here to apply constraints to the domain class. It looks like a pretended method is provided for each field in the class that accepts a map as an argument. The map entries are interpreted as constraints to apply to the model field.

The Builder pattern turns out to be a good guess as to how GORM is implementing this. GORM actually implements constraints through a builder class called ConstrainedPropertyBuilder. The closure that gets assigned to constraints is in fact some markup style closure code for this builder. Before executing the constraints closure, GORM sets an instance of ConstrainedPropertyBuilder to be the delegate for the closure. We are more accustomed to seeing builder code where the Builder instance is visible.

```
def builder = new ConstrainedPropertyBuilder()
builder.constraints {
}
```

Setting the builder as a delegate of any closure allows us to execute the closure as if it was coded in the above style. The constraints closure can be run at any time by Grails, and as it executes the `ConstrainedPropertyBuilder`, it will build a `HashMap` of the constraints it encounters for each field.

We can illustrate the same technique by using `MarkupBuilder` and `NodeBuilder`. The `Markup` class in the following code snippet just declares a static closure named `markup`. Later on we can use this closure with whatever builder we want, by setting the delegate of the `markup` to the builder that we would like to use.



```
class Markup {  
    static markup = {  
        customers {  
            customer(id:1001) {  
                name(firstName:"Fred",  
                      surname:"Flintstone")  
                address(street:"1 Rock Road",  
                        city:"Bedrock")  
            }  
            customer(id:1002) {  
                name(firstName:"Barney",  
                      surname:"Rubble")  
                address(street:"2 Rock Road",  
                        city:"Bedrock")  
            }  
        }  
    }  
    Markup.markup.setDelegate(new groovy.xml.  
        MarkupBuilder())  
    Markup.markup() // Outputs xml  
    Markup.markup.setDelegate(new groovy.util.  
        NodeBuilder())  
    def nodes = Markup.markup() // builds a node tree
```

One-to-many

A one-to-many relationship applies when an instance of class such as `customer` is associated with many instances of another class. For example, a customer may have many different invoices in the system, and an invoice might have a sale order object for each line on the invoice.

```
class Customer {  
    String firstName  
    String lastName
```

```

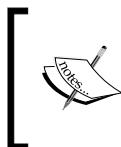
        statichasMany = [invoices:Invoice]
    }

class Invoice {
    statichasMany = [orders:SalesOrder]
}

class SalesOrder {
    String sku
    int amount
    Double price
    static belongsTo = Invoice
}

```

To indicate the "has many" associations between `Customer`/ `Invoice` and `Invoice`/ `SalesOrder`, we insert a static `hasMany` setting. We can also indicate ownership by using the `belongsTo` setting. In the preceding example, a sales order line has no relevance except on an invoice. We apply a `belongsTo` setting to bind it to the invoice. The `belongsTo` setting will cause deletes to cascade when the owning object is deleted. If an invoice is deleted, the delete will cascade to the sales order lines. `Invoice` does not belong to `Customer`, so for auditing purposes the invoice object will not be automatically deleted even if the customer is removed.



From the Groovy DSL point of view, `hasMany` is just a static table containing a map of IDs and `Class` objects. GORM can analyze this map at load time in order to create the correct table mappings.

GORM will automatically take care of the mapping between the `Customer`, `Invoice`, and `SalesOrder` classes by creating `invoice_sales_order` and `customer_invoice` join tables. We can peek at these by using `mysql`, to confirm that they exist.

```

mysql> describe customer_invoice;
+-----+-----+-----+-----+
| Field          | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| customer_invoice_id | bigint(20) | YES  | MUL | NULL    |       |
| invoice_id      | bigint(20) | YES  | MUL | NULL    |       |
+-----+-----+-----+-----+
2 rows in set (0.01 sec)

```

```
mysql>
```

Customer objects are linked to Invoice objects through the customer_invoice join table. Invoices are linked to sales orders via the invoice_sales_order join table, using foreign keys, as we can see from following DDL diagram:



The `hasMany` setting is defined as a Map containing a key and class for each domain object that is associated. For each `hasMany` key encountered on a domain class, GORM will inject an `addTo(key)` method. This translates to an `addToOrders` method, which is added to `Invoice`, as the key used was `Orders` and an `addToInvoices` method is added to `Customer`. Invoking these methods will automatically save an order or invoice in the database and also update the join tables with the correct keys.

```

def fred = new Customer(firstName:"Fred", lastName:"Flintstone")

fred.save()

def invoice = new Invoice()

invoice.addToOrders(new SalesOrder(sku:"productid01",
                                    amount:1, price:1.00))
  
```

```

invoice.addToOrders(new SalesOrder(sku:"productid02",
                                    amount:3, price:1.50))
invoice.addToOrders(new SalesOrder(sku:"productid03",
                                    amount:2, price:5.00))

fred.addToInvoices(invoice)

```

GORM adds the `addTo{association}` methods to a domain class by iterating the `hasMany` map in the class, and updating the `MetaClass` with a new method for each association that it encounters. Below we can emulate how GORM does just this. In this example, we iterate over the `hasMany` map of the `Customer` class. We add a closure method that returns a string corresponding to each map entry that we encounter. The closure names that we add are dynamically generated and include the key values from the `hasMany` map.



```

class Customer {
    static hasMany = [invoices:Invoice,
                     orders:SalesOrder]
}
class Invoice {
}
class SalesOrder {
}
CustomerhasMany.each {
    def nameSuffix = it.key.substring(0,1).toUpperCase()
    def relation = "${nameSuffix}${it.key.substring(1)}"
    Customer.metaClass."addTo${relation}" {
        "Add to ${relation}"
    }
}
def customer = new Customer()
assert customer.addToInvoices() == "Add to Invoices"
assert customer.addToOrders() == "Add to Orders"

```

When GORM establishes a one-to-many association with a Class, it also adds a field to the class with the same name as the key used in the `hasMany` map. So we can access the list of `orders` on an invoice as follows:

```
invoice.orders.each {println sku + " " + amount + " " + price}
```

Many-to-many

Many-to-many associations are rarer than any of the other associations, and can be tricky to model. In GORM, all we need to do is to make the association bi-directional and give ownership to one side by applying a `belongsTo` setting. GORM will take care of the rest.

A tunes database needs to model the fact that artistes perform on many songs but also that artistes collaborate on songs, so songs can have many artistes. Modeling this in GORM is the essence of simplicity.

```
class Artist {  
    String name  
    statichasMany = [songs:Song]  
}  
  
class Song {  
    String title  
    static belongsTo = Artist  
    statichasMany = [artists: Artist]  
}
```

When maintaining a database, it does not matter whether we add songs to artistes or artistes to songs—GORM will maintain both sides of the relationship.

```
def song1 = new Song(title:"Rich Woman")  
def song2 = new Song(title:"Killing the Blues")  
  
def artist1 = new Artist(name:"Jimmy Page")  
def artist2 = new Artist(name:"Alison Krauss")  
  
song1.addToArtists(artist1)  
song1.addToArtists(artist2)  
artist1.addToManySongs(song2)  
artist2.addToManySongs(song2)  
  
artist1.save()  
artist2.save()  
  
println artist1.name + " performs "  
artist1.songs.each { println "    " + it.title }  
println artist2.name + " performs"  
artist2.songs.each { println "    " + it.title }  
  
println song1.title + " performed by"  
song1.artists.each { println "    " + it.name }  
println song2.title + " performed by"  
song2.artists.each { println "    " + it.name }
```

We add both artistes to `song1` and then add `song2` to both artistes. We only need to save the artiste objects that are the owners of the relationships, and all associations are preserved. The example prints out the following:

```
Jimmy Page performs
    Killing the Blues
    Rich Woman
Alison Krauss performs
    Killing the Blues
    Rich Woman
Rich Woman performed by
    Jimmy Page
    Alison Krauss
Killing the Blues performed by
    Jimmy Page
    Alison Krauss
```

Composition

Composition is used when instead of having separate tables for each business object, we would like to embed the fields of a child object in the table of its parent. In the previous one-to-one examples, it may be useful for us to have `Address` and `Identity` classes to pass around in our Groovy code, but we may prefer the data for these to be rolled into one database table. To do this, all we need to do is to add an embedded setting to the `Customer` domain class, and GORM takes care of the rest.

```
class Customer {
    String firstName
    String lastName
    Address billing
    Address shipping
    Identity ident
    static embedded = ['billing','shipping','ident']
}

class Address {
    String street
    String city
}

class Identity {
    String email
    String password
}
```

We can see a useful application of embedding in the previous code snippet, where we needed two addresses—one for billing and one for shipping. This does not warrant a full one-to-many association between `Customer` and `Address` because we only ever have the two addresses. The fields from `Address` and `Identity` get mapped into columns in the `customer` table, as we can see here.

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 5.0.41 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> describe customer;
+-----+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id          | bigint(20) | NO   | PRI | NULL    | auto_increment |
| version     | bigint(20) | NO   |      |          |                |
| billing_city | varchar(255) | NO   |      |          |                |
| billing_street | varchar(255) | NO   |      |          |                |
| first_name  | varchar(255) | NO   |      |          |                |
| ident_email  | varchar(255) | NO   |      |          |                |
| ident_password | varchar(255) | NO   |      |          |                |
| last_name    | varchar(255) | NO   |      |          |                |
| shipping_city | varchar(255) | NO   |      |          |                |
| shipping_street | varchar(255) | NO   |      |          |                |
+-----+-----+-----+-----+-----+-----+
10 rows in set (0.00 sec)

mysql> 
```

Inheritance

By default, GORM implements inheritance relationships with a table-per-hierarchy. A class column in the table is used as a discriminator column.

```
class Account {
    double balance
}

class CardAccount extends Account {
    String cardId
}

class CreditAccount extends Account{
    double creditLimit
}
```

All of the properties from the hierarchy are mapped to columns in an account table. Entries in the table will have the class column set to indicate what class the object in the entry belongs to. You will note from the upcoming mysql table that all properties in the derived classes are set to nullable. This is one downside of using the table-per-hierarchy approach, but it can be overcome by using the mapping setting.

```
mysql> describe account;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| id | bigint(20) | NO | PRI | NULL | auto_increment |
| version | bigint(20) | NO | | | |
| balance | double | NO | | | |
| class | varchar(255) | NO | | | |
| credit_limit | double | YES | | NULL | |
| card_id | varchar(255) | YES | | NULL | |
+-----+-----+-----+-----+-----+
6 rows in set (0.01 sec)
```

mysql>

Mapping

We can use the mapping setting to apply a table-per-subclass strategy to inheritance mapping. This will overcome the need to allow all properties to be nullable. Mapping also allows us to map GORM domain classes onto a legacy database, as it gives us fine control over both table and column names in the relation model that we map to.

```
class Account {
    double balance

    static mapping = {
        table "fin_account"
        balance column:"acc_bal"
    }
}
```



The mapping closure is implemented in Grails in a fashion very similar to constraints. The mapping closure is executed, having set its delegate to an instance of `HibernateMappingBuilder`. Within the closure code for mapping, we can use any of the built-in methods to control how mapping should occur, or we can name the column field itself for more fine-grained control over the mapping of a column. To apply a table-per-subclass strategy and turn on caching, we can add the following:

```
static mapping = {
    tablePerSubclass = true
    cache = true
}
```

Querying

We've already seen in the examples some basic querying with `list()` and `get()`. The `list` method can be used with a number of named parameters that gives us finer control over the result set returned.

```
// list all customers
Customer.list().each {
    println "${it.firstName} ${it.lastName}"
}
// List the first 10 customers
Customer.list(max:10).each {
    println "${it.firstName} ${it.lastName}"
}
// List the next 10 customers
Customer.list(max:10, offset:10).each {
    println "${it.firstName} ${it.lastName}"
}
// List all customers in descending order sorted by last name
Customer.list(sort:"lastName",order:"desc").each {
    println "${it.firstName} ${it.lastName}"
```

We've seen the `get()` method in action, which uses the database ID to return an object. We can also use the `getAll()` method when we want to return more than one object, as long as we have the IDs that we need.

```
def customers = Customer.getAll(2, 4, 5, 7)
```

Dynamic finders

GORM supports a unique and powerful query function through the use of **dynamic finders**. A dynamic finder is a pretended method applied to a domain class, that can return one or many queried objects. Finders work by allowing us to invent method names, where the syntax of the query is bound into the method name. All finders are prefixed by `findBy` or `findAllBy`.

```
// Find the first customer called Fred
def fred = Customer.findByFirstName("Fred")
// Find all Flintstones
def flintstones = Customer.findAllByLastName("Flintstone")
// Find Fred Flintstone
def fred_flintstoner = Customer.findByFirstNameAndLastName("Fred",
"Flintstone")
```

Finder names can also include comparators, such as `Like`, `LessThan`, and `IsNotNull`.

```
// Find all customers with names beginning with Flint
def flints = Customer.findAllByLastNameLike("Flint%")
```

We can include associations in queries. Here we can find all invoices for Fred:

```
def fred = Customer.findByFirstNameAndLastName("Fred", "Flintstone")
def invoices = Invoice.findAllByCustomer(fred)
```

Dynamic finders open up a huge range of possible finder methods that can be used with a class. The more fields there are, the more possible finder methods there are to match. Rather than trying to second-guess what all of the possible combinations might be and adding them to the metaclass, GORM takes a slightly different approach.

From Groovy 1.0, we can add a method called `methodMissing` to the `MetaClass` object. This differs slightly from the `invokeMethod` call that we provided in previous chapters, because `methodMissing` is called as the last chance saloon before Groovy throws a `MethodMissingException`, whereas `invokeMethod` will be called for every method invocation.

Grails adds its own `methodMissing` implementation, which catches all of the missing method invocations with the prefixes `find` and `findAll` the first time that they occur. At that point, the actual method implementation is registered in the metaclass, so subsequent calls to that finder will not suffer the overhead.

GORM as a DSL

We've only scratched the surface of what can be done with GORM and Grails. For a more comprehensive view of GORM, take the time to look at the Grails reference at <http://www.grails.org>. The publishers of this book have also released an excellent reference, *Grails 1.1 Web Application Development* by Jon Dickinson, which covers GORM in much more depth than we have time to do here.

What is most interesting about GORM in relation to this book is how it achieves its goals through a suite of mini DSLs implemented in Groovy. DataSource configuration is achieved through a markup syntax that can be understood without any prior knowledge of Groovy. Domain classes could be designed and written by a database architect who can also be insulated from the intricacies of the Groovy language.

Gant

Apache Ant is the venerable old build system for the Java world. Arguably, Ant has been superseded in features and benefits by another Apache project, **Maven**. Ant still has a significant following and still is the de facto building standard for a lot of projects. Both systems now have Groovy scripting support in the form of Gant for Ant and GMaven for Maven. In this section we will be taking a look at Gant.

Ant

Ant was developed by **The Apache Software Foundation**. Ant was designed as a replacement for command-like make tools such as **make** and **nmake**. Ant was written in Java, so unlike other make tools it runs across platforms. Ant build scripts are written in XML. Build files contain one project and many target elements that can be defined to have dependencies on each other. Each target element will define one or more task elements, which are responsible for doing the actual work.

One benefit of Ant over existing make tools is that instead of invoking shell commands directly, Ant implements all of its functionalities through the task elements. Tasks are implemented as plugin modules to Ant itself. The plugin module takes care of any cross platform issues, so a `<mkdir>` task will create a directory regardless of what operating system we run on.

The plugin module approach for implementing tasks also means that additional tasks can be added to Ant in order to achieve any build operation we need. From its earliest days, there have been numerous additional plugin modules available for everything from building stripped down JAR files to deployment on every conceivable platform. The extensibility and modularity of the task plugin interfaces have been key to Ant's continued success over the years.

Here we take a look at an Ant build file that illustrates some of the important features. To try this out you need to edit the \${path-to} and \${version} to point to a copy of the groovy-all JAR file on your system.



Not all Groovy distributions have a copy of groovy-all-1.x.x.jar in lib. If it is missing from your distribution, you can download it from <http://groovy.codehaus.org>, or make use of the one that is packaged in the Grails distribution, under either the lib or embeddable directories.

```
<project name="hello" default="run">
    <property name="dist" value=". ./dist"/>
    <path id="hello.class.path">
        <pathelement location="${path-to}groovy-all-${version}.jar"/>
        <pathelement location="${dist}/classes"/>
        <pathelement path="${java.class.path}"/>
    </path>

    <taskdef name="groovyc"
        classname="org.codehaus.groovy.ant.Groovyc"
        classpathref="hello.class.path"/>

    <target name="init">
        <mkdir dir="${dist}/classes"/>
    </target>

    <target name="compile" depends="init">
        <groovyc srcdir=". " destdir="${dist}/classes" />
    </target>

    <target name="run" depends="compile">
        <java classname="Hello">
            <classpath refid="hello.class.path"/>
        </java>
    </target>
</project>
```

This simple Ant script defines three targets: `run`, `compile`, and `init`. The default target is `run`, which runs the next Groovy `Hello` script as a class file in the Java VM. The `run` target depends on `compile`, which compiles the Groovy script to a Java class file in the `dist/classes` directory. `Compile` depends on `init`, which creates a directory for classes.

Groovyc is not available in the default Ant distribution as an available task, so we use the `taskdef` tag to make the Groovyc Ant plugin available. This plugin is packaged in the `groovy-all-{version}.JAR` file. Ant is so pervasively used these days that it is commonplace now to find appropriate Ant tasks bundled with component and tool distributions.

Once everything is in place and Ant is installed, all we need to do is to invoke Ant from the command line to compile and run `Hello.groovy`.

```
// Hello.groovy

println "Hello, Groovy World!"


$ ant
Buildfile: build.xml

init:

compile:

run:
[java] Hello, Groovy World!

BUILD SUCCESSFUL
Total time: 2 seconds
$
```

AntBuilder

In our discussion on builders in the previous chapter, we came across `AntBuilder`, which is a Groovy builder class that implements a scripting interface to Ant through `GroovyMarkup`. With `AntBuilder`, we are able to run Groovy scripts that call Ant tasks directly, and can contain Groovy conditional logic. In the following example, we emulate the Ant script from above. The only things that we are missing with `AntBuilder` are the dependency-based targets.

```
def ant = new AntBuilder()
dist = "./dist"
taskdef name: "groovyc", classname: "org.codehaus.groovy.ant.Groovyc"

ant.mkdir(dir:"${dist}/classes")
ant.groovyc( srcdir: ".", destdir: "${dist}/classes",
    includes:"Hello.groovy")
ant.java(classname:"Hello") {
    classpath {
        pathelement path: "${path-to}groovy-all-${version}.jar "
        pathelement path: "${dist}/classes"
    }
}
```

To interpret the above code, we need to understand how `AntBuilder` treats method calls encountered within the `GroovyMarkup`.

- Method calls on the `AntBuilder` object translate to task names. These are pretended method calls. So `AntBuilder` will map the method call to an equivalent task plugin, and attempt to invoke it.
- Task attributes are passed to the pretended methods as maps, which allow us to use the named parameter syntax seen above.
- Nested code within closures map to nested Ant tasks or elements similar to the nesting in an Ant `build.xml` script.. In the following code example, we wrap the same code in an Ant sequential task.

```
def ant = new AntBuilder()
antSEQUENTIAL {
    dist = "./dist"
    taskdef name: "groovyc",
        classname: "org.codehaus.groovy.ant.Groovyc"
    mkdir(dir:"${dist}/classes")
    groovyc( srcdir: ".",
        destdir: "${dist}/classes",
        includes:"Hello.groovy")
    java(classname:"Hello") {
```

```
        classpath {
            pathelement path: "${path-to}groovy-all-
${version}.jar"
            pathelement path: "${dist}/classes"
        }
    }
}
```

An important point to note is that although the `AntBuilder` code is a `GroovyMarkup`, it is not producing any Ant build file XML. The mechanism that `AntBuilder` uses is to wrap the task plugins with Groovy scripting. So what is happening above is that the script is calling the tasks `sequential`, `taskdef`, `groovyc`, and so on, in sequence. We lose the dependency mechanism of Ant itself but we can add our own scripting logic, which may be more appropriate to what we need. Instead of a dependency, we can put more regular Groovy logic in our script:

```
def classesDir = new File("${dist}/classes")
if (!classesDir.exists())
    ant.mkdir(dir:"${dist}/classes")
```

Gant and AntBuilder

Although `AntBuilder` may be considered a mini DSL for building with Ant, the fact remains that you need to be a Groovy programmer to develop with it. We are still playing with some boilerplate to make use of the DSL, and there are obvious limitations in the lack of targets and dependencies. Gant takes the obvious next step. Internally, Gant extends from `AntBuilder`, but adds a bunch of capabilities to it. Gant also comes with its own runtime, so we do not have to construct an `AntBuilder` object in our script. Gant allows us to write simply write markup code directly in the script, with no precursors. Here we have a Gant script that is a complete functional replacement for our first Ant script. Just like the original, there are `init`, `run`, and `compile` targets.

```
dist = "./dist"

ant.taskdef ( name : 'groovyc' ,
              classname : 'org.codehaus.groovy.ant.Groovyc' )

target (init: "Create directories") {
    mkdir(dir:"${dist}/classes")
}

target(compile:"Compile groovy code") {
    depends(init)
    groovyc( srcdir: ".",
              destdir: "${dist}/classes",
```

```

        includes:"Hello.groovy")
    }

target(run:"Run in JVM") {
    depends(compile)
    java(classname:"Hello") {
        classpath {
            pathelement
            path: "${path-to}groovy-all-${version}.jar"
            pathelement
            path: "${dist}/classes"
        }
    }
}

setDefaultTarget(run)

```

Gant hides the fact that we are using an `AntBuilder` class. Actually, we are using Gant's own builder `GantBuilder`, which extends `AntBuilder`.

On startup, Gant will load the supplied `build.gant` script, and then run it. Prior to running the script, it sets up some additional properties in the binding. We can see some of these in use in the previous example.

The call to `ant.taskdef()` is a call to a `GantBuilder` instance that has been stored in the binding.

Calls to `target()` are in fact to a closure property target that has been stored in the binding. The `target` closure accepts a `Map` and a `Closure` as its parameters. The `Map` defines the name and description of a target, while the closure is stored for execution later when the `target` is invoked. All that `target` does when called is to store a closure with a target name as a key, along with a description.

The `setDefaultTarget(run)` call is also a closure stored in the binding, that sets the default target to `run`, which is the target set up by the previous target call.

If we were to pause Gant immediately after running the `build.gant` script, we would see that nothing more has happened other than setting up a new `groovyc` task in the builder. This is done through `ant.taskDef`, which stores the closures for three targets, namely `init`, `compile`, and `run`, and sets the default target to `run`.

Now Gant can start executing the build logic. It will start by calling the closure stored for the default target by first setting its delegate to the `GantBuilder` instance. This closure will now behave just like `AntBuilder` markup, with the one additional piece of logic being provided by `depends()`. The `depends()` method is implemented through an overloaded `invokeMethod` in the `GantBuilder` metaclass.

The depends() call will invoke the stored closure for the dependent target before continuing with the rest of the closure logic. In this way, by invoking the first target closure, Gant will chain through all of the dependent target closures, in sequence. Every time a target closure is invoked, the GantBuilder instance will be set as its delegate, so the result is the same as invoking one large AntBuilder closure containing all of the build logic.



If we were to dig into the code for Gant, we would find that some of its implementation classes, including GantBuilder, are not written in Groovy, but in Java. There is no particular reason for this that I can tell, other than that Gant was first written in 2006 and perhaps at the time the author felt that he had greater flexibility to work with the Groovy MOP in Java. In fact, the Java code does not do anything different from what we have already covered in Groovy. The two languages are interchangeable, anyway.

Gant is neatly packaged with a runtime command line **gant**. Just like Ant expects a default build script of `build.xml`, **gant** will first look for a default `build.gant`. By convention, **gant** scripts take the extension `.gant`, which helps to define them as standalone DSL scripts. Looking at the previous script, it would be hard to tell to the uninitiated that this is actually Groovy code. A configuration engineer can learn **gant** as a mini-language in its own right, and work effectively with it without knowing anything about Groovy. Of course, the Groovy aspect and the fact that we can augment **gant** scripts with full-blown scripting means that it is a superb power tool for the experienced configuration engineer or developer.

We run **gant** from the command line, as you would expect. Assuming that the above script is saved as `build.gant`, all we have to do is:

```
$ gant
[mkdir] Created dir: /Users/fdearle/ /gant/dist/classes
[groovyc] Compiling 1 source file to /Users/fdearle/gant/dist/classes
Hello, Groovy World!
$
```

If we need to know what targets are available, then:

```
$ gant -p
```

```
compile  Compile groovy code
init      Create directories
run       Run in JVM
```

Default target is run.

\$

Gant does not replace Ant, but builds on its capabilities. I personally am loath to use regular Ant XML in my builds scripts anymore. Gant provides so much more power and flexibility.

ATDD, BDD with GSpec, EasyB, and Spock

Test Driven Development (TDD) has become an essential capability for software developers over the past decade. TDD can mean different things to different organizations. It can mean the adoption of a full-blown **test first** style of coding, where unit tests are written before any functional code. It could just mean that you write extensive unit tests for every piece of functional code in the system. It may or may not mean the use of continuous integration builds that run a battery of unit tests after each code check in. Whatever TDD means to your organization, the chances are that flavors of xUnit test frameworks, including JUnit, WEBUnit, and HTTPUnit have been essential tools in your software developer's arsenal for some considerable time now.

The problem with xUnit-style testing is that these are all tools that are designed by programmers, for programmers. Your QA staff might be familiar with running xUnit tests and reporting on problems that they encounter, but they are less likely to be involved in originating the code tests with these frameworks. Unit tests tend to be written to test features at the class and method level rather than testing the intended behavior of the software.

Two alternate models of testing – **Acceptance Test Driven Development (ATDD)** and **Behavior Driven Development (BDD)** – have been advocated, primarily within the agile community, as a solution to this issue. The key to promoting ATDD and BDD is the creation of tools that allow all of the stakeholders, including developers, business analysts, and QA, to be able to use a common tool with a common language. Testing begins with a specification of the intended behavior for the software, which will be entered into the tool. This implies being able to develop a specification in a language that can be written and understood by of all the stakeholders, and not just the developers.

The common language of ATDD and BDD revolves around defining behavior in user-centric terms. Whether we are writing acceptance tests or defining behavior, we end up using terms such as **GIVEN**, **WHEN**, and **THEN** as follows:

- **GIVEN** a precondition
- **WHEN** certain actors perform certain actions
- **THEN** we expect a predetermined result

Of course, this type of terminology is ripe for using a mini-language or DSL to define the tests or behavior. And this is exactly what is happening. There are a plethora of BDD style testing frameworks out there already, the most notable being **RSpec**, a BDD-style framework implemented by using Ruby's dynamic features. At the time of writing this book, there are several BDD-style DSL frameworks for Groovy, including **GSpec**, **EasyB**, and **Spock**. We will take a brief look at these frameworks here.

GSpec

GSpec is the oldest BDD project for Groovy, having made its first appearance in 2007. The lack of current development by the project founder, and the fact that it is still defined as being in Alpha, would lead us to believe that Spock is superseding it. Below is an example of a GSpec specification. It's still worth taking a look at this from our perspective, to see what Groovy features are being used to implement the DSL. Here is segment of a specification that can be found at <http://groovy.codehaus.org/Using+GSpec+with+Groovy>. This script defines a specification that defines a part of the desired behavior for a `FixedSizeStack` class. Notice how the DSL is structured to give a pseudo English style to it.

```
import com.craig.gspec.GSpecBuilderRunner

def the = new GSpecBuilderRunner()

the.context('A non-empty stack') {
    initially {
        the.stack = new FixedStack()
        ('a'..'c').each { x -> the.stack.push x }
        the.stack.should_not_be_empty
    }

    specify('should return the top item when sent #peek') {
        the.stack.peek().should_equal 'c'
    }
}
```

```
specify('should NOT remove the top item when sent #peek') {
    the.stack.peek().should_equal 'c'
    the.stack.peek().should_equal 'c'
}

specify('should be unchanged when sent #push then #pop') {
    the.stack.push 'Anything'
    the.stack.pop()
    the.stack.peek().should_equal 'c'
}

specify('should return the top item when sent #pop') {
    the.stack.pop().should_equal 'c'
    the.stack.push 'c' // put it back the way it was
}

specify('should remove the top item when sent #pop') {
    the.stack.pop().should_equal 'c'
    the.stack.pop().should_equal 'b'
}

specify('should add to the top when sent #push') {
    the.stack.push 'd'
    the.stack.peek().should_equal 'd'
}
}

the.context('An empty stack') {
    initially {
        the.stack = new FixedStack()
        the.stack.should_be_empty
    }

    specify('should no longer be empty after #push') {
        the.stack.push 'anything'
        the.stack.should_not_be_empty
    }
}

the.context('A stack with one item') {
    initially {
        the.stack = new FixedStack()
        the.stack.push 'anything'
        the.stack.should_not_be_empty
    }
}
```

```
specify('should remain not empty after #peek') {
    the.stack.peek()
    the.stack.should_not_be_empty
}

specify('should become empty after #pop') {
    the.stack.pop()
    the.stack.should_be_empty
}
}

}
```

GSpec implements markup style syntax through the `GSpecBuilderRunner` class. Specifications are defined by invoking the pretended method `specify()` within the scope of a `context()` block. For each context, the GIVEN preconditions are set within an `initially()` block. Within each `specify()` block, we describe the preconditions that need to be met.

If we want to make an assertion about a condition, we can set a property on the `GSpecBuilderRunner` object. So in each `initially()` block, we set a `stack` property to be a new `FixedStack` object that will be the object acted upon in the specification. Any properties that are added to the Builder are wrapped with some new behavior supplied by GSpec, which adds some pretended property accessors starting with `should_be_` and `should_not_be`. These accessors are equivalent to assertions on the object. So `the.stack.should_not_be_empty` is equivalent to asserting `the.stack.isEmpty() == false`.

GSpec also adds some pretended methods to properties that are added to the builder. The methods `should_equal` and `should_not_equal` also act as assertions and will assert that the parameter passed is either `true` or `false`. Between the two GSpec supplies, a dialect of assertions exists that is easier to read than traditional assertions. For a `specify` block to be complete, it must contain at least one `should_style` assertion, although the DSL makes no attempt to ensure that this happens.

GSpec makes no attempt to hide the fact that it is using a Builder. From the previous DSL example we have seen techniques that could be used to conceal the Builder instance. However, the mechanism that GSpec uses to add the `should_be` and `should_equal` style assertions requires access to the builder instance.



`GSpecBuilderRunner` extends the `GSpecBuilder` class. `GSpecBuilder` has an overloaded `setProperty()` method that wraps every property set in the Builder instance as a `GSpecDelegate` object. It is the `GSpecDelegate` class that injects all of the `should_be` and `should_not_be` property accessors, and the `should_equal` and `should_not_equal` methods. As a result, a call to the stack `should_be_empty` translates to a `GSpecDelegate.getProperty()`. Instead of being a property lookup for `should_be_empty`, the `GSpecDelegate` recognizes this as an assertion, and asserts that `wrappedObject.isEmpty()` is true.

EasyB

EasyB is becoming one of the most popular BDD frameworks used by the Groovy community. The dialect that it provides is similar to GSpec in a lot of ways. Below is an example specification from the EasyB website. You can download EasyB from <http://easyb.org/download.html>. EasyB provides a BDD style dialect through the `description`, `before`, `it`, and, and `and` after keywords. The main specification keyword is `it`, which encourages you to follow a descriptive statement that must be fulfilled: for example, `it "should dequeue item just enqueued"`.

```
package org.easybbdd.specification.queue

import org.easybbdd.Queue

description "This is how a Queue must work"

before "initialize the queue for each spec", {
    queue = new Queue()
}

it "should dequeue item just enqueued", {
    queue.enqueue(2)
    queue.dequeue().shouldBe(2)
}

it "should throw an exception when null is enqueued", {
    ensureThrows(RuntimeException.class) {
        queue.enqueue(null)
    }
}
```

```
    }
}

it "should dequeue items in same order enqueued", {
    [1..5].each {val ->
        queue.enqueue(val)
    }
    [1..5].each {val ->
        queue.dequeue().shouldBe(val)
    }
}
```

The `it` specification keyword is followed by a closure block containing the specification assertions. Within that closure we can assert the state of any object by calling assertion methods on it. EasyB provides an extended list of assertion methods that can be called on an object within the closure. These are just a few of them: `shouldBe()`, `shouldBeLessThan()`, `shouldBeEqualTo()`, `isA()`, `isNotA()`, `shouldNotBeA()`.

GSpec and EasyB are quite similar in structure, but they use different mechanisms to implement their DSL-style keywords and assertions. EasyB keywords are implemented by adding named closures to the Script binding, as follows:

```
binding.before = {description, closure = {} ->
    ...
}
```

By setting a named closure as a variable in the binding, it becomes available as a callable function in the EasyB script. Later on, we will use this feature ourselves to build a Rewards DSL.



Within the `it` closure, EasyB applies the additional assertion methods `shouldBe()`, `shouldNotBe()`, and others through a `category` class. The `BehaviorCategory` class implements all of the assertion methods as follows:

```
class BehaviorCategory {
    static void shouldBeGreaterThan(Object self, value)
    {
    ...
    }
}
```

The implementation of the `it` keyword uses this `BehaviorCategory` class when calling the closure body. The `BehaviorCategory` is therefore applied to the entire closure code that follows the `it` keyword.

```
binding.it = { description, closure = {} ->
    use(BehaviorCategory) {
        closure()
    }
}
```

Placing a named closure in the binding gives us options for how we treat the closure body in the DSL. EasyB just applies a category to the closure block, as shown in the above code snippet. However, we can also set the closure delegate to a Builder if we prefer to have markup-style syntax in our closure blocks.

Spock

To finish off our look at BDD frameworks, we will look at Spock. Spock uses similar principles to GSpec and EasyB, as the next example from the Spock framework example code shows. This code snippet is from the Spock Stack example. You can find the full example, downloads, and documentation for Spock, hosted on Google Code at <http://code.google.com/p/spock/>.

```
import org.junit.runner.RunWith
import spock.lang.*

@Speck(java.util.Stack)
@RunWith(Sputnik)
class EmptyStack {
    def stack = new Stack()

    def "size"() {
        expect: stack.size() == 0
    }

    def "pop"() {
        when: stack.pop()
        then: thrown(EmptyStackException)
    }

    def "peek"() {
        when: stack.peek()
        then: thrown(EmptyStackException)
    }
}
```

```
def "push"() {
    when:
    stack.push("elem")

    then:
    stack.size() == 1
    stack.peek() == "elem"
}
}
```

Spock tries to hide as much boilerplate from the user as possible, by using annotations. The `@Speck` annotation in the preceding code snippet tells Spock to expand the `EmptyStack` class into a full-blown Spock test for the `java.util.Stack` class. There is no need to instantiate a Builder in GSpec. The `@Sputnik` tells Spock to use its own JUnit runner to run the specification.

A Spock specification consists of several parts, as follows:

```
@Speck
@RunWith(Sputnik)
class MyFirstSpecification {
    // fields
    // fixture methods
    // feature methods
}
```

- Fields: Class instance fields can be used to set up objects to be used by the fixture methods.
- Fixture methods: These include `setup()`, `cleanup()`, `setupSpeck()`, and `cleanupSpeck()`. These are called before and after each feature method. Together with fields, these make up the GIVEN preconditions of a behavior test.
- Feature methods: These describe the features of the specification to be tested. Feature methods are named with string literals so that they can be highly descriptive of the feature to be tested. For instance, the `pop` feature method can be named "pop from an empty stack should raise an exception".

Blocks

A block is a segment of code that starts with a label, for example, `when:`, and continues until the end of the feature method or the start of another block. Spock supports a number of different block types.

- `setup`: which also has an alias of `given`: and is used to denote the preconditions of a feature.
- `when`: `then`: which describe the stimulus and expected response of a feature.
- `expect` : is used when the stimulus and response can be combined into a single expression; for example, `expect : "ABC".toLowerCase() == "abc"`.
- `where` : allows multiple inputs to be fed into the feature method. For example, we can declare that `toLowerCase` should lower the case of all letters regardless of whether we start with all uppercase, mixed case, or all lowercase.

```
expect:
    a.toLowerCase() == b
where:
    a << ["ABC" , "Abc" , "abc"]
    b << ["abc" , "abc" , "abc" ]
```

The block label code is an unusual Groovy code. You may be aware of labels in Java and Groovy as a means of breaking or continuing from loops:

```
outer:
    for( i=0; i<10; i++ ){
        for( j=10; j>0; j-- ){
            if( j == 5 ) {
                break outer;           // exit entire loop
            }
        }
    }

outer:
    for( i=0; i<10; i++ ){
        for( j=10; j>0; j-- ) {
            if( j== 5 ) {
                continue outer;   // next iteration of i
            }
        }
    }
```

Attaching a label such as `setup : and expects :` to any statement is valid Groovy code, but in the absence of a loop it is redundant, as we cannot associate a `break ;` or `continue ;` statement to navigate to it. Neither Java nor Groovy has a `goto ;` statement. So how is Spock making use of these blocks?



The answer is that Spock uses an advanced feature of Groovy to provide its own implementation of the label-handling code. The **Abstract Syntax Tree (AST)** Transformation feature introduced in Groovy 1.6 allows Spock to transform the AST before it gets compiled into bytecode. In this respect, Spock is not a pure Groovy-based DSL because it is compiling its own code blocks behind the scenes. However, labeling blocks of code in this way is a clever subterfuge because it uses valid Groovy syntax to decorate the code in a way that makes the intention very clear to the reader.

BDD DSL style

GSpec, EasyB, and Spock all attempt to achieve the same objective. The ultimate goal is to wrap xUnit assertions in a user-friendly DSL that allows specifications to be written in a more user-friendly way. GSpec unashamedly wears its heart on its sleeve as a Groovy Builder. EasyB implements keywords through binding variables and categories. Spock subverts the Groovy syntax with AST transformations and lessens boilerplate with annotations.

For my money, EasyB provides the most usable syntax out of the three. GSpec forces the introduction of a Builder instance, which is an unnecessary boilerplate that may confuse users who are trying to read the specifications. Although Spock's clever manipulation of the AST is impressive, the resulting DSL is still structured in quite a programmatic idiom. EasyB on the other hand scans almost like a structured English representation of the specification.

Summary

In this chapter, we looked at some existing Groovy DSLs that are in current use and are free to download. GORM implements a full persistence layer over Hibernate that layers over standard Groovy classes. GORM allows us to decorate a regular POGO with settings for applying the most common associations and relationships that we can expect in our object models.

Much of what GORM provides in terms of querying via dynamic finders requires a Groovy-knowledgeable developer to appreciate and use them. However, the basic object modeling semantics provided through the `belongsTo`, `hasMany`, and other persistence settings could be used quite readily by a data modeling architect who has little or no knowledge of the Groovy language. The domain classes can be viewed as an independent model specification language, which has the advantage of being immediately usable by Groovy developers responsible for other parts of the system.

In the same vein, Gant provides a GroovyMarkup style syntax to interface with Ant, as an alternative to XML. Gant can be learned as a mini DSL for Ant or it can be augmented with Groovy program logic that makes it significantly more powerful than plain Ant XML build files.

GSpec, EasyB, and Spock all attempt to bring BDD style specification-based testing to the Java/Groovy platform. Each provides a means to write specifications by using GIVEN/WHEN/THEN style semantics that are easier to interpret than regular xUnit style testing frameworks. The pseudo-English style syntax of these DSLs should mean that the specifications can be understood by business stakeholders even if they still need to be coded by a Groovy-proficient developer.

Most importantly, in this chapter, we have seen how all of these projects exploit different Groovy features in order to implement DSL style structures and syntax.

7

Building a Builder

Builders are a powerful feature of Groovy. The Groovy libraries contain an expanding set of Builders for everything from XML and HTML markup to managing systems via JMX. Even so you will always come across circumstances where the semantics of a builder would be useful in your own application.

We've seen how to build a rudimentary builder by using the Groovy MOP and pretended methods in Chapter 5. Thankfully, the Groovy libraries provide us with easier means of developing our own builders. In this chapter, we will look at some of the ways in which we can use Groovy and the MOP to create our own builder classes.

- To begin with, we will recap the Groovy features that enable the Groovy builder pattern in order to understand how they work.
- We will look at how to build a rudimentary builder with features from the Groovy MOP.
- We will implement our own database seed data Builder by using two of the builder support classes provided in Groovy: `BuilderSupport` and `FactoryBuilderSupport`.

Builder code structure

The real beauty of Groovy's builder paradigm is the way in which it maps the naturally nested block structure of the language to the construction process. The process of defining parent-child relationships between objects through nested code blocks is well-established through other markup languages, such as XML and HTML.

The transition from writing XML or HTML to the GroovyMarkup equivalent is an easy one. To make use of a builder, we don't need to have any intimate understanding of the Groovy MOP or of how the builder is implemented. We just need to know how to write the builder code so that it conforms to the correct language semantics. The code structure of the builder pattern relies on just a few Groovy language features.

- Closure method calls: The distinctively nested block structure in the builder pattern is facilitated by Groovy's special handling of closures when they are passed as method parameters. This allows the closure block to be declared inline after the other method call parameters.
- Closure method resolution: When a method is invoked within the body of a closure and that method does not exist in the closure instance, Groovy uses a resolve strategy to determine which object (if any) should be tried to locate that method.
- Pretended methods: The Groovy MOP allows us to respond to method calls that do not exist in a class—in other words to "pretend" that these methods exist.
- Named parameters: When we pass a `map` parameter to a method, we can declare the individual map elements alongside the other method parameters, giving the effect of a named parameter list.
- Closure delegate: Changing the delegate of a closure allows another class to handle its method calls. When we change the delegate to a `builder` class, this allows the builder to orchestrate how the method calls are handled.

Closure method calls

When we declare a Groovy method that accepts a closure as its last parameter, Groovy allows us to define the body of the inline closure immediately after the method call containing the other parameters. A method call followed by an inline closure block has all the appearance of being a named block of code. It's when we nest these method calls within each other that we get the distinctive builder-style code blocks.

This style of coding is not unique to builders. We can nest other method calls in the same way. In the following example, we have three methods defined within a script: `method1()`, `method2()`, and `method3()`. Nesting calls to these methods gives us some code that is very similar to a builder block, but is not actually a builder block. The cool thing about the builder pattern is that it uses this existing feature from the language and turns it into a whole new coding paradigm.

The screenshot shows the GroovyConsole interface. At the top, there's a toolbar with various icons. Below it is a code editor window containing Groovy script. The script defines three methods: method1, method2, and method3, and demonstrates closures with nested scopes. The output pane below the code editor shows the results of running the script. The message "Execution complete. Result was null." is displayed at the bottom.

```

def method1 (params, Closure closure) {
    println "method1: ${params}"
    closure.call()
}

def method2 (Closure closure) {
    println "method2:"
    closure.call()
}

def method3 (params) {
    println "method3: ${params}"
}

method1(param: "one") {
    method2 {
        method3 "hello"
    }
    method1( 123 ) {
        method1 ( "nested" ) {
            method3 10
        }
    }
}

method1: [param:one]
method2:
method3: hello
method1: 123
method1: nested
method3: 10

```

Execution complete. Result was null. 4

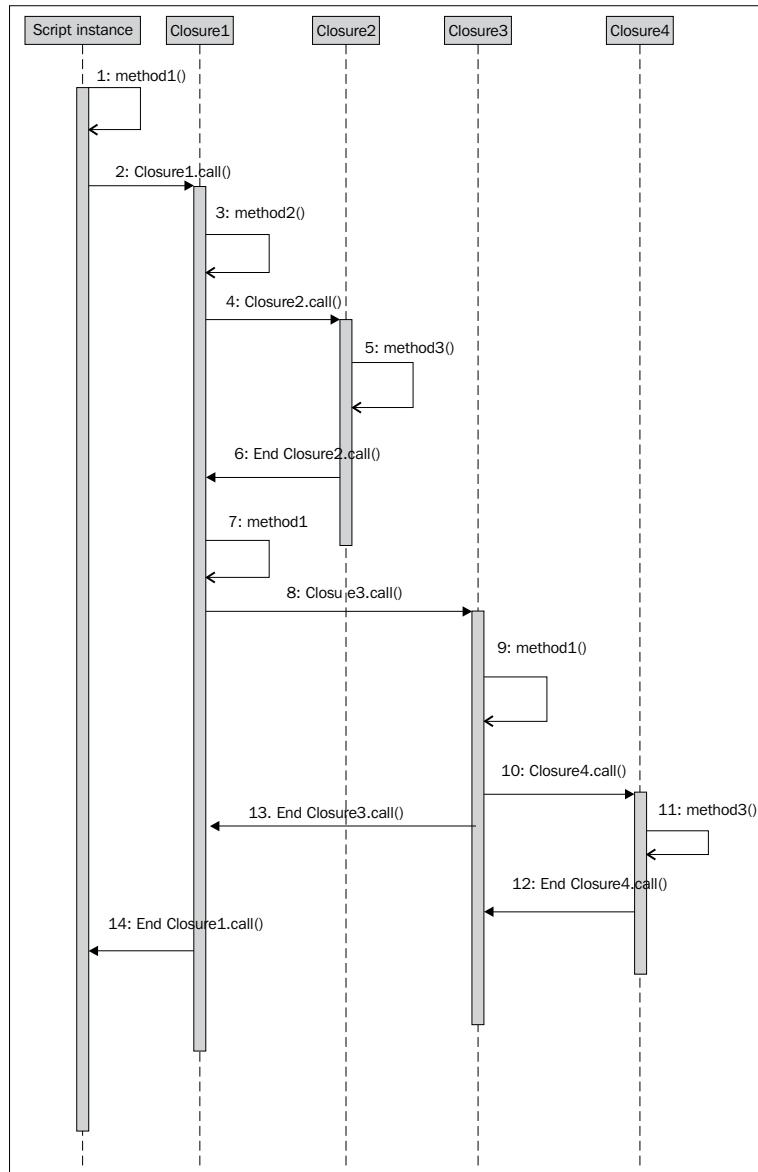
The success of building our own builder class by using the Groovy MOP depends largely on understanding the sequence in which these methods get called. The output gives us an idea of what might be happening and what the true sequence of events is. Let's decorate the code a little to show what is happening. The comments show what scope we are running in.

```

// Script scope
method1(param: "one") { // Closure1 scope
    method2 { // Closure2 scope
        method3 "hello"
    } // End Closure2
    method1( 123 ) { // Closure3 scope
        method1 ( "nested" ) { // Closure4 scope
            method3 10
        } // End Closure4
    } // End Closure3
} // End Closure1

```

The main block of code runs within the scope of the script. Each inline closure is in fact an anonymous instance of a `Closure` object. For the purpose of this exercise we will name these instances `Closure1` to `Closure4`. The first call to `method1()` occurs in the outer scope of the script so we would expect this method to be passed to the script instance. The subsequent method calls all happen within the scope of one or other of the anonymous closure instances, so we expect these methods to be invoked on the individual closure instances. The following sequence diagram illustrates this:



Resolve Strategy: OWNER_FIRST

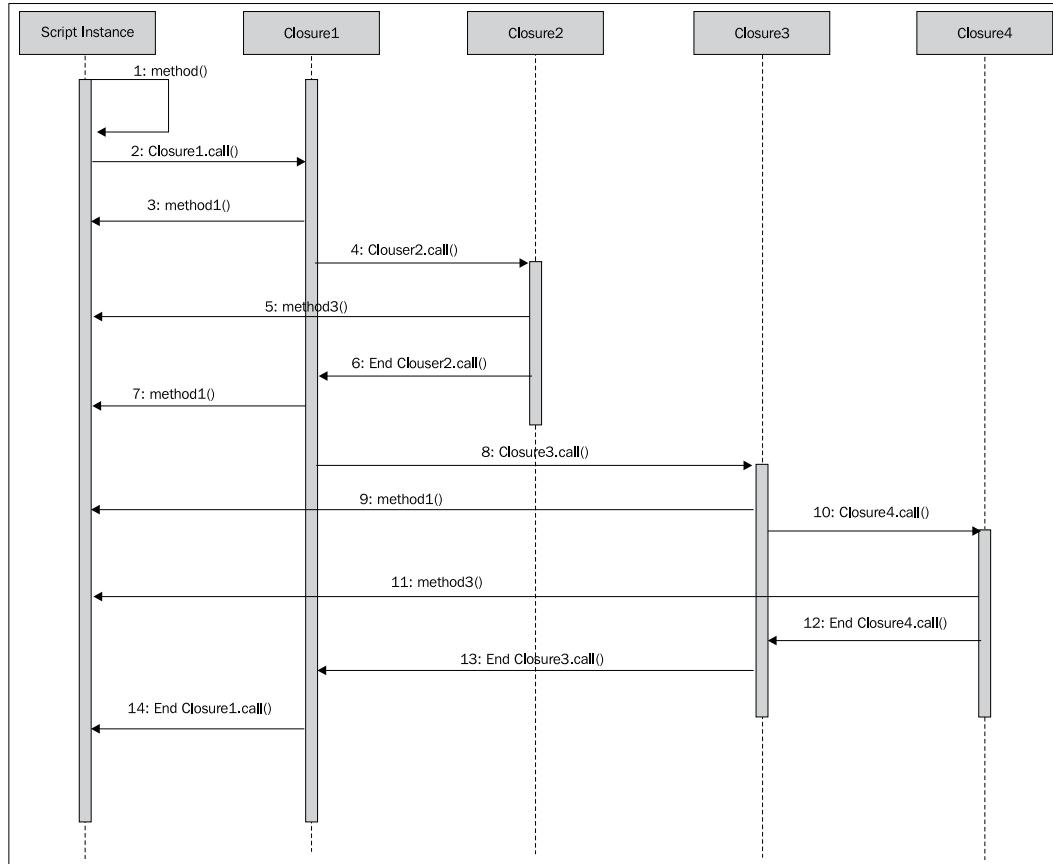
The one problem with the previous diagram is that we know that the closure instances don't implement the `method1()` to `method3()` methods. So this sequence diagram shows what methods are initially called, but it does not show what methods actually get called. When Groovy makes a call to a method on a closure, it does not always expect it to be present.

If a method is not present in the closure itself, Groovy will try to find it by looking in the owner of the closure, or its delegate, or both. The order in which this occurs is called the **resolve strategy** of the closure. The default resolve strategy is `OWNER_FIRST`, which means that the owner of the closure will be queried first for the method, followed by the delegate. If the owner of the closure happens to be another closure, then the resolve strategy will continue its search for a match in the owner of the owner and so on, until a match is found or the outer scope is reached.



The resolve strategy can be changed for a closure by calling `Closure.setResolveStrategy`. We can change the resolve strategy to any of the following self-explanatory strategies: `OWNER_FIRST`, `OWNER_ONLY`, `DELEGATE_FIRST`, `DELEGATE_ONLY`, and `NONE`.

Although the preceding sequence diagram reflects the first port of call for each method invocation, what in fact happens is that the resolve strategy kicks in and the method calls will percolate out through the closure instances. A match will eventually be found in the script instance, which is the only place where the actual methods exist. Therefore, the actual calling sequence is better represented as follows:



The insight that Groovy designers had when designing the builder pattern was that this natural nesting of closures could be used to map to any construction process that involved a parent-child type of relationship. Even without using a `builder` class, we can nest closures' method calls to create a pseudo builder. In the next example, we declare three methods that we can use to construct a rudimentary tree structure out of map objects.

The `root()` method creates the initial tree map and inserts a root element into it. We can nest as many levels deep as we like with the `node()` method, as it will remember its parent node and add sub nodes to it. The `leaf()` method is the only one to take a value and it does not expect to be passed a closure, as it will create the leaf elements in the tree structure.

```
def current
def root (Closure closure) {
    def tree = [:]
    def root = [:]
    tree["root"] = root
    def parent = current
    current = root
    closure.call()
    current = parent
    return tree
}

def node (key, Closure closure) {
    def node = [:]
    current[key] = node
    def parent = current
    current = node
    closure.call()
    current = parent
}

def leaf (key, value ) {
    current[key] = value
}

// pseudo builder code
def tree = root {
    node ("sub-tree-1") {
        leaf "leaf-1", "leaf object 1"
    }
    node ("sub-tree-2"){
        node ("node-1"){
            leaf "leaf-2", "leaf object 2"
        }
    }
}
```

```
assert tree == [
    root: [
        "sub-tree-1": [
            "leaf-1": "leaf object 1"
        ],
        "sub-tree-2": [
            "node-1": [
                "leaf-2": "leaf object 2"
            ]
        ]
    ]
]
```

Pretended methods

Many Groovy builders rely on the ability to describe arbitrarily-named elements. When we make use of markup builders to generate XML, we need to be able to insert whatever tag names are required to conform to the schema that we are using. Given that elements are created in method calls, we also need to be able to make arbitrarily- named method calls during the markup process.

With Groovy, we can respond to methods that don't exist as concrete methods of a class. The term we use for this type of methods is **pretended methods**. Groovy provides two means for implementing a pretended method.

invokeMethod

The `PoorMansTagBuilder` class that we covered in Chapter 5 uses `invokeMethod` as a means of pretending methods. The `PoorMansTagBuilder` class works by handling all method calls to the builder, and invoking the closure argument manually. With `invokeMethod`, we can respond appropriately to any arbitrary method call. In this case, we output the appropriate XML tags.

```
class PoorMansTagBuilder {
    int indent = 0
    Object invokeMethod(String name, Object args) {
        indent.times {print "    "}
        println "<${name}>"
        indent++
        args[0].delegate = this // Change delegate to the builder
        args[0].call()
```

```
    indent--
    indent.times {print "      "}
    println "</${name}>"
}
}
```

This is a simple case that we are using just to illustrate the mechanism. Although the technique works for simple cases, extending it to implement a more complete tag builder would rapidly result in complex and hard-to-maintain code.

methodMissing

Since Groovy 1.5, an alternative to `invokeMethod` was provided. The `methodMissing` mechanism differs slightly from `invokeMethod`, as it is only called when a method call fails to be dispatched to any concrete method. To update the `PoorMansTagBuilder` for using `methodMissing` instead of `invokeMethod`, all we need to do is replace the method name that we declare.

```
class PoorMansTagBuilder {
    int indent = 0
    def methodMissing(String name, args) {
        indent.times {print "      "}
        println "<${name}>"
        indent++
        args[0].delegate = this // Change delegate to the builder
        args[0].call()
        indent--
        indent.times {print "      "}
        println "</${name}>"
    }
}

def builder = new PoorMansTagBuilder ()

builder.root {
    level1{
        level2 {
    }
}
}
```

Closure delegate

Earlier, we looked at how to code a pseudo builder using methods declared within a script. The resolve strategy in that example passed method calls in the nested closure up to the owner of the closure. The `builder` block in the previous example is also in the scope of a script. Let's decorate it as we did before, to identify the various anonymous closure instances.

```
// method root() called on PoorMansTagBuilder
builder.root { // Closure1
    // method level1 called on Closure1 instance
    level1{ // Closure2
        // method level2 called on Closure2 instance
        level2 { // Closure3
        }
    }
}
```

The first method call to `root()` is made against the `builder` instance, so it will be handled directly by `PoorMansTagBuilder.methodMissing()`. Nested method calls will first be dispatched to the enclosing closure. The `level1()` and `level2()` methods won't be found in the closure instances, so we would normally expect the resolve strategy to dispatch these methods up the chain of owners until a method is found. This normal dispatch chain would end up at the script instance, so these methods would cause a `MethodMissingException` to be thrown.

The secret of how this works is in the handling of the delegate for closure instances. The builder block starts with a direct method call onto the `builder` instance, `builder.root()`. Anonymous closure, `Closure1`, is passed as a parameter. The call to `root()` will fail and fall through to `methodMissing`. In this simple example, `arg[0]` is always the closure because we are not processing parameters on our tags. A more sophisticated version would need to scan the parameters for the closure instance.

At this point we have access to the closure, so we can set its delegate to the `builder` instance. Now when the `level1()` and `level2()` calls are encountered, the resolve strategy will try the owner first and then try the delegate as follows:

- The `level1()` call will not be resolved in `Closure1`. It won't be found in the owner of `Closure1`, which is the script, but it will be resolved in the delegate, which is the `builder` instance. `PoorMansTagBuilder.methodMissing` will field the method and also set the delegate for the anonymous closure, `Closure2`.
- The `level2()` call happens in the scope of `Closure2` but will not be resolved there. First its owner, `Closure1`, will be tried, and then its delegate, which once again is the `builder` instance.

BuilderSupport

Under the hood, all of Groovy's own Builders are implemented by using the `invokeMethod` or `methodMissing` and delegate techniques that we have described above. We could choose to start creating our own builder classes by using these features alone. Perhaps the biggest challenge when creating a builder with these features alone is that the MOP concepts of pretended methods and delegate handling don't fit well with the task at hand – namely the construction of complex objects. It would be nice to have APIs that reflected the task at hand in a better way.

Thankfully, the complexities of managing `invokeMethod` or `methodMissing` calls and figuring out who the delegate should be are encapsulated into the builder support classes provided in the Groovy packages. The most basic support class is `groovy.util.BuilderSupport`.

BuilderSupport hook methods

`BuilderSupport` provides an interface to the building process that nicely mirrors the node-based construction process on which most builder classes are based. Instead of overriding `invokeMethod` as in the initial example, we override the node construction methods provided by `BuilderSupport`.

These methods provide the hooks such that instead of coding at the MOP level with pretended method invocations and delegates, we can code with respect to the node construction process. The important hook methods that we need to know about are listed here. These methods encapsulate the building process out into node creation style events that make far more sense from an object construction point of view. Then we never need to worry about pretended methods or delegates again.

- **`createNode(Object name)`**
Called by `BuilderSupport` whenever a pretended method is encountered. The name parameter contains the name of the pretended call. The responsibility of the hook is to return a new object of a type appropriate to the method call. The specific hook to be called depends on what parameters are passed to the method.
- **`nodeCompleted(Object parent, Object node)`**
Called after all of the children of a `node` have been created.
- **`setParent(Object parent, Object child)`**
Called after `createNode` for each child node, in order to allow any parent-child relationships to be established.



BuilderSupport takes care of all of the nitty-gritty of handling pretended methods for us. Had our PoorMansTagBuilder worked only for parameter-less tags, BuilderSupport would have detected which type of call is being made and called the appropriate `createNode` for us. The `setParent` method is only called if a parent node exists.

How this all hangs together is best illustrated by way of an example. So let's start by creating a really dumb builder that just logs these methods as they are encountered. This will give us a feel for the sequence in which the methods are called.

```
class LogBuilder extends BuilderSupport {  
    def indent = 0  
    def createNode(name) {  
        indent.times {print "  "}  
        println "createNode(${name})"  
        indent++  
        return name  
    }  
    def createNode(name, value){  
        indent.times {print "  "}  
        println "createNode(${name}, ${value})"  
        indent++  
        return name  
    }  
    def createNode(name, Map attributes){  
        indent.times {print "  "}  
        println "createNode(${name}, ${attributes})"  
        indent++  
        return name  
    }  
    def createNode(name, Map attributes, value){  
        indent.times {print "  "}  
        println "createNode(${name}, ${attributes}, ${value})"  
        indent++  
        return name  
    }  
    void setParent(parent, child){  
        indent.times {print "  "}  
        println "setParent(${parent}, ${child})"  
    }  
    void nodeCompleted(parent, node) {  
        indent--  
        indent.times {print "  "}  
        println "nodeCompleted(${parent}, ${node})"  
    }  
}
```

To use this builder, all that we need to do is to construct one and start writing some markup with it. Here we have some markup for building customer records, but as this builder does not care what the method tags are, we could write whatever markup we please.

```
def builder = new LogBuilder()

def customers = builder.customers {
    customer{
        id(1001)
        name(firstName:"Fred", surname:"Flintstone")
        address("billing", street:"1 Rock Road", city:"Bedrock")
        address("shipping", street:"1 Rock Road", city:"Bedrock")
    }
}
```

If we run this script, we will generate the following output. We can see from the output exactly what the sequence of calling is, and what parameters are being passed. We've used this simple example for illustrating how the `BuilderSupport` class works, but it is actually a useful debugging tool in general for using with any builder that's not behaving as expected. By replacing any existing builder instance in your code with a `LogBuilder`, it will output the construction sequence for you, which may identify the problem.

```
createNode(customers)
createNode(customer)
setParent(customers, customer)
createNode(id, 1001)
setParent(customer, id)
nodeCompleted(customer, id)
createNode(name, [firstName:Fred, surname:Flintstone])
setParent(customer, name)
nodeCompleted(customer, name)
createNode(address, [street:1 Rock Road, city:Bedrock], billing)
setParent(customer, address)
nodeCompleted(customer, address)
createNode(address, [street:1 Rock Road, city:Bedrock], shipping)
setParent(customer, address)
nodeCompleted(customer, address)
nodeCompleted(customers, customer)
nodeCompleted(null, customers)
```

From this output, we can trace the sequence in which the hooks are called. Nodes are created from the top down. The `createNode` hook for the parent is called first. The `createNode` hook for a child is called next, and `setParent` is called for each individual child after both the parent and the child have been created. The `nodeCompleted` hook is called only after all of the children have been created and their parent-child relations set.

The default implementation of `BuilderSupport` manages the current node cursor by itself. Two additional hooks to consider are:

- `setCurrent(Object current)`
- `Object getCurrent()`

Certain builder implementations might want to manage the notion of a "current" node object in order to maintain a cursor on the construction process. If so, both of these hooks will need to be implemented.

Now that we understand the building mechanism, it is a trivial matter to change our `LogBuilder` script to create some actual markup. Here, with a few modifications, we can turn our script into `PoorMansTagBuilder 2.0`.

```
class PoorMansTagBuilder20 extends BuilderSupport {  
    def indent = 0  
  
    def createNode(name) {  
        indent.times {print "  "}  
        println "<${name}>"  
        indent++  
        return name  
    }  
    def createNode(name, value){  
        indent.times {print "  "}  
        println "<${name}>" + value  
        indent++  
        return name  
    }  
    def createNode(name, Map attributes){  
        indent.times {print "  "}  
        print "<${name} " "  
        print attributes.collect {  
            "${it.key}='${it.value}'"  
        }.join(' ')  
        println ">"  
        indent++  
        return name  
    }  
    def createNode(name, Map attributes, value){  
        indent.times {print "  "}  
        print "<${name} " "
```

```
print attributes.collect {
    "${it.key}='${it.value}'"
}.join(' ')
println ">" + value
indent++
return name
}
void setParent(parent, child) {
    // Don't care since we are just streaming to output
}
void nodeCompleted(parent, node) {
    indent--
    indent.times {print "  "}
    println "</${node}>"
}
}
```

Once again, this is a simple implementation of a tag builder. We are making no interpretation of the method tags that are being passed in, so for each `createNode` call all we do is output a `<TAG>` with parameters and attributes if necessary. The `setParent` call is not relevant to us because we are just streaming output to standard out. We will see in the next example where we need to implement this. Finally, the `nodeCompleted` call just closes the tag `</TAG>`.

Now we can apply this builder to the same `customers` markup script that we did before, as follows. The only change required is to instantiate a `PoorMansTagBuilder20` in place of the original builder class.

```
def builder = new PoorMansTagBuilder20()
def customers = builder.customers {
    customer {
        id(1001)
        name(firstName:"Fred", surname:"Flintstone")
        address("billing", street:"1 Rock Road", city:"Bedrock")
        address("shipping", street:"1 Rock Road", city:"Bedrock")
    }
}
```

Running this script will log some reasonably well-formed XML as follows:

```
<customers>
  <customer>
    <id>1001
    </id>
    <name firstName='Fred' surname='Flintstone' >
    </name>
    <address street='1 Rock Road' city='Bedrock' >billing
    </address>
    <address street='1 Rock Road' city='Bedrock' >shipping
    </address>
  </customer>
</customers>
```

As a markup builder, this falls well short of the features in the Groovy MarkupBuilder, but it does show just how easy it is to put together a quick builder to fit the need of the day. Now let's consider what we've learned, and look at building something a little more useful.

A database builder

Every application that includes a database needs some means of setting up seed, demo, or test data. I have worked on numerous enterprise applications during my career and invariably the management of different data sets becomes as much of an effort over time as the development of the application itself. In my own experience of working with **Independent Software Vendors (ISVs)**, whose applications need to be deployed on multiple customer sites with multiple versions, the problem becomes acute.

ISV companies often have competing needs for data sets. The sales organization needs a predictable data set for its demos to customers. The test department needs data sets that allow them to test specific features of the application. Project management requires specific seed data to be available, which is tailored to each customer site prior to installation. With all of these competing requirements, the IT department has a limited set of database instances available on which to install and test all of these configurations.

There are various ways of managing data sets. The most common is to maintain SQL scripts that take care of the database insertions. Building a simple database will require multiple insertions into a multitude of tables. The SQL script needs to be written in such a way as to maintain the integrity of foreign key references. It's not an easy thing to do, and requires intimate knowledge of the schema.

Suppose we are working with a Grails application. Take for example the one-to-many relationship we looked at in Chapter 6:

```
class Customer {  
    String firstName  
    String lastName  
    statichasMany = [invoices:Invoice]  
}  
  
class Invoice {  
    statichasMany = [orders:SalesOrder]  
}  
  
class SalesOrder {  
    String sku  
    int amount  
    Double price  
    static belongsTo = Invoice  
}
```

Grails has a migration plug-in that can be installed. The migration tool will execute a SQL update script to migrate our database between versions. To use the migrate tool to add some simple test data for the above classes, we need to know how GORM maps from the Groovy POGO objects to relational tables.

In Chapter 6, we also saw how these classes in fact map to five tables in the relational database. There are three main tables that represent the business objects (`customer`, `invoice`, and `sales_order`) and there are two mapping tables used to manage the foreign key mappings (`customer_invoice` and `invoice_sales_order`) that relate customers to invoices and invoices to sales orders.

To set up a simple test case with one customer, one invoice and three sales orders would require nine insertions across these tables. Apart from being error prone and difficult to maintain, the script will be incomprehensible to anyone who is not intimately acquainted with SQL. What starts out as a simple data input spec for a test case becomes a development task for a domain SQL expert who understands the GORM mapping model.

An alternative to this approach is to use the GORM APIs to build the test data. At least if we do this then we don't have to concern ourselves with the foreign key relationships between tables. The script below will set up our simple data set with one customer, one invoice, and three sales orders:

```
def fred = new Customer(firstName:"Fred", lastName:"Flintstone")  
  
fred.save()
```

```
def invoice = new Invoice()

invoice.addToOrders(new SalesOrder(sku:"productid01", amount:1,
price:1.00))
invoice.addToOrders(new SalesOrder(sku:"productid02", amount:3,
price:1.50))
invoice.addToOrders(new SalesOrder(sku:"productid03", amount:2,
price:5.00))

fred.addToInvoices(invoice)
```

This is somewhat better than the SQL script approach, but it does impose a procedural construction onto the data, where the test data is typically defined declaratively. While I've used GORM to illustrate my point here, the same issues will crop up whatever persistence mechanism we use.

Ideally, we want to be able to describe our data in a declarative style. The syntax of the data definition should as closely match the structure of the resulting data as possible. This is an ideal situation in which to use a builder to take care of construction. With a builder, it should be possible to create a declarative markup style script for building data sets. The builder can take care of the complexities of construction.

Let's first of all imagine how a builder for customers may look in use. We probably want to handle multiple customers, so a top-level `customers` method is useful. We could have multiple customer blocks nested below that. Nesting is a good way of depicting ownership in a one-to-many relationship, so our `Customers` markup would probably look something like the following:

```
builder.customers {
    customer{
        invoice {
            salesOrder()
            salesOrder()
            salesOrder()
        }
    }
}
```

We need to be able to set the fields of each entity as it is created. We could have a pretended method for each field as follows:

```
builder.customers {
    customer {
        firstName("Fred")
        lastName("Flintstone")
        invoice {
            salesOrder {
```

```
    sku("productid01")
    amount(1)
    price(1.00)
  }
  salesOrder {
    sku("productid02")
    amount(2)
    price(1.00)
  }
  salesOrder {
    sku("productid03")
    amount(3)
    price(1.00)
  }
}
}
```

This will work. However, it is not immediately clear to a reader of this script that `lastName` is an object attribute and `invoice` is a new subsidiary object. A better option is to set object attributes as mapped parameter values. The following script is far clearer in its intent, so this is the one we will try to implement:

```
builder.customers {
  customer(firstName:"Fred", lastName:"Flintstone") {
    invoice {
      salesOrder(sku:"productid01", amount:1, price:1.00)
      salesOrder(sku:"productid02", amount:2, price:1.00)
      salesOrder(sku:"productid03", amount:3, price:1.00)
    }
  }
}
```

As it happens, the `BuilderSupport` hook methods and their calling sequence work perfectly in step with the GORM APIs that we need in order to construct our customer records.

- The `createNode` method will be called in the correct top down sequence, allowing us to create the appropriate `Customer`, `Invoice`, or `SalesOrder` as required.
- The `setParent` hook is called after both parent and child objects have been constructed, allowing us to call `Customer.addToInvoices` or `Invoice.addToOrders` when we need to.
- The `nodeCompleted` hook can be used to intercept when an object needs to be saved.

The following code snippet contains a rudimentary builder class based on `BuilderSupport` that constructs customer, invoice, and sales order objects through GORM. The same style of builder could work equally well with whatever other persistence method we choose.

```
class CustomersBuilder extends BuilderSupport {
    def createNode(name) {
        Object result = null
        switch (name) {
            case "customer":
                return new Customer(firstName:"", lastName(""))
            case "invoice":
                return new Invoice()
            case "salesOrder":
                return new SalesOrder(sku:"default", amount:1, price:0.0)
        }
    }
    def createNode(name, value) {
        Object result = createNode(name)
        if (value instanceof Customer && result instanceof Invoice)
            value.addToInvoices(result)
        if (value instanceof Invoice && result instanceof SalesOrder)
            value.addToOrders(result)
        return result
    }
    def createNode(name, Map attributes) {
        Object result = null
        switch (name) {
            case "customer":
                return new Customer(attributes)
            case "invoice":
                return new Invoice(attributes)
            case "salesOrder":
                return new SalesOrder(attributes)
        }
    }
    def createNode(name, Map attributes, value) {
        Object result = createNode(name, attributes)
        if (value instanceof Customer && result instanceof Invoice)
            value.addToInvoices(result)
        if (value instanceof Invoice && result instanceof SalesOrder)
            value.addToOrders(result)
        return result
    }
}
```

```
void setParent(parent, child){  
    if (child instanceof Invoice && parent instanceof Customer)  
        parent.addToInvoices(child)  
    if (child instanceof SalesOrder && parent instanceof Invoice)  
        parent.addToOrders(child)  
}  
void nodeCompleted(parent, node) {  
    if (node != null)  
        node.save()  
}  
}
```

Here we have implemented all four `createNode` methods. The method tag is passed as the `name` parameter to `createNode`. So we construct a `Customer`, `Invoice`, or `SalesOrder` object based on the tag that we are processing. We allow a `parent` object to be set in the `value` parameter to the method. This allows us to construct a `child` object outside of the scope of a `parent`, and set its `parent` later.

The `setParent` method takes care of adding invoices to customers and sales orders to invoices. Testing the `instanceof` both `parent` and `child` ensures that we don't attempt to add an invoice if it is declared outside of the customer.

All that remains for `nodeCompleted` to do is to save the `node` object that we have created to the database. When we put all of this together, we can make use of our `CustomerBuilder` to build a simple test database as follows:

```
def builder = new CustomersBuilder()  
  
def customers = builder.customers {  
    fred = customer(firstName:"Fred",lastName:"Flintstone") {  
        invoice {  
            salesOrder(sku:"productid01", amount:1, price:1.00)  
            salesOrder(sku:"productid02", amount:2, price:1.00)  
            salesOrder(sku:"productid03", amount:3, price:1.00)  
        }  
    }  
    invoice2 = invoice(fred)  
  
    salesOrder(invoice2, sku:"productid04", amount:1, price:1.00)  
    salesOrder(invoice2, sku:"productid05", amount:1, price:1.00)  
    salesOrder(invoice2, sku:"productid06", amount:1, price:1.00)  
}
```

By allowing a parent object to be passed as the value parameter, we have made the markup script more flexible. As you can see from the preceding code, `invoice` and `salesOrder` tags can be declared directly as children of a parent object, or they can be declared independently. This gives us a bit more flexibility in what types of mapping relationships we can support where ownership between parent and child might be optional, or in more complex scenarios where many-to-many relationships might need to be declared.

FactoryBuilderSupport

`BuilderSupport` is the base class for many of the builder classes provided in the Groovy packages. As we can see from the previous examples, it is easy to work with. We have built quite a useful database builder tool in relatively few lines of code.

However, one issue with `BuilderSupport` is that the hook functions are in effect funnels for handling all of the possible tags that we might like to process in our markup. In our `CustomerBuilder` we are handling just four different tags.

This is not a realistic scenario for most database schemas. We could expect to have dozens more tag types that we need to handle if we wanted to expand this example into something that would work with a typical database schema for even a modestly sized application. Funneling all of these tags into one `createNode` would create an unwieldy mess of code.

```
def createNode(name) {
    Object result = null
    switch (name) {
        case "customer":
            return new Customer(firstName:"", lastName(""))
        case "invoice":
            return new Invoice()
        case "sales_order":
            return new SalesOrder(sku:"default", amount:1, price:0.0)
        case "another_object":
            return new Something()
            ....... and more!
    }
}
```

Groovy provides a second builder support class that neatly overcomes this problem. The `groovy.util.FactoryBuilderSupport` class is based on the factory pattern, and delegates the handling of individual tag objects to Factory classes. Originally, this support class was just provided as part of the `SwingBuilder`. Because it was clear that this was more generally useful, the code was then refactored to be generally usable as a standalone `Builder` class. Since then, it has become the basis for other builders, such as the recently-added `JmxBuilder`, and is available to us for deriving our own factory-based builders.

`FactoryBuilderSupport` works by orchestrating the construction process in concert with Factory classes. When the `FactoryBuilderSupport` class encounters a method tag, it constructs an appropriate Factory object to handle it. The factory provides method hooks that implement the construction of the individual object and the setting up of parent-child relationships between the objects.

To implement a builder with the `FactoryBuilderSupport` class, we must first declare a Factory class for each object type that we wish to process. Factory classes are derived from the `groovy.util.AbstractFactory` class and need to overload some or all of the following methods from the `AbstractFactory` class:

- `newInstance`

Called by `FactoryBuilderSupport` whenever it wants an object of a particular type to be constructed. It is similar to the `createNode` methods of `BuilderSupport` except that there is just one `newInstance` method, which accepts all argument types regardless of whether a value or attributes are supplied or not.

- `onHandleNodeAttributes`

Allows the Factory class to take over the management of attributes. It can stop the builder from processing attributes by returning `true`.

- `setParent` and `setChild`

Provide hooks for managing the parent-child relationships between objects.

- `isLeaf`

We set this method to return `true` if the method tag being handled should be a leaf node and stops the builder treating any subsequent method calls as object declarations.

- `onNodeCompleted`

Called when a node is completed, in order to allow any finalization of the object to be performed. It is similar to `nodeCompleted` in `BuilderSupport`.

To build a replacement for the `CustomerBuilder` with `FactoryBuilderSupport`, we first need to define Factory classes for each of the tag methods that we need to process. The first of these is the `customers` tag, which is straightforward enough. This tag does not cause any objects to be created, so all we do is return the tag name as the object created.

```
public class CustomersFactory extends AbstractFactory {

    public boolean isLeaf() {
        return false
    }

    public Object newInstance(FactoryBuilderSupport builder,
        Object name, Object value, Map attributes
    ) throws InstantiationException, IllegalAccessException {
        return name
    }

}
```

We then define a factory class for the `customer` object. The methods that we need to implement are `isLeaf` (returns `false`), `newInstance` (to create the `customer` object), and `onNodeCompleted` (to save it).

```
public class CustomerFactory extends AbstractFactory {

    public boolean isLeaf() {
        return false
    }

    public Object newInstance(FactoryBuilderSupport builder,
        Object name, Object value, Map attributes
    ) throws InstantiationException, IllegalAccessException {
        Customer customer = null
        if (attributes != null)
            customer = new Customer(attributes)
        else
            customer = new Customer()
        return customer
    }

    public void onNodeCompleted(FactoryBuilderSupport builder,
        Object parent, Object customer) {
        customer.save()
    }
}
```

The factory for invoices is equally straightforward. The only addition is that we need to take care of the parent-child relationship between customer and invoice. We do this by adding a `setParent` method, which will call `addToInvoices` on the `customer` object if required. We also need to check the `value` parameter passed to `newInstance` to see if a parent is being set at this point.

```
public class InvoiceFactory extends AbstractFactory {

    public boolean isLeaf() {
        return false
    }

    public Object newInstance(FactoryBuilderSupport builder,
        Object name, Object value, Map attributes
    ) throws InstantiationException, IllegalAccessException {
        Invoice invoice = null
        if (attributes != null)
            invoice = new Invoice(attributes)
        else
            invoice = new Invoice()
        if (value != null && value instanceof Customer)
            value.addToInvoices(invoice)
        return invoice
    }

    public void setParent(FactoryBuilderSupport builder,
        Object parent, Object invoice) {
        if (parent != null && parent instanceof Customer)
            parent.addToInvoices(invoice)
    }

    public void onNodeCompleted(FactoryBuilderSupport builder,
        Object parent, Object invoice) {
        invoice.save()
    }
}
```

The factory for sales orders is identical to invoices except that we now return `true` from `isLeaf` because a sales order object will always be a leaf node in our tree.

```
public class SalesOrderFactory extends AbstractFactory {
    public boolean isLeaf() {
        return true
    }
}
```

```
public Object newInstance(FactoryBuilderSupport builder,
    Object name, Object value, Map attributes
) throws InstantiationException, IllegalAccessException {
    SalesOrder sales_order = null
    if (attributes != null)
        sales_order = new SalesOrder(attributes)
    else
        sales_order = new SalesOrder()
    if (value != null && value instanceof Invoice)
        value.addToOrders(sales_order)
    return sales_order
}

public void setParent(FactoryBuilderSupport builder,
    Object parent, Object sales_order) {
    if (parent != null && parent instanceof Invoice)
        parent.addToOrders(sales_order)
}

public void onNodeCompleted(FactoryBuilderSupport builder,
    Object parent, Object sales_order) {
    sales_order.save()
}
}
```

All of the intelligence of how to orchestrate the construction process is encapsulated in the `FactoryBuilderSupport` class. So literally all we need to do for the whole builder to work is to register the Factory classes with appropriate tag names.

```
public class CustomersFactoryBuilder extends FactoryBuilderSupport {
    public CustomersFactoryBuilder(boolean init = true) {
        super(init)
    }

    def registerObjectFactories() {
        registerFactory("customers", new CustomersFactory())
        registerFactory("customer", new CustomerFactory())
        registerFactory("invoice", new InvoiceFactory())
        registerFactory("sales_order", new SalesOrderFactory())
    }
}
```

FactoryBuilderSupport uses reflection at runtime to detect what registration methods to run. By scanning the list of methods in the MetaClass instance and looking for methods that begin with "register", FactoryBuilderSupport detects whether any additional registration methods are provided in the derived builder class. In the preceding example, the only registration method added is registerObjectFactories, but we could well have written:

```
def registerCustomers() {
    registerFactory("customers", new CustomersFactory())
}

def registerCustomer() {
    registerFactory("customer", new CustomerFactory())
}

def registerInvoice() {
    registerFactory("invoice", new InvoiceFactory())
}

def registerSalesOrder() {
    registerFactory("sales_order", new SalesOrderFactory())
}
```

FactoryBuilderSupport would detect all of these and run them in turn. Which method you use is a matter of choice. The only issue that you need to be aware of is that the registration methods will not be called in any predetermined order. If there are dependencies in your registration code, then it's best to group these into a single registration method.

To finish, we can drop this modified builder right where we previously used *CustomersBuilder*, and it will work in the same way.

```
def builder = new CustomersFactoryBuilder()

def customers = builder.customers {
    fred = customer(firstName:"Fred", lastName:"Flintstone") {
        invoice {
            salesOrder(sku:"productid01", amount:1, price:1.00)
            salesOrder(sku:"productid02", amount:2, price:1.00)
            salesOrder(sku:"productid03", amount:3, price:1.00)
        }
    }
    invoice2 = invoice(fred)

    salesOrder(invoice2, sku:"productid04", amount:1, price:1.00)
    salesOrder(invoice2, sku:"productid05", amount:1, price:1.00)
    salesOrder(invoice2, sku:"productid06", amount:1, price:1.00)
}
```

In terms of management and maintenance, this version is far superior. Adding capabilities now for new tables will simply involve writing a new Factory class and registering it.

Summary

In Chapter 5, we discussed how builders worked via the Groovy MOP. In this chapter, we have taken a deeper look at how features of the MOP are used to implement the builder pattern. We've looked at the language features used to create a builder, and seen how they involve implementing pretended methods and influencing how method calls are resolved. Implementing a builder directly by using the MOP in this way focuses on the nuts and bolts of the semantics of the builder, rather than the construction process.

In this chapter, we have seen how Groovy provides two useful support classes that make it much simpler to implement our own builders than if we use the MOP. We've seen how to use `BuilderSupport` and `FactoryBuilderSupport` to create our own builder classes.

Using these support classes greatly simplifies the implementation of builders. Hopefully, this will inspire you to see opportunities to develop your own Groovy-based builders for your own applications. You can find the full documentation for all of the classes that we covered here on the Codehaus website. The Groovy document for the classes can be found at <http://groovy.codehaus.org/api/groovy/util/package-summary.html>.

8

Implementing a Rules DSL

In this chapter, we will look at how we can use Groovy to build a DSL that is capable of implementing business rules for an application. The example we will use is a system for implementing rewards and bonuses of various kinds as part of a promotions system for an online broadband media provider.

Our provider hosts a service that allows users to view videos and play games online. The provider needs to be able to deploy offers to his users rapidly and with the minimum amount of development time. We will come up with a Groovy-based DSL that expresses rewards in such a way that they can be rapidly developed and deployed in a language that can also be understood by business stakeholders.

This DSL relies on a new concept that we have not covered yet, which is the use of the Groovy binding. To begin with, we will look at Groovy bindings—how they work and how we can make use of them to improve our DSLs. We will cover a number of useful techniques that make use of the binding.

- Using the binding in combination with closures to introduce built-in methods into a DSL.
- Adding closures to the binding to implement structured named blocks in a DSL.
- How Boolean and other values added to the binding can be used to build contextual data for a DSL.
- How to return values and results from a DSL script.

We will use all of these techniques in concert, and build a sample DSL step by step.

Groovy bindings

Every Groovy script has an associated binding object. The binding is where instances of variables referenced within the script are stored. The binding is an instance of the class `groovy.lang.Binding`, and we can access it in any script by referencing the built-in variable `binding`, as the next example will show.



When we reference a previously undeclared variable in a script, Groovy creates an instance of the variable in the binding. On the other hand, variables that are defined within the script are considered local variables and are not found in the binding. The latter provides a convenient placeholder where Groovy can store these variables. This also presents the DSL with an opportunity. By manifesting variables in the binding, we can manipulate the script with predefined values. By adding a closure into the binding, we have the opportunity to provide built-in methods for the DSL.

In the following example, when we reference a new variable named `count` in a script, we see how that variable is stored in the binding. If we explicitly declare the variable `local` with `def`, we can use both variables interchangeably, but only `count` is stored in the binding.

```
count = 1

assert count == 1
assert binding.getVariable("count") == 1
binding.setVariable("count", 2)
assert count == 2
assert binding.getVariable("count") == 2

def local = count

assert local == 2
try {
    binding.getVariable("local")
    assert false
} catch (e) {
    assert e instanceof MissingPropertyException
}
```



Most of the script examples we have encountered in the book can be run either on the command line or the Groovy Console `groovyConsole`. The examples in this chapter are best run from the command line. The Groovy console maintains a single binding object. So each time you execute a script from the buffer, you are inheriting objects that were probably stored there during the previous runs. The examples in this chapter all assume a clean binding, so running successive examples in `groovyConsole` will lead to unpredictable results.

The power of bindings with regard to their use in DSLs comes from the fact that we can add a variable to the binding on the fly. If `count` does not exist as a variable in the script, then it can be added by a call to `setVariable`, as follows:

```
binding.setVariable("count" , 1)

assert binding.getVariable("count") == 1

binding.setVariable("count" , 2)

assert binding.getVariable("count") == 2
```

The Binding class also implements the property access APIs, such as `setProperty`, `getProperty`, and `getProperties`. This means that the binding will allow bean-like access, including the use of the subscript operator.

```
binding.count = 1

assert binding.getProperty("count") == 1

binding.setProperty("count" , 2)

assert binding.count == 2
```

The examples above might look like a clumsy way of getting and setting variables in a script, but the binding becomes really useful when we execute a script that we have loaded. Here we set a property `message` in the binding and then use the `GroovyShell` class to execute a script snippet that uses it.

```
def Binding binding = new Binding()

binding.message = "Hello, World"

shell = new GroovyShell(binding)

shell.evaluate("println message")
```

This will output the string Hello, World to the console. In other words, we have managed to introduce a variable into this script called message that has the preset value Hello, World.

In Chapter 4, we wrote a simple Twitter DSL by using just closures. This suffered from the fact that the main Twitter functions were implemented through a class. Users with a programming background will not be put off by the need to start all calls with GeeTwitter, but it will appear clumsy to other users. We saw how we can add built-in methods to the DSL by using the CompilerConfiguration class and creating a sub class of Script. This allows us to add verbs such as login and follow to the DSL.

Now we are going to explore an alternative method of creating built-in methods. Using the binding, we can add these verbs by adding closure properties to our script binding. We can also exploit the fact that Groovy allows us to take the address of a class method and use it as a closure. All we need to do in this case is to modify the startup script for GeeTwitter as follows:

```
#!/usr/bin/env groovy

String.metaClass.search = { Closure c ->
    GeeTwitter.search(delegate, c)
}

if(args) {
    def binding = new Binding()
    binding.login = GeeTwitter.&login
    binding.sendMessage = GeeTwitter.&sendMessage
    binding.follow = GeeTwitter.&follow
    binding.search = GeeTwitter.&search
    binding.eachFollower = GeeTwitter.&eachFollower
    binding.eachFriend = GeeTwitter.&eachFriend

    def shell = new GroovyShell(binding)
    shell.evaluate (new File(args[0]))
}
else
    println "Usage: GeeTwitter <script>"
```

In this version, we are adding six closures to the binding before we execute the DSL script. This gives us a greater degree of flexibility as to where we implement our built-in methods. These can be methods from any class or standalone closure instances. Now in the GeeTwitter scripts, we can use much more expressive language. To send a message to all my followers, I will write:

```
login 'myid', 'mypassword'
eachFollower {
    sendMessage it, "Thanks for taking the time to follow me!"
}
```

Exploiting bindings in DSLs

There are numerous ways in which we can use bindings in our DSLs. In this section, we will discover how to use closures in the binding to implement several different DSL styles. We will also look at how simply adding properties to the binding can be an effective way to augment a DSL with shorthand.

Closures as built-in methods

In the previous GeeTwitter example, we created closures in the binding by referencing the address of class member. If we are writing the DSL from scratch, the simpler option would be to do away with the GeeTwitter class altogether and just implement the methods as closures in the first place. If we had done this in the original DSL, the set-up code for the binding might look like this:

```
binding.login = {id, password ->
    binding.twitter = new Twitter(id, password)
}
```

Now, instead of storing the shared Twitter instance in a static member of the GeeTwitter class, we are storing it directly in the binding. In effect, this has the same effect as if we had written `twitter = new Twitter(id, password)` as the first line of our script.

Closures as repeatable blocks

We've seen how using a closure within the binding can give the impression of having built-in functions in our DSL. We can also use closures in the binding to allow a nested block structure to be represented in the DSL. Using this style, we can repeat a block multiple times within a single script. This is useful when we have a DSL that needs to define multiple instances of the same entity or logic within the same script. Take the following script example:

```
block {
    nestedBlock {
    }
}
block {
    nestedBlock {
    }
}
```

We can implement this structure by adding two closures to the binding called `block` and `nestedBlock`. The `block` and `nestedBlock` closures accept a closure as their only parameter. We saw in Chapter 5 how this is exactly the same mechanism that is used to implement builders. The `block` and `nestedBlock` closures need to manage their delegates, in order to ensure that the expected binding scopes are preserved.

```
binding.block = { closure ->
    def cloned = closure.clone()
    cloned.delegate = delegate
    this.enclosing = "block"

    println "block encountered"
    cloned()
}

binding.nestedBlock = { closure ->
    assert closure.delegate.enclosing == "block"
    def cloned = closure.clone()
    cloned.delegate = delegate
    this.enclosing = "nestedBlock"

    println "nested block encountered"
    cloned()
}

block {
    nestedBlock {
    }
}
block {
    nestedBlock {
    }
}
```



In these examples, we cloned the passed-in closure before changing the delegate. This is considered the best practice with a DSL, in case the original closure is also used externally. This is also advisable if we make any changes to the closure resolve strategy.

By adding an `enclosing` property to the `block` and `nestedBlock` closures, we ensure that `nestedBlock` is only allowed within `block`. Placing `nestedBlock` outside of `block` will trigger the assertion. Running the original script with these closures in the binding will give the output below:

```
block encountered
nested block encountered
```

Using a specification parameter

If we like, we can also use closures that take one or more parameters in addition to the closure parameter. A common style, when using a block structured DSL as above, is to add a specification parameter to identify individual blocks. The implementation can choose to ignore this parameter, or it can be used as a means of identifying the individual blocks.

```
binding.block = { spec, closure ->
    def cloned = closure.clone()
    cloned.delegate = delegate
    this.enclosing = "block"

    println "${spec} encountered"
    cloned()
}

binding.nestedBlock = { spec, closure ->
    assert closure.delegate.enclosing == "block"
    def cloned = closure.clone()
    cloned.delegate = delegate
    this.enclosing = "nestedBlock"

    println "${spec} encountered"
    cloned()
}

block ("first block") {
    nestedBlock ("first nested") {
    }
}
block "second block", {
    nestedBlock ("second nested") {
    }
}
```

This outputs:

```
first block encountered
first nested encountered
second block encountered
second nested encountered
```

In the previous code snippet, we see two styles of declaring a specification parameter:

```
block ("spec") {  
}  
block "spec", {  
}
```



The first uses Groovy 'function call' style, passing the closure after the method parentheses. The second uses a parameter list, where one of the parameters is the inline closure. Which one you choose is a matter of personal preference and style.

Some frameworks, such as EasyB, have a preference for the latter style in their examples. My own personal preference is for the former, for the simple reason that it is easier for non-technical users to grasp the necessity for a `(something) {}` syntax rather than a `something, {}` syntax.

Closures as singleton blocks

The previous DSL style allows us to implement logic in repeatable named blocks. A DSL script of this style could be run once or multiple times. For instance, the DSL could describe a set of business rules to be executed every time we encounter a certain event. Sometimes we will need a DSL to define logic that is only ever going to be run once, or that needs to be stored and executed at will at some later date. In this case, it may be better to limit the user to a single block instance, by having them define the closure directly.

```
setup = {  
    println "Initialized"  
}  
  
teardown = {  
    println "finished"  
}
```

Here we declare two named closures: `setup` and `teardown`. We can now provide default implementations of `setup` and `teardown` in the runtime that we use to load and evaluate this script.



For brevity, in some of the following examples, we will use `GroovyShell` to evaluate our DSL scripts from a `GString`. In most real-life DSL scenarios, you will want to externalize your DSL code. `GroovyShell` can also be used to load and evaluate a script from a file.

```
def binding = new Binding()
binding.setup = {
    println "Setup block is missing"
    throw new Exception("Setup block is missing")
}

binding.teardown = {
    println "Teardown block is missing"
    throw new Exception("Teardown block is missing")
}

def shell = new GroovyShell(binding)
shell.evaluate(
"""
setup = {
    'setup called'
}
teardown = {
    'teardown called'
}
"""

)

setup = binding.setup
assert setup() == 'setup called'
// ... do something now and save teardown closure for later
teardown = binding.teardown
assert teardown() == 'teardown called'
```

An exception will be thrown if either `setup` or `teardown` has not been provided. This is a useful tactic to use to ensure that one and only one block is executed from the DSL. It also gives us control over the timing of when the blocks are actually executed.

The only word of caution to heed is that while using both this style of block and the previous in a single DSL, users will need to beware of the subtle difference between `block {}` and `block = {}`. Groovy will allow a user to specify either, and this can give unexpected results that might be confusing to the general user.

Using binding properties to form context

Most DSLs will need to have some predetermined knowledge of the domain within which they operate. So, for instance, if we were to write a DSL that described the rewards that a user might get for making purchases, it makes sense that the DSL would have built-in access to the user's account details and his purchasing history, rather than requiring complex lookups to be performed within the DSL.

The binding is the ideal place in which to store these details. Depending on the sophistication of the DSL target audience, we could decide to embed existing domain objects in the binding, or alternatively we could look up or pre-calculate values that make sense to the DSL, and embed these.

```
class Account {
    double spend = 11.00
    double balance = 100.00
    boolean active = true

    void credit (double value) {
        balance += value
    }
}

def binding = new Binding()
binding.reward = { closure ->
    closure.delegate = delegate
    closure()
}

binding.apply = { closure ->
    closure.delegate = delegate
    closure()
}

// lookup account in binding
def account = new Account()
binding.account = account
binding.monthSpend = account.spend
binding.credit = Account.&credit

assert account.balance == 100.00

def shell = new GroovyShell(binding)
shell.evaluate(
"""
    reward {
        apply {
            if (account.active && monthSpend > 10.00)
                credit 5.00
        }
    }
"""
)

assert account.balance == 105.00
```

Here we embed an account object in the binding, along with a calculated value for the user's monthly spend to date. We also have introduced a shortcut for the account credit method, by including a closure called credit, which is taken from the address of the Account.credit method.

Another useful technique is to pre-determine states and boolean conditions, and store them in appropriately-named binding variables. For the rewards DSL, some common tests that we might need to make are whether the account is active, and whether the minimum spending threshold has been reached. Setting these conditions into binding variables will further improve the readability of the DSL.

```
reward {  
    apply {  
        if (ACTIVE && REWARD_THRESHOLD_EXCEEDED)  
            credit 5.00  
    }  
}
```

Storing and communicating results

Capturing the values of variables set in the DSL can also be done through the binding. The delegate is set by each calling closure, so any variables defined in the scope of the DSL blocks will be available as binding variables to the calling closures.

```
def binding = new Binding()  
binding.outerBlock = { closure ->  
    closure.delegate = delegate  
    closure()  
    println "outerBlock: " + binding.message  
}  
  
binding.innerBlock = { closure ->  
    closure.delegate = delegate  
    closure()  
    println "innerBlock: " + binding.message  
}  
  
def shell = new GroovyShell(binding)  
shell.evaluate(  
    """  
        outerBlock {  
            innerBlock {  
                message = "Hello, World!"  
            }  
        }  
    """  
)  
println "caller: " + binding.message
```

In the above example, the `message` variable is set in the innermost block of the DSL, but we can reference it from the outer block closure, and also from the calling script. Variables set like this in the binding are global to the script; so care must be taken to initialize them to default values before referencing them. Otherwise, subsequent blocks within the DSL will reuse the values. Below, the `Hello, World!` message, value is still set when the second `outerBlock` is evaluated.

```
outerBlock {  
    innerBlock {  
        message = "Hello, World!"  
    }  
}  
outerBlock {  
    println message  
}
```

The output produced by this will not be what we expected. Setting a default value for `message` in the closure definition for `binding.outerBlock` will overcome this.

```
inner: Hello, World!  
outer: Hello, World!  
Hello, World!  
outer: Hello, World!  
caller: Hello, World!
```

We know that closures and methods in Groovy will return a value even when no `return` statement is used. The value returned is the result of the last statement executed in the method or closure. We can exploit this in our DSLs. The value returned from the `innerBlock` above is the result of `message = "Hello, World!"` – in other words, the string `"Hello, World!"`. We can define a closure that captures a string value from the DSL, as follows:

```
binding.messageBlock = { closure ->  
    closure.delegate = delegate  
  
    binding.message = closure()  
    println "messageBlock: ${binding.message}"  
}
```

This allows us to define a message string by using the following DSL code:

```
outerBlock {  
    messageBlock {  
        "Hello, World!... message"  
    }  
}
```

Using this style, we can define a DSL block that expects a Boolean return value and use it to define a conditional expression. Going back to the reward DSL we used earlier, we could write the following conditional DSL code:

```
reward {  
    appliesWhen {  
        ACTIVE && REWARD_THRESHOLD_EXCEEDED  
    }  
}
```

We can document to our DSL users that `appliesWhen` declares a condition that must be met if the reward between the curly braces is to be awarded.

Bindings and the GORM DataSource DSL

We can now see how powerful the binding is in helping us to structure expressive DSLs. Before we go on to develop our own DSL using all of these techniques, we'll briefly revisit a DSL that we came across in an earlier chapter. The following DSL script is the mini DSL used in GORM configuration to set up DataSources for connection to the various development, production, and test database environments used by Grails. With the knowledge that we now have about how bindings work, it should be clear what techniques the Grails folks have used to implement this.

```
environments {  
    development {  
        dataSource {  
            dbCreate = "create-drop"  
            url = "jdbc:mysql://localhost/groovydsl"  
            driverClassName = "com.mysql.jdbc.Driver"  
            username = "root"  
            password = ""  
        }  
    }  
    test {  
        dataSource {  
            dbCreate = "create-drop"  
            url = "jdbc:hsqldb:mem:testDb"  
        }  
    }  
    production {  
        dataSource {  
            dbCreate = "update"  
            url = "jdbc:hsqldb:mem:testDb"  
        }  
    }  
}
```

In the above script, the individual `dataSource` blocks are used to define variables such as `dbCreate`, `url`, and so on. These variables are available within the binding and are picked up by the development, test, and production closures respectively, when configuring the corresponding database connection.

Building a Rewards DSL

The old adage that 80 percent of business comes from your existing customers while 20 percent comes from new customers is as true today as it ever was. Every business, at some point in time, considers offering incentives to its customers in order to increase sales. Rewards can take the form of everything from the selective discounting of end-of-line items, through buy-one-get-one-free promotions, to customer loyalty points schemes.

Marketers constantly devise new ways to promote products and services to customers, but often the problem is that these promotions can be difficult to manage when they need to be implemented in the various back-end systems. Configuring a reward could involve applying cross-cutting logic across several systems. Developing and deploying a promotion can take weeks or months to complete, while the marketing department wants to be able to respond to the conditions in the market today.

In this next example, we will take an imaginary broadband service provider that provides access to on-demand video and games content. We will devise a simple Groovy-based DSL that expresses reward programs in simple to understand terms. Although the DSL code is not going to be developed by a marketer, a marketer should be able to understand what the code does simply by reading it. This DSL also has the added benefit of being something that can be deployed directly. As such, the DSL should be able to serve as both the specification and the implementation of the reward.

Designing the DSL

Before attempting to design our DSL, it makes sense for us to review our business domain and understand our requirements.

BroadbandPlus

Users of **BroadbandPlus**, our imaginary broadband service, can subscribe to three levels of access: BASIC, PLUS, and PREMIUM. There is a range of content that a subscriber can consume, including games, movies, and music. Each subscriber can consume any mix of content up to the maximum allowed on their plan, after which they need to pay for any additional content that they consume.

To simplify everything, we will track and allow the subscriber to pick and mix their content. Each type of media consumed has an "access point" value, which is debited from the user's account when they consume it. Each subscriber type is allocated a set number of access points each month, based on their plan. The following tables show how the access point structure works:

Subscription Plan	Monthly Subscription	Access Points
Basic	\$9.99	120
Plus	\$19.99	250
Premium	\$39.99	550

Roughly speaking, an access point equates to 10 cents in value, so basic subscribers are benefiting from a 20 percent bonus versus non-subscribers. Moreover, plus subscribers get 25 percent extra while premium subscribers get a whopping 37.5 percent.

Media	Points	Type of Access	Out of Plan Price
Movies			
New Release:	40	Daily	\$3.99
Other:	30	Daily	\$2.99
Games			
New Release:	30	3 days access	\$2.99
Other:	20	3 days access	\$1.99
Songs	10	Download	99c

Before consumption of any media is allowed, the system does a `canConsume` test. This test is passed if the user has enough access points, or if access has been granted already to the content and has not yet expired. If the `canConsume` test is passed, access to the media is granted when the first `consume` call is made, otherwise the user is prompted to approve the purchase of the media, followed by the `consume` call for an authorized purchase.

From the point of view of our rewards program, the APIs that we need to be concerned with are defined in the following stub class for `BroadbandPlus`:

```
class BroadbandPlus {

    boolean canConsume(subscriber, media) {
    }
    void consume(subscriber, media) {
    }
}
```

```
void purchase(subscriber, media) {  
}  
void upgrade(subscriber, fromPlan, toPlan) {  
}  
}
```

Reward types

We are in the business of encouraging the subscriber to continue to consume content out of plan or to upgrade to a higher plan. So our rewards programs will offer incentives that apply at the time of purchase or upgrade. Rewards usually consist of allocating free points, but we also want to be able to offer free content or extended access.

Our partners are the studios, game developers, and record labels that publish our content. We don't mind what content the subscriber consumes so long as our revenue comes in. As our partners get a revenue share based on which titles the subscriber consumes, they will want to be able to sponsor promotions that target specific content and specific publishers. In other words, these particular types of targeted reward programs need to be activated at the point of consumption rather than purchase.

Some examples of the types of rewards that we might like to deploy are:

- Consume any new release this week and get 10 free access points on your account.
- Earn 10 percent bonus points for every game purchased.
- Watch any Disney movie for 25 percent off.
- Upgrade from Basic to Plus and get 100 free access points on your account.

The Reward DSL

Taking all of this into account, we can make an attempt at writing a DSL. The requirements that we have for our DSL can be summarized as follows:

- Rewards need to be triggered by different events in the system. These events are **consumption** (when a user consumes a product—that is watch a movie, play a game, and so on), **upgrades** (when the user upgrades their subscription plan), and **purchases** (any time that the user spends some cash).
- Rewards need to be based on one or more conditions, such as the user's spending history or the type of media being consumed.
- Rewards can result in the granting of different benefits, for example free access to a video, bonus points, and extended access.

Based on these requirements, and with an understanding of how we can structure a DSL using closures and binding variables, we can make an attempt at how our DSL might look.

```
onConsume = { // or on_purchase or on_upgrade
    reward ( "Reward Description" ) {
        condition {
            // Condition(s) that need to apply
        }
        grant {
            // benefits that can accrue
        }
    }
}
```

We can implement the conditional nature of the above code snippet by using a binding variable to collect the result of the condition block. The closure below shows this in action. The `condition` closure collects the result of its own closure, which in turn dictates whether the `grant` closure is invoked.

```
binding.condition = { closure ->
    closure.delegate = delegate

    binding.result = (closure() && binding.result)
}

binding.grant = { closure ->
    closure.delegate = delegate

    if (binding.result)
        closure()
}
```

If the target audience for this DSL were to be only software developers, then this would be adequate. A single condition block returning a result could fully capture the logic required to allow or disallow a reward. The problem with this is that the only way to express multiple conditions is through Groovy conditional logic. Programmers don't mind reading this, but the other audiences for the DSL will quickly get confused by the Groovy syntax involved. Groovy `&&` and `||` operators, along with the operator precedence rules, are not going to make for easy reading to the general audience.

Ideally, we want to have a single condition in each block, so we need to allow multiple blocks, and provide an easy way to describe inclusive and exclusive sets of conditions. Adding two more closures to the DSL gives us just that.

```
reward ( "anyOf and allOf blocks" ) {
    allOf {
        condition {
        }
        ... more conditions
    }
    condition {
    }
    anyOf {
        condition {
        }
        ... more conditions
    }
    grant {
    }
}
```

Now we can have multiple condition blocks within a reward. All condition blocks must be true for the reward to be granted. The `allOf` and `anyOf` condition blocks can each themselves contain multiple condition blocks. For an `allOf` to be true, all child conditions must be true. For an `anyOf` block to be true, at least one of the child conditions must be true.

To implement this scheme, we need to tell the condition block whether it should AND (`&&`) or OR (`||`) its result to the current condition of the expression. We shall store the current status of the condition in a binding variable called `result` and decide whether to `&&` or `||` based on the status of the binding Boolean `useAnd`. To begin with, for each reward we presume that the reward is passed, and set `result` to `true`. We set the default operator to `&&` by setting `use_and` to `true`.

```
binding.reward = { spec, closure ->
    closure.delegate = delegate
    binding.result = true
    binding.useAnd = true
    closure()
}

binding.condition = { closure ->
    closure.delegate = delegate
}
```

```
if (binding.useAnd)
    binding.result = (closure() && binding.result)
else
    binding.result = (closure() || binding.result)
}
```

To implement the `allOf` closure block, we store the current states of `result` and `useAnd`, before calling the child closure. We `&&` or `||` the stored result with the new result from the closure, giving us the new boolean state of the expression. The starting presumption of an `allOf` block is that it is `true`. It will be set to `false` if any one of the child conditions returns `false`.

```
binding.allOf = { closure ->
    closure.delegate = delegate
    def storeResult = binding.result
    def storeAnd = binding.and
    binding.result = true // Starting premise is true
    binding.and = true

    closure()

    if (storeAnd) {
        binding.result = (storeResult && binding.result)
    } else {
        binding.result = (storeResult || binding.result)
    }
    binding.and = storeAnd
}
```

The `anyOf` closure block is identical to `allOf` except that the starting presumption is `false`. The operator that we now use is `||`, so if any one of the child conditions returns `true`, the overall result will be `true`.

```
binding.anyOf = { closure ->
    closure.delegate = delegate
    def storeResult = binding.result
    def storeAnd = binding.and

    binding.result = false // Starting premise is false
    binding.and = false

    closure()
    if (storeAnd) {
        binding.result = (storeResult && binding.result)
    } else {
```

```
        binding.result = (storeResult || binding.result)
    }
    binding.and = storeAnd
}
```

This gives us a very flexible conditional logic that we can include in the DSL, which is still very legible to a general audience. We can test the effectiveness of this with a few assertions. The `result` binding variable is available to the DSL script, so we can use that to assert that the result is as expected. Next we want try out some of the ways to use our conditional DSL, but first let's put our DSL into a script that we can run from the command line.

This script sets up our mini DSL by adding some closure to the binding. It then invokes the reward script passed to it via the command line. We can test out the validity of our conditional logic by passing some of the following conditional scripts:

- A reward defined without any conditions should always be true.

```
reward ( "No conditions" ) {
    assert result == true
}
```

- With a single condition, the state of the reward is dictated by that one condition.

```
reward ( "One false condition" ) {
    condition {
        false
    }
    assert result == false
}
```

- By using `allof` and `anyof` combined with a condition, we can nest to any depth without losing legibility.

```
reward ( "nested anyOf and allOf conditions" ) {
    anyOf {
        allof {
            condition {
                true
            }
            condition {
                false
            }
        }
        condition {
            false
        }
    }
}
```

```
    }
    anyOf {
        condition {
            false
        }
        condition {
            true
        }
    }
}
assert result == true
}
```

Handling events: deferred execution

Running this particular DSL from the command line is not particularly useful. In order to be useful, our rewards DSL will eventually need to be hooked into the runtime of our Broadband provider's back-end applications. Rewards need to be applied as the result of different events in these systems. Some rewards will be applied at the point of consumption of the individual media. Other rewards will be applied at purchase time, and others when significant account events occur, such as upgrading from one plan to another.

Placing all of the rewards in a single DSL script means that they will all be executed in sequence when we load and evaluate the script. We could decide to separate the rewards into three script files, one for each type of event. On each event being triggered in the application, we would load and execute the appropriate reward script; for a consumption event, we load the `onConsume` script; and so on. This will work, but considering the amount of loading and parsing required for each event, it could become a drag on performance.

There is an alternative approach that will allow us to keep all of the rewards in a single script and also allow us to load the full script only once. Earlier on in this chapter, we talked about using closures as a singleton. In other words, having a single instance of a named closure when the user supplies the implementation. We can partition the rewards script into three possible sections by using this pattern.

```
onConsume = {
    reward {
        ...
    }
    ...
}
```

```
onPurchase = {
    reward {
        ...
    }
    ...
}
onUpgrade = {
    ...
}
```

By using this structure for the DSL, the loading and evaluating of the script simply causes the three closure assignments to be executed. On completion of evaluation of the script, there will be three closure variables in the binding, which contain the reward logic. This small change to the structure means that we can now load and evaluate the script just once on initialization, and invoke the specific rewards for consume, purchase, and upgrade as required.

Here we see how we can use the `GroovyShell` to defer execution of a closure block:

```
def binding = new Binding()
binding.saved = {

binding.deferred = { closure ->
    closure.delegate = delegate
    closure()
}

def shell = new GroovyShell(binding)
shell.evaluate(
"""
saved = {
    println "saved"
    deferred {
        println "deferred"
    }
}
""")
storeSaved = binding.saved
```

In the preceding snippet, the `evaluate` method of the DSL script will set `saved` to be the closure that is provided within the script. We can store this in a closure variable to be reused later. Evaluating the saved closure in the `GroovyShell` once again gives us the opportunity to set up the binding to support our DSL. Prior to calling `saved()` in the evaluated script, we need to set its delegate. This ensures that the saved closure inherits the binding that we pass into the script.

```
def binding = new Binding()

binding.saved = store_saved
binding.deferred = { closure ->
    closure.delegate = delegate
    closure()
}

def shell = new GroovyShell(binding)
shell.evaluate("saved.delegate = this;saved()")
```

We now have a means of stripping out the closures defined within the main script and executing them at will.

Convenience methods and shorthand

The final piece of our DSL that we need to take care of is to add some convenience methods and shorthand binding variables, in order to make the DSL more legible. First, we add some actions to be taken when a reward is granted. In the case of our rewards DSL, the actions that we want for now are the ability to extend access to a video or game, or to add points to a subscriber's account.

In reality, we would implement many more of these, including granting access to specific videos or games. The features that we can add to the DSL are limited only by our imaginations and the features provided in the back-end systems that we are working with. Most likely our DSL would evolve over time, with new action verbs and other shorthand features being added.

```
binding.extend = { days ->
    def bbPlus = new BroadbandPlus()
    bbPlus.extend( binding.account, binding.media, days)
}
binding.points = { points ->
    binding.account.points += points
}
```

We implement the extend action through a method call on the `BroadbandPlus` service. The `points` action is just a shorthand way of updating the account points value that is always in the binding. In this way we can extend the vocabulary of the DSL to include any actions that we might like to perform on the system.

We further improve the legibility of our scripts by adding some shorthand to be used in conditionals. Common tests that we encounter when deciding to grant a reward are the type of media being consumed or purchased, and whether it is a new release or not. We add some shorthand to the DSL by including boolean binding variables for these common conditions.

```
binding.is_new_release = media.newRelease
binding.is_video = (media.type == "VIDEO")
```

The Offers

Putting all of these DSL features together, we now have a mini DSL that will allow us to define how rewards should be granted to subscribers based on their consumption and purchasing behavior. Earlier in the chapter, we listed some reward types that we would like to support. Let's see now how well we can express those rewards by using the DSL that we have just designed. If we've done our job well, the reward DSL should be all we need to read to fully understand the intent and impact of applying the reward.

```
onConsume = {
    reward ( "Watch a Pixar Movie, get 25% extra points." ) {
        allOf {
            condition {
                media.publisher == "Disney"
            }
            condition {
                isVideo
            }
        }
    }

    grant {
        points media.points / 4
    }
}

reward ( "Rent a new release, get extra night rental" ) {
    condition {
        isNewRelease
    }
}

grant {
    extend 1
}
```

```
        }
    }
}

onPurchase = {
    reward ( "Earn 10% bonus points on all games." ) {
        condition {
            isGame
        }
        grant {
            points media.points / 10
        }
    }
}

onUpgrade = {
    reward ("Upgrade to PLUS and get 100 free points") {
        condition {
            toPlan == "PLUS"
        }
        grant {
            points 100
        }
    }
}
```

The RewardService class

We now have a very usable DSL design that can express the rewards as a script. All that remains is to implement the means to integrate this DSL into our application. It makes sense to package all of this functionality into a service class that can be called by our application when needed. To do this, we provide a class called `RewardService`.

The `RewardService` class provides a static method, `loadRewardRules`, that needs to be called first, in order to initialize the rewards. This method takes care of the initial loading of the rewards from the script file. Initial default implementations of the `onConsume`, `onPurchase`, and `onUpgrade` closures are provided by the `RewardService` class. Their only purpose is to provide a stub, which will be called if any of these closures has not been provided by the DSL. Once loaded, the `RewardService` maintains static copies of the closure, to be called as needed by the event hook methods.

RewardService provides three event hook methods: `applyRewardsOnConsume`, `applyRewardsOnPurchase`, and `applyRewardsOnUpgrade`. These hook methods are to be called in response to consume, purchase, and upgrade events in the system. The hook methods take care of preparing the binding with the necessary closures and binding variables, before performing a deferred invocation of the `onConsume`, `onPurchase`, or `onUpgrade` closure that was stored earlier. The convenience methods `prepareClosures` and `prepareMedia` set up some of the common binding variables, which implement our built-in action methods and other boolean shorthands.

```
class RewardService {
    static boolean on_consume_provided = true
    def static onConsume = {
        on_consume_provided = false
    }
    static boolean on_purchase_provided = true
    def static onPurchase = {
        on_purchase_provided = false
    }
    static boolean on_upgrade_provided = true
    def static onUpgrade = {
        on_upgrade_provided = false
    }

    void prepareClosures (Binding binding) {
        binding.onConsume = onConsume
        binding.onPurchase = onPurchase
        binding.onUpgrade = onUpgrade
        binding.reward = { spec, closure ->
            closure.delegate = delegate
            binding.result = true
            binding.and = true
            closure()
        }
        binding.condition = { closure ->
            closure.delegate = delegate

            if (binding.and)
                binding.result = (closure() && binding.result)
            else
                binding.result = (closure() || binding.result)
        }

        binding.allOf = { closure ->
            closure.delegate = delegate
            def storeResult = binding.result
```

```

def storeAnd = binding.and
    binding.result = true // Starting premise is true
    binding.and = true

closure()

if (storeAnd) {
    binding.result = (storeResult && binding.result)
} else {
    binding.result = (storeResult || binding.result)
}
binding.and = storeAnd
}

binding.anyOf = { closure ->
    closure.delegate = delegate
def storeResult = binding.result
def storeAnd = binding.and

    binding.result = false // Starting premise is false
    binding.and = false

    closure()
if (storeAnd) {
    binding.result = (storeResult && binding.result)
} else {
    binding.result = (storeResult || binding.result)
}
binding.and = storeAnd
}

binding.grant = { closure ->
    closure.delegate = delegate

    if (binding.result)
        closure()
}
binding.extend = { days ->
    def bbPlus = new BroadbandPlus()
    bbPlus.extend( binding.account, binding.media, days)
}
binding.points = { points ->
    def bbPlus = new BroadbandPlus()
    binding.account.points += points
}

```

```
    }
    void prepareMedia(binding, media) {
        binding.media = media
        binding.is_new_release = media.newRelease
        binding.is_video = (media.type == "VIDEO")
        binding.is_game = (media.type == "GAME")
        binding.is_song = (media.type == "SONG")
    }
    static void loadRewardRules() {
        Binding binding = new Binding()

        binding.onConsume = onConsume
        binding.onPurchase = onPurchase
        binding.onUpgrade = onUpgrade

        GroovyShell shell = new GroovyShell(binding)
        shell.evaluate(new File("rewards/rewards.groovy"))

        onConsume = binding.onConsume
        onPurchase = binding.onPurchase
        onUpgrade = binding.onUpgrade
    }
    void applyRewardsOnConsume(account, media) {
        if (on_consume_provided) {
            Binding binding = new Binding()
            binding.account = account
            prepareClosures(binding)
            prepareMedia(binding, media)

            GroovyShell shell = new GroovyShell(binding)
            shell.evaluate("on_consume.delegate = this;onConsume()")
        }
    }
    void applyRewardsOnPurchase(account, media) {
        if (on_purchase_provided) {
            Binding binding = new Binding()
            binding.account = account
            prepareClosures(binding)
            prepareMedia(binding, media)

            GroovyShell shell = new GroovyShell(binding)
            shell.evaluate("on_purchase.delegate = this;onPurchase()")
        }
    }
    void applyRewardsOnUpgrade(account, plan) {
```

```

        if (on_upgrade_provided) {
            Binding binding = new Binding()
            binding.account = account
            binding.to_plan = plan
            binding.from_plan = account.plan
            prepareClosures(binding)

            GroovyShell shell = new GroovyShell(binding)
            shell.evaluate("on_upgrade.delegate = this;onUpgrade()")
        }
    }
}

```

The BroadbandPlus application classes

In order to show this DSL working, we need to flesh out some classes in order to implement a rudimentary application skeleton for our imaginary BroadbandPlus service. We won't scrutinize these classes in too much detail, as their main purpose is to provide the hooks to exercise our DSL, and not to represent a working system.

To begin with, we need to define an `Account` class. The `Account` class maintains the basic subscription details for a subscriber, including the plan he is on and his remaining points for this period. It also maintains the current list of media that he has access to. Once the consumption of an item starts, the media is added to this list, along with an expiry date. The expiry date can be extended at any time by calling the `extendMedia` method.

```

class Account {
    String subscriber
    String plan
    int points
    double spend
    Map mediaList = [:]
    void addMedia (media, expiry) {
        mediaList[media] = expiry
    }
    void extendMedia(media, length) {
        mediaList[media] += length
    }
    Date getMediaExpiry(media) {
        if(mediaList[media] != null) {
            return mediaList[media]
        }
    }
}

```

The Media class is used to describe individual items from the media catalog. Properties of the class define its price and access points value, along with other properties that help to categorize it, such as the media type (VIDEO, GAME, or SONG), the publisher, and whether it is a new release or not.

```
class Media {  
    String title  
    String publisher  
    String type  
    boolean newRelease  
    int points  
    double price  
    int daysAccess  
}
```

The BroadbandPlus class implements the backend services that we expect, and defines the APIs that we need to manage the consumption of media, purchasing, and account upgrades. These APIs make calls to the RewardService as required, in order to apply the various rewards for onConsume, onPurchase, and onUpgrade.

The consume API will add the consumed media to the account's media list on the first consumption. The purchase API adds the points value of the purchased media to the account's points balance. Upgrade takes the current period balance into account by simply adding the points difference between the original and upgrade plans.

```
class BroadbandPlus {  
    def rewards = new RewardService()  
  
    def canConsume = { account, media ->  
        def now = new Date()  
        if (account.mediaList[media] ?.after(now) )  
            return true  
  
        account.points > media.points  
    }  
    def consume = { account, media ->  
        // First consume add media to accounts access list  
        if (account.mediaList[media.title] == null) {  
            def now = new Date()  
            account.points -= media.points  
            account.mediaList[media] = now + media.daysAccess  
            // Rewards only applied on first consumption  
            rewards.applyRewardsOnConsume(account, media)  
        }  
    }  
}
```

```

def purchase = { account, media ->
    rewards.applyRewardsOnPurchase(account, media)
    account.points += media.points
    account.spend += media.price
}
def upgrade = { account, newPlan ->
    if (account.plan == "BASIC" && newPlan == "PLUS")
        account.points += 130
    if (account.plan == "BASIC" && newPlan == "PREMIUM")
        account.points += 430
    if (account.plan == "PLUS" && newPlan == "PREMIUM")
        account.points += 300

    rewards.applyRewardsOnUpgrade(account, newPlan)
    account.plan = newPlan
}
def extend = {account, media, days ->
    if (account.mediaList[media] != null)  {
        account.mediaList[media] += days
    }
}
}
}

```

Testing with GroovyTestCase

Finally, we can test to see if our reward scripts are being triggered as expected, by writing a `GroovyTestCase` for it. `GroovyTestCase` is the Groovy implementation of JUnit, and comes packaged with the Groovy libraries. In order to make use of it, all that we need to do is to write individual tests for the features that we want to validate. Here we can verify that each individual reward that we have defined is being triggered. We do this by setting up consumption, purchase, and upgrade scenarios that we expect to trigger the reward.

In the `setUp` method, we create an `account` object with a `BASIC` plan. We set up four different `media` objects, and load the reward rules. We can assert that the outcome was as expected. For instance, in the first test we consume a Disney video and assert that the bonus points have been added. For completeness, we then consume a non-Disney video and see that no bonus points have been added.

```

class TestConsume extends GroovyTestCase {
    def account
    def up
    def terminator
    def halo3

```

```
def halo1
def bbPlus

void setUp() {
    account = new Account(plan:"BASIC", points:120, spend:0.0)
    up = new Media(title:"UP", type:"VIDEO", newRelease:true,
                   price:3.99, points:40, daysAccess:1,
                   publisher:"Disney")
    terminator = new Media(title:"Terminator", type:"VIDEO",
                           newRelease:false, price:2.99, points:30,
                           daysAccess:1, publisher:"Fox")
    halo3 = new Media(title:"Halo III", type:"GAME",
                      newRelease:true, price:2.99, points:30,
                      daysAccess:3, publisher:"Microsoft")
    halo1 = new Media(title:"Halo", type:"GAME",
                      newRelease:false, price:1.99, points:20,
                      daysAccess:3,publisher:"Microsoft")
    bbPlus = new BroadbandPlus()
    RewardService.loadRewardRules()
}
void testDisneyReward() {
    assert bbPlus.canConsume( account, up)
    assert account.points == 120
    def expected = account.points - up.points + up.points/4
    bbPlus.consume(account, up)
    assert account.points == expected
    bbPlus.consume(account, terminator)
    assert account.points == expected - terminator.points
}
void testExtensionReward() {
    bbPlus.consume(account, up)
    bbPlus.consume(account, terminator)
    def now = new Date()
    // Extension applied to Up but not Terminator
    assert account.getMediaExpiry(up).after(now + 1)
    assert account.getMediaExpiry(
                    terminator).after(now + 1) == false
}
void testPurchaseRewardOnGames() {
    assert account.points == 120
    bbPlus.purchase(account, terminator)
    bbPlus.consume(account, terminator)
    assert account.points == 120
    bbPlus.purchase(account, halo1)
```

```

        bbPlus.consume(account,    hal01)
        assert account.points == 122
    }
    void testUpgradeToPlusReward() {
        assert account.points == 120
        bbPlus.upgrade(account,    "PLUS")
        // Should have 250 for PLUS and 100 bonus
        assert account.points == 350
        bbPlus.upgrade(account,    "PREMIUM")
        // Should have 550 for PREMIUM and 100 bonus
        // from the previous upgrade
        println account.points
        assert account.points == 650
    }
    void testUpgradeToPremiumReward() {
        assert account.points == 120
        bbPlus.upgrade(account,    "PREMIUM")
        // Should have 550 for PREMIUM and 100 bonus
        println account.points
        assert account.points == 650
    }
}

```

Running this test case should pass all of the tests. This would verify that all of the rewards that we have deployed in the DSL are being activated as expected. However, the final `testUpgradeToPremiumReward` fails. This reveals a flaw in our reward logic. The conditions that we have used allow a bonus for upgrading from BASIC to PLUS. If the subscriber then upgrades to PREMIUM, they keep the bonus points. However, a subscriber upgrading from BASIC straight to PREMIUM is disadvantaged by not receiving the bonus, which was not our intention.

```

onUpgrade = {
    reward ("Upgrade to PLUS and get 100 free points") {
        anyOf {
            condition {
                toPlan == "PLUS"
            }
            allOf {
                condition {
                    toPlan == "PREMIUM"
                }
                condition {
                    fromPlan == "BASIC"
                }
            }
        }
    }
}

```

```
        }
    }
    grant {
        points 100
    }
}
}
```

Changing the `onConsume` reward script fixes this anomaly. Running our tests again will show them all passing as expected. So now we have a rewards DSL script with a service class that implements it. We've hooked the rewards service into the domain service so that it is triggered on the important events within the back-end services. We've tested out the rewards by using some test cases. We should now have a good degree of confidence that if `BroadbandPlus` was a real application service, our reward programs would be getting called at the appropriate times.

Summary

We covered a lot of ground in this chapter. We took a look at Groovy bindings to see how they can be used in our DSL scripts. By placing closures strategically in the binding, we can emulate named blocks of code. We can also provide built-in methods and other shorthand by including closures and named Boolean values in the binding. These techniques can be used to great effect to write DSL scripts that can be read and understood by stakeholders outside of the programming audience.

We've worked through a full implementation of a DSL for customer rewards by using these techniques, and we've seen how such a DSL can be integrated into an existing application. The reader should now have the confidence to start generating their own domain-specific DSLs that implement features in a similar way and integrate them into their own applications.

In the next chapter, we will look in more detail at the issues involved in integrating DSL scripts into our Java applications. The example here used pure Groovy, but what if the application that we wanted to add was Java-based? Would we still be able to use a Groovy DSL?

9

Integrating it all

In this final chapter, we are going to look at all of the different ways in which we can integrate Groovy with Java. All of our examples in the book to date have been written in pure Groovy. We now want to look at ways of leveraging Groovy within our existing Java applications.

One of the biggest advantages of Groovy for Java developers is its close relationship with the Java language itself. There are many ways in which we can exploit this relationship for our benefit. We will be looking at all of the following techniques:

- We will explore how to integrate the languages at the class level in order to call Groovy code from Java and to call Java code from Groovy.
- We will take a look at some of the dependency issues that we face when integrating the two languages, and will see how dependency injection frameworks like Spring can help us overcome these issues.
- We will look at the `GroovyClassLoader` as a means of accessing Groovy classes that have already been compiled to bytecode.
- We will revisit the `GroovyShell` class and look at `GroovyScriptEngine` as a means of directly invoking and parsing Groovy scripts from our Java code.

Mixing and matching Groovy and Java

When we integrate Groovy and Java code, one concern that we have is the level of integration that we want to achieve. Groovy compiles to Java bytecode, and the resulting classes are indistinguishable from classes written in Java. With this in mind, we could simply take the approach that our application is written in both languages, and mix and match classes from Groovy and Java as needs arise.

Need the power of a builder? Why not write a class in Groovy that implements a Java interface from our application? We can compile the class with `groovyc` and package it up in a JAR with all of the Java classes. No one will be any the wiser. A tight level of integration with our Java classes suits this type of usage. We can use Ant, Maven, or some other build tool – preferably Groovy-based tools, such as Gant and Gradle – to manage the compilation of the different language elements and the final packaging of the different elements into one complete application.

Most of us starting out with Groovy will build some standalone projects alongside our main applications. As a scripting language, Groovy is ideal for building developer tools such as Gant and EasyB. Not many organizations are brave enough to plunge straight into using Groovy as a part of their core application development. If you are lucky enough to be starting afresh with a green field project, the Grails framework is a great alternative to developing with Java Enterprise Edition JEE.

However, most of us face the reality of a significant existing investment in Java frameworks and technologies. So we need to be able to gently introduce new technologies into our existing environments. What are our options when looking to introduce a little Groovy into our application? Typically, we will be looking to apply the new language to an area that will least impact our existing code. The ideal scenario would be where some or all of the Groovy elements are considered to be outside of the "core" application to begin with. For instance, we might be using Groovy to implement the integration gateway to a third-party application, or to add some new business rules to the application. Many of these incremental improvements to your application might well be suitable for including a custom DSL for configuration or for defining new business rules.

In the last chapter, we built a DSL for applying rewards to user accounts based on purchasing behavior. You could have a similar need for DSL scripts that might arise. Maybe you would like to open up some elements of the application to be developed in collaboration with domain experts who are not professional programmers. If so, you could consider designing a DSL that can be integrated and made a part of the deployed application. Such requirements are an ideal place to start your Groovy introduction while also using the power of Groovy to include some DSLs.

Regardless of where you start, you are likely to have a wide variety of places where Groovy code might need to be integrated into your core applications. To go with this variety of sources, we have various levels of trust that we can place on the Groovy code that we integrate. In most cases, we are prepared to fully trust this code as it will be developed by the same teams and individuals who develop the main application, and is a part of the core application. We are happy to intimately integrate this Groovy code at the class level, mixing and matching Java and Groovy in our regular builds. With other Groovy code, such as DSLs that are developed outside of the core, we will want to dynamically load the code and keep it at arm's length, due to the potentially untrusted nature of its origins.

Fortunately, Groovy has many different mechanisms for integrating the two languages. We will cover them all, and see how we can use them to solve our specific integration needs.

Calling Groovy from Java

The simplest way to integrate Groovy and Java is to simply compile both to .class files, and package them together. So long as all the compiled classes can be resolved in the same classpath, the Java and Groovy code can coexist without any special treatment. This is the most intimate level of integration, and is suitable for circumstances where the Groovy code is a trusted a part of the application.

POJOs and POGOs

Groovy automatically generates getters and setters for any class that is compiled, so that the resulting class can be called from a Java application as if it was a normal Java bean— or as they have become known, a **Plain Old Java Object** (POJO). Take a simple Groovy class with two string fields. Groovy will generate the `setFirstName()`, `getFirstName()`, `setLastName()`, and `getLastName()` methods for us. The Java class file that is generated when we compile this to bytecode will therefore have these methods compiled into the bytecode. We refer to these classes as **Plain Old Groovy Objects** (POGOs).

```
class POGO {  
    String firstName  
    String lastName  
}  
  
def POGO pogo = new POGO()  
pogo.setFirstName("Fred")  
assert pogo.getFirstName() == pogo.firstName
```

So the simple Groovy class shown above is interchangeable at runtime with the more verbose Java version of the same object. This interchangeability means that wherever we are used to making use of a POJO in our Java code, we can integrate a Groovy class and it will act the same as if it were a POJO. This means that libraries such as Commons BeansUtils will operate happily on our Groovy POGOs just like they do with our POJO code, but with a little less coding. Here we see the same four-line POGO implemented in Java as a POJO.

```
public class POJO {  
    String firstName;  
    String lastName;
```

```
public String getFirstName() {
    return firstName;
}
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
public String getLastName() {
    return lastName;
}
public void setLastName(String lastName) {
    this.lastName = lastName;
}
}
```

What this means is that by placing both of these classes in the classpath of our application, we can use them both transparently as if both were originally written in Java. So a client application need not know what language the original class was written in to be able to use it. Here we see how both classes are transparently accessible from a Java client program:

```
public class CallGroovyFromJava {

    public static void main(String[] args) {
        POGO groovyObject = new POGO();

        groovyObject.setFirstName("Fred");
        groovyObject.setLastName("Flintstone");

        System.out.println("Pogo : " + groovyObject.getFirstName() );

        POJO javaObject = new POJO();

        javaObject.setFirstName("Barney");
        javaObject.setLastName("Rubble");

        System.out.println("POJO : " + javaObject.getFirstName());
    }
}
```

The Groovy compiler has the useful ability to compile both Java and Groovy from a single command. Assuming that all three of the above class files are in our current directory, we can compile them all at once by using the `-j jointCompilation` option. The `-j` option instructs `groovyc` to invoke `javac` whenever a `.java` file is encountered. Here we use a single compile command to compile both the `POGO`, `groovy` and `POJO`.`java` files:

```
$ groovyc * -j -Jclasspath=$GROOVY_HOME/embeddable/groovy-all-1.6.0.jar:.  
$ java -cp $GROOVY_HOME/embeddable/groovy-all-1.6.0.jar:.  
CallGroovyFromJava  
Pogo : Fred  
POJO : Barney  
$
```

Apart from being convenient to compile everything in one go, the joint compiler has an additional benefit in resolving dependency issues. In the above example, if we tried to compile the Java and Groovy separately, we would get a compile error when first compiling `CallGroovyFromJava.java`. The `javac` complier knows to seek out and compile any additional dependent classes that it encounters. Unfortunately, it does not know about Groovy files, so compiling this file first will give us an error:

```
$ javac CallGroovyFromJava.java  
CallGroovyFromJava.java:4: cannot find symbol  
symbol  : class POGO  
location: class CallGroovyFromJava  
    POGO groovyObject = new POGO();  
           ^  
  
CallGroovyFromJava.java:4: cannot find symbol  
symbol  : class POGO  
location: class CallGroovyFromJava  
    POGO groovyObject = new POGO();  
           ^  
  
2 errors  
$
```

We can work around this of course by compiling our Groovy sources first. However, the handiest option by far is to allow the joint compiler to figure it out for us.

Calling Java from Groovy

We can call Groovy classes directly from Java, and similarly we can call Java classes directly from Groovy once all of the classes are in the same classpath. Here we can see how the Groovy client code is able to make a property style reference to `firstName` on the POJO object. Any existing Java bean can be imported into Groovy and used as if it also was a Groovy object.

```
class CallJavaFromGroovy {
    static void main(args) {
        def groovyObject = new POGO(firstName:"Fred",
                                lastName:"Flintstone")
        def javaObject = new POJO(firstName:"Barney",
                                lastName:"Rubble")

        println groovyObject.firstName
        println javaObject.firstName
    }
}
```

We can also write this as a Groovy script without the surrounding class, as shown in the next code snippet. As long as the Groovy and Java classes are on the classpath, they will be treated the same.

```
def groovyObject = new POGO(firstName:"Fred", lastName:"Flintstone")
def javaObject = new POJO(firstName:"Barney", lastName:"Rubble")

println groovyObject.firstName
println javaObject.firstName
```

Here we can see a useful side effect of how Groovy implements the named parameter shortcut. Although the parameters `firstName` and `lastName` appear to be named parameters, they are in fact elements of a Map object passed to a built-in constructor that takes a Map parameter. Each element in the map is interrogated and the equivalent setter is called in the construction process. Even though the original Java POJO class does not implement this Map constructor, the Java class, when used in Groovy code, gets wrapped as a Groovy object, and the Map constructor is available for use. We can pass named parameters into a Java object even though no such constructor has been implemented in the original Java class.

Privacy concerns

One point to note when mixing and matching Groovy and Java is Groovy's lax attitude to privacy. When we write DSL scripts, we will generally want to locate our scripts in the default package and, ignoring package structures, will tend to de-clutter our code and make it more readable. However, when we do this we can start to notice subtle differences between how Groovy and Java handle privacy. Here we modify the POGO class so that there is a private getter and setter for lastName.

```
class PrivateIgnored {  
    String firstName  
    String lastName  
  
    private String getLastName() {  
        return lastName  
    }  
    private void setLastName(String last) {  
        lastName = last  
    }  
}
```

Applying `private` to the Groovy property getters and setters has different subtle effects on Groovy and Java clients. With the above version of the POGO, we can still access `lastName` from a Groovy client by referencing `privateIgnored.lastName`. On the other hand, a Java client will report that `private.getLastName()` has private access.

Although the `private` modifier is ignored by any client code written in Groovy, the modifier does make its way into the compiled class, so an equivalent Java client will not be able to access it. If the intention is to hide getters and setters from Java altogether, then a better way to achieve this is to apply the `private` modifier to the field itself in the Groovy class.

```
class PrivateAccess {  
    String firstName  
    private String lastName  
}
```

This is subtly different from the first class insofar as it suppresses the generation of getter and setter for `lastName` entirely. In effect, this has little impact on either the Groovy or Java code. We can still use the subscript operator in Groovy to access `lastName`. As this is the preferred access method on Groovy, it won't impact our code much. However, attempting to call the getter or setter directly will cause a `MissingMethodException` to be thrown.

```
def privateIgnored = new PrivateIgnored(firstName:'Fred',
lastName:'Flintstone')
privateIgnored.firstName = 'Fred'
privateIgnored.setLastName('Rubble')
println "${privateIgnored.firstName} ${privateIgnored.lastName}"

def privateAccess = new PrivateAccess()

privateAccess.lastName = "Flintstone"
try {
privateAccess.getLastName() // MissingMethodException
} catch (e) {
    println e
}
```

An equivalent Java program trying to access `lastName` in either class will display compile errors. Attempting to access `PrivateIgnored.setLastName` will cause a "`setLastName(java.lang.String)` has private access in `PrivateIgnored`" compilation failure, whereas attempting to access `PrivateAccess.setLastName` will cause a "cannot find symbol" compilation failure for the method.

Interfaces in Java and Groovy

Interfaces are interchangeable in the same way as Groovy and Java classes. We can either declare an interface in Java and base a Groovy class on it, or we can declare the interface in Groovy and base a Java class on it. This is a tremendously useful feature, as it allows us to exploit standard interfaces from our existing Java patterns, and seamlessly start to introduce classes, which are now written in Groovy instead of Java into an existing application.

```
// Interface declared in Java
public interface JavaInterface {
    void serviceApi(String param);
}

// Service in Groovy implements Java interface
class GroovyService implements JavaInterface {
```

```
void serviceApi(String caller) {
    println caller +
        " calling Groovy implementation of Java Interface"
}

// Interface declared in Groovy
interface GroovyInterface {
    void serviceApi(String param)
}

// Service in Java implements Groovy interface
public class JavaService implements GroovyInterface{
    public void serviceApi(String caller) {
        System.out.println(caller +
            " calling Java implementation of Groovy Interface");
    }
}

// Client code written in Java does not care what language the
// Service or interface was implemented in
public class ServiceClient {
    public static void main(String[] args) {
        GroovyService groovyService = new GroovyService();
        groovyService.serviceApi("Java client");

        JavaService javaService = new JavaService();
        javaService.serviceApi("Java client");
    }
}
```

In our previous chapter, we built a `RewardService` class. This implemented the runtime hooks for our Rewards DSL. In that chapter, we integrated this into a dummy Groovy application. However, there should be no reason why the `RewardService` class cannot be called from Java. The service class can just as easily be integrated into an existing Java application.

The RewardService class implements a Rewards DSL through the extensive use of closures. The APIs that it provides to a client application, on the other hand, are all regular Java-style methods. To a Java application, it will appear that there is another Java-based service class called RewardService that implements the APIs such as `applyRewardsOnConsume()`. Although the closures are not visible to Java, the important APIs are, so we should be able to make use of RewardService directly from within a Java application.

A version of RewardService designed to work with an existing Java application will differ from the Groovy version insofar as it presumably would need to work with pre-existing Java domain classes. This is not an issue, as we've seen that Groovy is agnostic as to whether these domain classes were implemented in Java or Groovy. We can imagine Java versions of the `Account` and `Media` domain classes that are not much different from their Groovy counterparts.

```
// Groovy
class Media {
    String title
    String publisher
    String type
    boolean newRelease
    int points
    double price
    int daysAccess
}

// Java equivalent needs to be implemented as a JavaBean
// so default constructor and getters/ setters are required.
public class Media {
    private String title = null;
    private String publisher = null;
    .....

    Media () {
        title = "";
        publisher = "";
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getPublisher() {
```

```
        return publisher;
    }
    public void setPublisher(String publisher) {
        this.publisher = publisher;
    }
    ...
}
```

Integrating the `RewardService` DSL into our application can therefore generally be achieved with minimal fuss. As long as the compile time dependencies are resolved and we package `RewardService` into the JARs of the existing application, the Java application will happily call `RewardService` APIs as if they are native Java methods, and `RewardService` can interrogate the Java domain classes as if they are native Groovy classes. Unfortunately, this does raise one awkward gotcha: when integrating Java and Groovy, resolving dependencies can sometimes be a chicken and egg situation.

Resolving dependencies

The `RewardService` example is a typical scenario that we will come across when integrating the two languages. Once compiled and packaged into the application JARs, the Java and Groovy elements of this application will work seamlessly together. The problem arises at compile time. The Groovy class `RewardService` class will happily resolve the use of a Java `Account` class once it is made available to it as a class file via the classpath.

However, the Groovy compiler does not have any knowledge of how to deal with the class if it is only available to it in source form. The `groovyc` and `javac` compilers will automatically compile any classes encountered in the source as long as the compiler can find the corresponding source files in the path.

We saw earlier in the chapter how the Groovy joint compiler can help here by figuring out the dependencies and calling the correct compiler based on whether it encounters a Java class or a Groovy class. This is fine if the application is compartmentalized enough that we can compile the entire source in one step. The Grails project has a mixture of Groovy and Java sources, which are all compiled together in one step by using `groovyc -j`. This may not be feasible for many medium-to-large Java applications where there can be hundreds, if not thousands, of classes to compile across a range of components. If we are introducing Groovy into an existing development environment, we don't have to tell all of our sister teams that they now need to switch compilers.

A typical Ant or Maven build script that is used to build an application might have separate compile targets for the various subcomponents of the application. Into this we want to introduce a new compile target that will compile the Groovy code for the Reward service classes. Here we hit a potential problem. We have application classes in Java that make use of the RewardService class and therefore reference it. We can't compile these application classes without first compiling RewardService. The RewardService class, on the other hand, makes reference to the domain classes, so these classes need to be compiled before the RewardService class is. Depending on how we have structured our compile targets in the build scripts, we could have a catch-22 on our hands due to a circular dependency.

Fortunately in this case, all that is required is a refactoring of the build targets. Separating out the compilation into three separate targets will fix the problem. We will need one compile target for the domain classes, which needs to be the first step in the compilation process. Next we need a compile target, which compiles the RewardsService with groovyc, which is called in sequence after the domain classes. Finally, we need a third target for the application service classes. The build script needs to ensure that the targets are compiled in this order, and then everything will work.

Here, we can resolve the issue by changing the build targets to accommodate this. However, there will always be circumstances where a refactoring of the code will be the only viable solution. When working with legacy code, the safest approach will often be to have your Groovy classes implement a Java interface. Consider a minor reworking of RewardService. The Java application needs to call out certain RewardService APIs when consumption, purchase, and upgrade events occur in the application. We can separate this out into a service interface.

```
// Service interface in Java
public interface RewardInterface {
    void applyRewardsOnConsume(Account account, Media media);
    void applyRewardsOnPurchase(Account account, Media media);
    void applyRewardsOnUpgrade(Account account, String plan);
}

// Groovy RewardService implements Java interface
class RewardService implements RewardInterface {
...
}
```

All references to the RewardService in the Java application are now made through the Java interface. The existing Java compile targets will encounter references to RewardInterface, and the javac compiler will know how to deal with it. To ensure that we don't make any direct reference to RewardService, we can use the factory pattern to create concrete instances of the RewardService class and store them as RewardInterface instances.

Dependency injection with Spring

By far the best way to resolve these types of application dependencies is with a dependency injection framework such as Spring. Using Spring's dependency injection framework, we no longer need to worry about wiring together the various application dependencies. Spring will take care of all of this for us. Let's take a look at a simplified BroadBandPlus service and how we might use Spring to resolve the dependencies between it and our Groovy-based RewardService.

```
package broadbandplus;

public class BroadbandPlus {
    private RewardInterface rewardService;

    public BroadbandPlus(RewardInterface rewardService) {
        this.rewardService = rewardService;
        rewardService.init();
    }

    void purchase (Account account, Media media) {
        rewardService.applyRewardsOnPurchase(account, media);
        account.setPoints( account.getPoints() - media.getPoints());
    }
}
```

The BroadbandPlus service class implements a purchase API, which has a dependency on the RewardService implementation. Within the constructor, we initialize the rewardService instance by calling init, which will load our Reward DSL scripts. This example lacks any of the normal initialization and lookup code that we might expect to be required in order to initialize the rewardService instance. We'll see in a bit how Spring takes care of this for us.

As suggested in the previous section, we have separated the RewardService into an interface and an implementation. In this version, we have implemented the interface in Java so that it can be easily shared between our BroadbandPlus code in Java and the RewardService implementation in Groovy.

```
package broadbandplus;

public interface RewardInterface {
    void applyRewardsOnConsume(Account account, Media media);
    void applyRewardsOnPurchase(Account account, Media media);
    void applyRewardsOnUpgrade(Account account, String plan);
    void init();
}
```

```
package broadbandplus

class RewardService implements RewardInterface {
    ...
    void init() {
        loadRewardRules()
    }

    void loadRewardRules() {
        Binding binding = new Binding()

        binding.on_consume = on_consume
        binding.on_purchase = on_purchase
        binding.on_upgrade = on_upgrade

        GroovyShell shell = new GroovyShell(binding)
        shell.evaluate(new File("offers/Offer.groovy"))

        on_consume = binding.on_consume
        on_purchase = binding.on_purchase
        on_upgrade = binding.on_upgrade
        println "Rules loaded"
    }
    ...
}
```

The only changes required for `RewardService` from our previous version is to have it implement our `RewardInterface`, and to make a call to `loadRewardRules` from `init`. The `BroadbandPlus` service class only ever makes use of `RewardInterface`. So as far as it is concerned, `RewardService` could equally well be implemented in Java. The fact that it is a Groovy class making use of Groovy DSLs makes no difference to it.

This also makes `BroadbandPlus` service an ideal candidate for using Spring to inject its dependencies. Spring defines all of its application dependencies in the **application context**. The application context is in effect a sophisticated factory mechanism that allows us to externalize all of the inter object wiring from our code. The application context is configured in an XML file usually called `beans.xml` or `application-context.xml`. Below is a `beans.xml` file for our `BroadbandPlus` application.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:
xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:lang="http://www.springframework.org/schema/lang"
      xsi:schemaLocation="
```

```
http://www.springframework.org/schema/beans http://www.
springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/lang http://www.springframework.
org/schema/lang/spring-lang-2.0.xsd">
<bean id="rewardService" class="broadbandplus.RewardService">
</bean>
<bean id="bbPlus" class="broadbandplus.BroadbandPlus">
    <constructor-arg>
        <ref bean="rewardService" />
    </constructor-arg>
</bean>
</beans>
```

In this application context, we define two beans and specify how they should be handled. The rewardService bean has no special configuration information other than to let the application context know what its class name is. The bbPlus bean has some additional configuration details, namely when we tell it to pass an instance of RewardService to its constructor whenever it creates a new instance of BroadbandPlus for us. Let's see this in action:

```
package broadbandplus;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

import broadbandplus.RewardInterface;

import java.io.File;
import java.util.Arrays;

public class BroadbandPlusApp {

    public static void main(String[] args) {
        try {
            ApplicationContext ctx = new
                ClassPathXmlApplicationContext("broadbandplus/beans.xml");
            BroadbandPlus bbPlus = (BroadbandPlus) ctx.getBean("bbPlus");

            Account account = new Account("BASIC", 120);
            Media media = new Media("UP", "Disney", 40, "VIDEO", true);
            Media media2 = new
                Media("Halo", "Microsoft", 40, "GAME", false);
```

```
        System.out.println("Account points starting : " +
                           account.getPoints());
        bbPlus.purchase(account, media);
        System.out.println("Account points after UP : " +
                           account.getPoints());
        bbPlus.purchase(account, media2);
        System.out.println("Account points after Halo : " +
                           account.getPoints());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

All that we need to do to create an instance of BroadbandPlus, with its dependencies to RewardService resolved, is to ask the Spring application context to load it for us. We do this by creating a Spring `ClassPathXmlApplicationContext` object and passing it the location of our `beans.xml` file. Spring will now figure out the correct order in which to instantiate our dependent beans. An instance of the `RewardService` class will be created, and this instance will be passed into the constructor for `BroadbandPlus`. What gets returned is an instance of the `BroadbandPlus` class, along with a fully-initialized `RewardService` bean that is ready to use.

GroovyClassLoader

Up to this point in this chapter, we have looked at how we can integrate Groovy, and where the classes are to be found on a regular application classpath. For a more dynamic means of loading our Groovy classes, we can use the `GroovyClassLoader` class. `GroovyClassLoader` is a subclass of the regular Java class loader `java.lang.ClassLoader`, and provides the means to load a Groovy class, or even parse a Groovy source file and instantiate a class object from it. From this we can create an instance of any Groovy object that we have loaded or parsed.

```
import java.io.File;
import groovy.lang.GroovyClassLoader;

public class LoadService {

    public static void main(String[] args) {
        GroovyClassLoader classLoader = new GroovyClassLoader();

        JavaInterface service = null;
        try {
            service = (JavaInterface)classLoader.parseClass(
                new File("GroovyService.groovy")).newInstance();
        } catch (Exception e) {

```

```
        e.printStackTrace();
    }

    service.serviceApi("Class loaded by GroovyClassLoader");
}

}
```



When compiling the examples in the remainder of this chapter, use `groovyc` to compile both the Java and Groovy classes. The `groovyc` compiler will automatically resolve any references to `GroovyClassLoader`, `GroovyObject`, and others that are imported into the example. Otherwise you will need to include the `groovy-all.1.x.x.jar` from your Groovy installation in your classpath when compiling with `javac`. You will still need to include this JAR in your classpath when running the examples with `java`.

Here we are reusing the previous example of a Groovy class `GroovyService` that implements a Java interface. Now instead of compiling the `GroovyService` class and packaging it on our classpath, we can use the `GroovyClassLoader` to parse the class at runtime. Parsing the Groovy source returns a class object for the `GroovyService` class. We instantiate an instance of `GroovyService` by calling the `newInstance` method of `Class`.

Because `GroovyService` implements `JavaInterface`, we can make use of `GroovyService` without having to worry about any compile time dependencies. This allows us to package the Groovy code outside of the main application. An implementation of a DSL along with its supporting Groovy classes could all be in a separate source folder, for instance.

In the previous example, we passed a `File` object to `parseClass`. `GroovyClassLoader` also has other versions of the `parseClass` method that can load the class source from a string or an input stream. With a bit of imagination, we can exploit these to allow DSL-enabling scripts to be stored and retrieved from a database instead of from a file system.

Once we have retrieved a class object by using the `GroovyClassLoader`, we have a number of options for how we treat the new instance that we have created. Depending on how much intimate knowledge of the returned class we want to embed in our code, we can isolate our usage via an interface, just as we did in the previous example. We can make, direct reference, if we like, to the Groovy class in our Java source, secure in the knowledge that the new instance will behave as a normal Java class. So all, the public methods implemented by the class will also be available to us, although this will potentially reintroduce the dependency issues that we discussed previously.

The Groovy objects returned as new instances are regular Java objects. We can call any public methods on them and access any public fields. If we want even more control over the returned instance, a useful feature to exploit is the fact that, under the covers, every Groovy object implements the `groovy.lang.GroovyObject` interface. We came across this in our discussions of metaprogramming in Chapter 5. One of the methods in the `GroovyObject` interface is the `invokeMethod` interface.

By using the `invokeMethod`, we have intimate access to all of the `GroovyService` methods, including closures. Suppose `GroovyService` has a closure called `myClosure` defined for it, as shown here:

```
class GroovyService implements JavaInterface {  
    void serviceApi(String caller) {  
        println caller +  
            " calling Groovy implementation of Java Interface"  
    }  
  
    def myClosure = {  
        println "Closure called"  
    }  
}
```

In Groovy, we can invoke this closure as if it were another method of the class. This closure will not show up in the compiled Java bytecode as a callable method, so it is not directly accessible from Java. We can get around this by using `invokeMethod` from the `GroovyObject` interface, which conveniently allows us to invoke the closure in the same way as any other Groovy method.

```
import java.io.File;  
import groovy.lang.GroovyClassLoader;  
import groovy.lang.GroovyObject;  
  
public class JavaClient {  
    public static void main(String[] args) {  
        GroovyClassLoader classLoader = new GroovyClassLoader();  
        JavaInterface service = null;  
        try {  
            service = (JavaInterface)classLoader.parseClass(  
                new File("GroovyService.groovy")).newInstance();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
  
        // service.myClosure(); // Compile error, no such method
```

```
// Can invokeMethod for myClosure()
GroovyObject groovyObj = (GroovyObject) service;
groovyObj.invokeMethod("myClosure", new Object[0]);
}
}
```

Integrating scripts

We can load any Groovy class and start making use of it from Java by using `GroovyClassLoader`. Once loaded, we can create new instances of the class at will, which is beneficial for the many uses we might make of Groovy, but is limiting when it comes to embedding a Groovy-based DSL.

In order to build most of the DSLs that we have encountered in the book, we need to be able to integrate not just individual Groovy classes but also Groovy scripts into our application. Groovy scripts are subtly different from Groovy classes.

As in Java, we can define individual Groovy classes in their own namesake source files. The compiler `groovyc` will compile these to a class file of the same name, as you might expect. Scripts, as we know, on the other hand, can have multiple classes defined within them, and may also contain Groovy statements outside of any class method.

When a script is compiled with `groovyc`, a new class file is generated for each class defined within the script, and an extra class file is generated for the script itself. This extra class in this file will typically be given the name of the script file (if one exists) or an auto-generated class name (if one does not). The resulting class will extend the `groovy.lang.Script` class, and the standalone statements encountered in the script will be compiled into the `run()` method of this class.

```
// MyScript.groovy
class MyClass {
    String name
}

def MyClass mycl = new MyClass(name:"Fred")
println mycl.name
```

If we compile this script with groovyc, we will get two files—`MyScript.class` and `MyClass.class`. So what would happen if we tried to parse and load this into a Java application by using `GroovyClassLoader`? The answer is much the same as when compiled with `groovyc`—two classes, instead of one, will be parsed and loaded. The class object returned will be `MyScript.class`, which will be a derived class of `groovy.lang.Script`. The `MyClass` class object is also parsed and loaded by the `GroovyClassLoader` so that the code that references it within the script can be resolved.

```
import java.io.File;
import groovy.lang.GroovyClassLoader;
import groovy.lang.Script;

public class LoadScript {

    public static void main(String[] args) {
        GroovyClassLoader classLoader = new GroovyClassLoader();

        Script script = null;
        try {
            script = (Script)classLoader.parseClass(
                new File("MyScript.groovy")).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }

        script.run();
    }
}
```

The standalone code referenced within the body of the script has been compiled into the `run()` method of the returned `Script` object. Therefore, we can execute it by calling the `run()` method on the returned instance object.

We saw in the last chapter how controlling the variables that are set in the binding of the script can be the foundation of many useful DSL features. Prior to calling `run()`, we have the opportunity to set variables into the binding of the script.

```
class MyClass {
String name
}

// Expect binding_name to be set to something within the binding of
// the script
def MyClass mycl = new MyClass(name:binding_name)
```

```
println mycl.name

import java.io.File;
import groovy.lang.GroovyClassLoader;
import groovy.lang.Script;
import groovy.lang.Binding;

public class LoadScript {

    public static void main(String[] args) {
        GroovyClassLoader classLoader = new GroovyClassLoader();

        Script script = null;
        try {
            script = (Script)classLoader.parseClass(
                new File("MyScript.groovy")).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }

        Binding binding = new Binding();
        binding.setVariable("binding_name", "Fred");
        script.setBinding(binding);
        script.run();
    }
}
```

It is possible to do almost anything with the binding from Java, including adding closures. A closure is just an instance of a `Closure` class. So to hand-craft a closure in Java, we extend `Closure` and implement the `doCall()` method.

Although it is possible to implement closures in Java, it is more than likely too much hard work. The best approach to embed your DSL into Java is therefore a combined approach, where the runtime and initialization code is either directly packaged as compiled Groovy classes, or is dynamically loaded by using `GroovyClassLoader`. The DSL scripts can then be loaded more conveniently, by using the Utility classes in Groovy that designed to do this, such as `GroovyShell` and `GroovyScriptEngine`.

GroovyShell

We have come across the `GroovyShell` class on several occasions, before.

`GroovyShell` allows us to load and evaluate Groovy scripts on the fly. In the last chapter, we used the `GroovyShell` in `RewardService` to load Reward DSL scripts and configure the binding, as follows:

```
void applyRewardsOnConsume(account, media) {
    if (on_consume_provided) {
        Binding binding = new Binding()
        binding.account = account
        prepareClosures(binding)
        prepareMedia(binding, media)

        GroovyShell shell = new GroovyShell(binding)
        shell.evaluate("onConsume.delegate = this;onConsume()")
    }
}
```

CompilerConfiguration

One feature of the `GroovyShell` that we did not cover is its use of the `CompilerConfiguration` class. This class gives us more fine-tuned control over the compilation process. `CompilerConfiguration` has numerous property getter and setter methods that can be used to control everything from debug and warning levels of the compiler to the classpath to use.

Perhaps the most interesting property that we can set on the `CompilerConfiguration` is `scriptBaseClass`. By default, all scripts have `java.lang.Script` set as their base class. But using this setting, we can change the base class to another derived class of `Script` of our own making. What is interesting about this is that it can be used as an alternative to the binding when setting up default variables or adding built-in methods to the script.

The `prepareClosures()` method called in the previous instance in `applyRewardsOnConsume()` sets up a closure called `points` in the binding.

```
binding.points = { points ->
    binding.account.points += points
}
```

The alternative to using the `CompilerConfiguration` would be to declare our own `RewardScript` class. We can set this into the `CompilerConfiguration` as the `scriptBaseProperty()` when constructing the `GroovyShell` object. We make this class abstract, as it is not going to implement `run()`. The method `run()` will be implemented later by `GroovyShell` when it evaluates our script.

```
import groovy.lang.Script

abstract class RewardScript extends Script {
    def points = { points ->
        binding.account.points += points
    }
}

// Setup CompilerConfiguration
def config = new CompilerConfiguration()
config.setScriptBaseClass("RewardScript")
GroovyShell shell = new GroovyShell(binding, config)
shell.evaluate("on_consume.delegate = this;on_consume()")
```

When loading a Script containing a DSL, we will often want to include other support classes that support and are shared by the code in the DSL. These classes will need to be on the `classpath` of the DSL script, but we don't necessarily want to have to place all of these supporting classes in the same location at the script itself. Nor do we always want to package these in the main application classpath. The `CompilerConfiguration` allows us to set up a classpath that is specific to the context of the script that we load with the `setClasspath` method.

We can control the level of debugging information printed when an exception is thrown in the script, via the `setDebug` method. This will turn on full stack traces, which will be printed to standard error output. By default, all output from the script will be printed to the same standard output as the calling application. We can change this to whatever we like by using the `setOutput` method, which allows us to set the `PrintWriter` to use for all standard output.

If we have a DSL script that can change while our application is running, we can use the `CompilerConfiguration` to manage auto recompiles so that we are always using the latest version of the script. The `setRecompileGroovySource` will cause Groovy sources to reload and recompile if they change on disk. The `setMinimumRecompilationInterval` allows us to configure the amount of time to wait before the sources are checked against the compiled versions in memory.

GroovyScriptEngine

`GroovyShell` is ideal for loading stand-alone scripts such as the Rewards DSL. More complex scenarios where scripts are interdependent can be tricky to manage with the `GroovyShell`. In this case, we have the option of using the `GroovyScriptEngine` class.

GroovyScriptEngine has a more extensive list of constructors to choose from than the Groovy shell. This allows us a greater degree of flexibility as to where we locate our classes. It also has the ability to monitor changes in the scripts as they occur and reload them automatically. This is extremely useful if we want to be able to make changes to a DSL script on the fly and have the changes executed immediately, without restarting the application.

```
def gse = new GroovyScriptEngine([".", ".../staging", ".../production"])
gse.run("MyScript.groovy", new Binding())
```

The first line above constructs the GroovyScriptEngine with an array of locations to search for scripts. The second line will attempt to load and run the script `MyScript.groovy`. GroovyScriptEngine will first search in the local directory, then in staging, and finally in production, and then load the first occurrence of `MyScript.groovy` that it comes across.

We can call `run()` multiple times for `MyScript.groovy`. If any changes occur in the script in between runs, the script will be reloaded by GroovyScriptEngine and the fresh version will be run on subsequent calls.

If you would like more control over how your scripts are loaded, you could consider writing your own resource connector. By implementing the `groovy.util.ResourceConnector` interface, we can support the loading of Groovy scripts from pretty much wherever we choose, such as from a remote file system or a database.

Summary

In this chapter, we have looked at the many different ways in which we can integrate Groovy code into Java. We've explored the issues around tightly integrating the two languages at compile time. We've seen how this can lead to dependency issues arising when Java code references Groovy classes and vice versa. We've taken a look at how we can use dependency injection frameworks like Spring to resolve some of these issues.

We've explored the various utility classes provided by Groovy that allow us to load and execute Groovy classes and scripts. In the rest of this book, we have gone through the numerous features of the Groovy language that enable the building of domain-specific languages. Now, with the features covered in this final chapter, you will have all the tools at your disposal to be able to fully integrate your own domain-specific scripting into your own Java applications, with confidence.

Index

Symbols

.class files 263
@attribute
 about 131
@Sputnik 194
-j jointCompilation option 265
3GLs 10
3rd Generation Languages. *See* 3GLs
4GLs 10
4th Generation Languages. *See* 4GLs

A

AbstractFactory class, FactoryBuilderSupport 194
 isLeaf method 221
 newInstance method 221
 onHandleNodeAttributes method 221
 onNodeCompleted method 221
 setChild method 221
 setParent method 221
Abstract Syntax Tree (AST) 196
Acceptance Test Driven Development. *See* ATDD
Account.credit method 237
Account class 255
account credit method 237, 255
account object 237
addTo(key) method 172
addTo{association} methods 173
addToInvoices method 172
addToOrders method 172
allof condition blocks 244
amount parameter 116
Ant

 about 180
 benefits 180
 build file 181
 Groovyc ant plugin 182
 invoking, from command line 182
 script, targets 182
ant.taskdef() call 185
AntBuilder
 about 129, 183
 and Gant 184, 186
 ant.taskdef() 185
 build.gant script 185
 code 184
 dependency-based targets 183
 depends() call 186
 depends() method 185
 setDefaultTarget(run) call 185
 target() 185
 target closure 185
AntBuilder class 157, 185
Ant script, targets
 compile 182
 init 182
 run 182
anyof condition blocks 244
applyRewardsOnConsume() 270
applyRewardsOnConsume, event hook
 method 252
applyRewardsOnPurchase, event hook
 method 252
applyRewardsOnUpgrade, event hook
 method 252
assertion methods, EasyB 192
assertions, Groovy language 41
associations relationship, GORM
 about 165

constraints 169, 170
many-to-many 174, 175
one-to-many 170-173
one-to-one 166, 167

ATDD
about 187
common language 188
autoboxing, Groovy language 42

B

BDD
about 157, 187
common language 188

BDD DSL style 196

BehaviorCategory class 192

Behavior Driven Development. *See BDD*

belongsTo, domain class 167

binding.outerBlock 238

Binding class 229

binding property 107

bindings, DSL
Account.credit method 237
and GORM DataSource DSL 239, 240
binding.outerBlock 238
block closures 232
closures, as built-in methods 231
closures, as repeatable blocks 231, 232
closures, as singleton blocks 234, 235
context forming, binding properties used
235, 237
enclosing property 232
GeeTwitter class 231
nestedBlock closures 232
results, storing 237, 238
specification parameter, using 233, 234

block, Spock
expect:, type 195
setup, type 195
types 195
when: then, type 195
where:, type 195

boilerplate
about 96
refactoring 97-99
removing 97
search, improving 102-104

boolean binding variables 250

BroadbandPlus
about 240, 241
application classes 255

BroadbandPlus application classes
about 255
BroadbandPlus class 256
consume API 256
extendMedia method 255

builder
about 117
AntBuilder 129
code structure 199, 200
design pattern 117, 118
DOMBuilder 129
Groovy Builders, using 119
JMXBuilder 129
MarkupBuilder 129
namespaced XML 121
NodeBuilder 129, 130
program logic, using 128
SAXBuilder 129
SwingBuilder 129

builder.root() 208

builder block 208

builder class 201

builder design pattern
about 117, 118
and GroovyMarkup 125-128
Builder 118
components 118, 119
ConcreteBuilder 118
Director 119
Product 119

BuilderSupport
about 209, 220
hook methods 209-213
database builder 214-216

BuilderSupport class 211

BuilderSupport hook methods
createNode(Object name) 209
createNode(Object name, Map attributes)
209
createNode(Object name, Map attributes,
Objects value) 209
createNode(Object name, Object value) 209

nodeCompleted(Object parent, Object node)
 209

nodeCompleted hook 212

Object getCurrent() hook 212

setCurrent(Object current) hook 212

setParent(Object parent, Object child) 209

setParent call 213

built-in methods

about 108, 110

adding 107

C

callable function, EasyB 192

category class 192

closure

about 62

and collection methods 63, 64

as method parameters 64

calling 68, 69

disadvantage 62

named closure field, finding 70

parameters 71

return values 78

scope 78-81

Closure.setDelegate method 147

Closure.setResolveStrategy 203

Closure1 parameter 208

closure delegate, Groovy

about 200, 208, 209

builder block 208

level1() call 208

level2() call 208

root() method call 208

closure method calls, Groovy

about 200

builder class 201

leaf() method 205

node() method 205

OWNER_FIRST 203, 205

resolve strategy 203

root() method 205

closure method resolution, Groovy

about 200

Closure object 202

closure parameter 233

closures

about 19, 93

using, as built-in methods 231

using, as repeatable blocks 231, 232

using, as singleton blocks 234, 235

closures, as method parameters

about 64

as DSL 65

parameters, forwarding 66, 67

closures, Groovy language 46-48

closures, parameters

and doCall() method 72-74

curried parameters 77

default parameter values 75

list 71

multiple parameters, passing 74, 75

syntax definitions 71

zero parameters, enforcing 75

closure scope

about 78-81

delegate variable 81

example 80

GStrings, using 79

owner variable 81

this variable 81

collection methods

and closures 63, 64

collections, Groovy language

lists 54, 55

maps 55-57

ranges 53, 54

command-line interface

adding 105, 106

CompilerConfiguration class 230, 282

composition relationship, GORM 175, 176

ConfigSlurper class 161

ConstrainedPropertyBuilder, builder class
 169

constraints, associations 169, 170

context() block 190, 193

control structures, Groovy language

about 49

Elvis operator 51

Groovy Falsehoods 50

Groovy truth 49

Java ternary operator 50

loops 52
switch statement 51, 52

createNode(Object name), BuilderSupport hook methods 209

createNode(Object name, Map attributes), BuilderSupport hook methods 209

createNode(Object name, Map attributes, Objects value), BuilderSupport hook methods 209

createNode(Object name, Object value), BuilderSupport hook methods 209

createNode hook 212

createNode method 217, 219

curried closures 77

currying 76

Customer, domain class 175

Customer.invokeMethod() 146

Customer.prettyPrint() method 146

Customer class 164, 165

customer object 222

D

database builder, BuilderSupport

- createNode method 217, 219
- customers method 216
- data sets, managing 214
- Independent Software Vendors (ISVs) 214
- migration plug-in 215
- nodeCompleted hook 217
- setParent hook 217
- setParent method 219

DataSource configuration, Grails

- database, creating 159
- DataSource.groovy source file, editing 160
- dbCreate field 161

declareNamespace() method 123

delegate keyword

- example 147, 148
- resolving rules 148

depends() call 186

depends() method 185

doCall() method 281

- and parameters 72-74

Document Type Definition. See DTD

domain class 165

Domain-Specific Languages. See DSL

DOMBuilder 129

downloading

- Twitter4J 88

DSL

- 3GL systems (3rd Generation Languages) 10
- 4GLs systems (4th Generation Languages) 10

about 7-9

allOf closure block, implementing 245

allof condition blocks 244

anyOf closure block, implementing 245

anyof condition blocks 244

binding 231

boilerplate, removing 96

built-in methods, adding 106

command-line interface, adding 105

ComplexNumber class 15

design 14

- external versus internal 14

EXTOL 12

for process engineers 12, 13

general-purpose languages 9

high-level programming language 10

implementing 14

language-oriented programming 11

named parameters 116

operator overloading 15, 16

programming languages, evolution 9

spreadsheet macros 10

stakeholder, participation 13, 14

structuring 243

Twitter APIs working with 86

users 11, 12

writing, requisites 242, 243

DSL, designing

- BroadbandPlus 240, 241
- BroadbandPlus application classes 255-257
- convenience methods 249, 250
- events, handling 247, 249
- GroovyTestCase, testing with 257, 259
- offers 250
- Reward, types 242
- Reward DSL 242-247
- RewardService class 251-255
- shorthand binding variables 249, 250

DSLs
 bindings 231
DTD 8
dynamic finders
 about 179
 findAllBy 179
 findBy 179

E

eachFollower method 93
eachFriend method 93
each method 204 140
EasyB
 about 191, 192
 assertion methods, list 192
 BehaviorCategory class 193
 it, specification keyword 191, 192
 URL, for downloading 191
 versus GSpec 192

Eclipse 34
EJB 8
Elvis operator 50
EmptyStack class 194
enclosing property 232
Enterprise Java Beans. *See EJB*

evaluate method
 about 105
ExpandoMetaClass
 about 151, 152
 constructors, adding 155
 dynamic method, naming 153
 existing method, replacing 152
 overload methods, adding 154
 static methods, adding 153
 static methods, overriding 153

EXTended Operations Language. *See EXTOL*
external DSL
 versus internal DSL 14, 15

EXTOL 12

F

FactoryBuilderSupport
 AbstractFactory class, methods 221
 builder, implementing 221

CustomerBuilder, replacement building 222
groovy.util.AbstractFactory class 221
groovy.util.FactoryBuilderSupport class 221
registerObjectFactorie, registration methods 225
working 221
FactoryBuilderSupport class 221
feature methods 194
FixedStack class 188
fixture methods 194
followers 85
follow method 93

G

\$groovyConsole command 32
Gant
 about 157, 180
 and AntBuilder 184, 186
 Ant 180
 ant.taskdef() call 185
 AntBuilder 183
 gant, command line 186
 setDefaultTarget(run) call 185
 target() call 185
GantBuilder metaclass 185
Gant project 135
GeeTwitter
 fleshing out 100-102
GeeTwitter.follow() 136
GeeTwitter.login method
 calling 102
GeeTwitter class 99, 231
GeeTwitter DSL 116
GeeTwitterScript.groovy class 110
get method 164
getAll method 178
getClass method 137
getDeclaredFields method 138
getFirstName method 263
getLastName method 263
getMetaClass method 144
getMethods method 138
getPackage method 138
getProperty 150

GGX 87
glaforge 86
GORM
about 8, 157, 158
as DSL 180
associations 165
composition 175, 176
Grails 158
grails-app directory 159
inheritance 176
model, building 161
querying 178
relationships, modeling 165
GORM classes 158
GORM DataSource DSL
and bindings 239, 240
GORM model
building 161, 162
domain classes, using 163, 164
Grails
grails-app directory 159
grails-app/conf 159
grails-app/controllers 159
grails-app/domain 159
grails-app/view 159
grails-app directory
DataSource, configuration 159, 161
grails-app/conf 159
grails-app/controllers 159
grails-app/domain 159
grails-app/view 159
grails create-domain-class command 169
Grails Object Relational Mapping. *See GORM*
groovu.util.ConfigSlurper, utility class 161
Groovy
about 17, 18
advantage 261
bindings 228, 229
builder 117, 199
builder, code structure 199, 200
calling, from Java 263
closure delegate 208
closure method calls 200-202
Closures 93
closures 19, 61
compiler 33
console 32
delegate keyword 147
downloading 26
ExpandoMetaClass 151, 152
friends, searching 93-95
groovyc command 33
groovyConsole command 32
groovysh command 30, 31
installing 26
integrating, with Java 261
interfaces 268-271
language features 18
list and maps, manipulation 18, 19
markup 21
metaprogramming 137
method pointer 136
named parameters 114
operator overloading 20
optional typing 18-21
owner keyword 147
pretended methods 206
regular expression support 20
running 26
script engine 27-29
search method, adding 95, 96
shebang scripts 29, 30
shell 30, 31
static typing 18
SwingBuilder 132
this keyword 146
VM interface 17
with JVM 17, 18
Groovy & Grails eXchange Conference. *See GGX*
Groovy, calling from Java
about 263
POGOs 263
POJOs 263, 265
Groovy, running
groovyc command 33
Groovy compiler 33
groovyConsole command 32, 33
Groovy script engine 27-29
groovysh command 30, 31
Groovy shell 30, 31
shebang scripts 29, 30
groovy.lang.Binding class 228

groovy.util.AbstractFactory class 221
groovy.util.ResourceConnector interface
 implementing 284
Groovy Builders, using
 GroovyMarkup 119, 120
 MarkupBuilder class 120, 121
Groovyc 182
groovyc command 33
GroovyClassLoader class
 about 276-278
 GroovyClassLoader class 283
 GroovyShell 282
 scripts, integrating 279-281
groovy command 105
Groovy compiler 33
groovyConsole 88
Groovy falsehoods 50
Groovy IDE
 Eclipse 34
 IntelliJ IDEA 34
 NetBeans 33
 other editors 34
 other IDEs 34
Groovy language
 assertions 41
 autoboxing 42
 closures 46, 48
 collections 53
 control structures 49
 Elvis operators 50, 51
 Groovy shorthand 36
 Groovy truth 49, 50
 loops 52
 methods 46
 module structure 34-36
 operators 57
 regular expressions 43-46
 strings 43
 switch statements 51, 52
Groovy language, features
 closures 19
 Groovy markup 21-23
 list and maps, manipulation 18, 19
 operator overloading 20
 optional syntax 21
 optional typing 18, 20
 regular expression support 20
 static typing 18
GroovyMarkup
 and Builder design pattern 125-128
Groovy script engine 27-29
GroovyScriptEngine class 284
Groovy Server Pages (GSPs) 159
GroovyService class 277
groovysh command 30, 31
GroovyShell
 about 282
 CompilerConfiguration class 282
Groovy shell 30, 31, 229
GroovyTestCase
 about 257
 running 259
 setUp method 257
Groovy truth 49
Groovy with Java, integrating
 about 261, 262
 compiling, to .class files 263
 dependencies, resolving 271, 272
 dependencies injection, Spring used 273-275
 Groovy, calling 263
 interfaces 268
 Java, calling 266
 privacy concerns 267, 268
GSpec
 about 188, 189
 FixedStack object 190
 GSpecBuilder class 191
 GSpecBuilderRunner class 190
 GSpecBuilderRunner object 190
 initially() block 190
 setProperty() method 191
 should_be property accessor 191
 should_equal method 190
 should_not_be property accessor 191
 should_not_equal method 190
 should_style assertion 190
 specify() block 190
 specify() method 190
 stack property 190
 versus EasyB 192
GSpecBuilder class 191

`GSpecDelegate.getProperty()` 191
`GSpecDelegate` class 191
`GSpecDelegate` object 191
`gTwitter` 86
`Gygwin` 105

H

`hasMany` map 173

I

`Identity` class 167
`Independent Software Vendors (ISVs)` 214
inheritance relationship, GORM
 about 176
 mapping 177, 178
`initially()` block 190
`IntelliJ IDEA` 34
internal DSL
 versus external DSL 14, 15
International Obfuscated C Code Contest.
 See IOCCC
`invokeMethod` 206, 207
`IOCCC` 9
`isLeaf` method 221

J

`Java`
 calling, from Groovy 266
 integrating, with Groovy 261
 interfaces 268-271
Java Virtual Machine. *See JVM*
`JMXBuilder` 129
`JRuby` 17
`JVM` 7
`Jython` 17

K

keywords
 `delegate` 147
 `owner` 147
 `static` 153
 `this` 146
 `use` 142

L

language-oriented programming 11
`lastName`, object attribute 217
`leaf()` method 205
`level1()` call 208
`level1()` method 208
`level2()` method 208
`list` method 164
lists 54, 55
`LogBuilder` 211
loops 52

M

many-to-many, associations 174, 175
mapping, inheritance 177, 178
maps 55-57
markup, Groovy 21-23
`MarkupBuilder`
 about 129
 working 149-151
`MarkupBuilder.customers()` 124
`MarkupBuilder` class
 about 120
 groovyConsole, running 120
`Markup` class 170
`Media` class 256
`message` variable 238
`Metaclass`
 `.metaClass` property, accessing 144
 about 144, 145
 `invokeMethod()` 145
 `MetaClass.invokeMethod()` 145
 storing 144
`MetaClass.invokeMethod()` 145
`MetaClass` instance 225
Meta Object Protocol. *See MOP*
metaprogramming
 about 137
 and MOP 137
`method1()` 202
`method3()` 203
`method` call 63
`methodMissing` 207, 208
`MethodMissingException` 208
`methodMissing` method 179

method pointer

assigning 136

methods, Groovy language 46

mini languages. *See* DSL

model classes 163

Model View Controller (MVC) architecture
159**module structure, Groovy language** 34-36**MOP**

about 119

category 142, 143

Expando 140, 141

Metaclass 144

reflection 137, 138

reflection, shortcuts 139

MVC 118**myMethod()** 121**N****named closure field**

finding 70

named parameters

about 114, 115

in DSL 116

method call, invoking 115

named parameters, Groovy 200**name parameter** 219**namespace.tag** 123**namespaced XML**

about 121

GroovyMarkup syntax 124

StreamingMarkupBuilder 122

xmlns namespaces 122

Napkin Look & Feel 132**NetBeans** 33**newInstance method** 221**node() method** 205**NodeBuilder**

about 129, 130

GPath, using 131

nodeCompleted(Object parent, Object node), BuilderSupport hook methods
209**nodeCompleted hook** 212, 217**null safe dereference operator** 58**O****Object getCurrent() hook** 212

one-to-many, associations 170-173

one-to-one, associations

Address class 168

Customer class 168

Identity, Customer class 168

onHandleNodeAttributes method 221**onNodeCompleted method** 221**operator overloading, DSL** 15, 16**operator overloading, Groovy** 20, 59**operators, Groovy language**

about 57

null safe deference 58

operator overloading 59

spread 57, 58

spread-dot 57, 58

optional typing, Groovy 18-21**owner keyword**

about 147

example 147, 148

resolving rules 148

P**Plain Old Groovy Objects.** *See* POGOs**Plain Old Java Object.** *See* POJO**POGO class** 158, 164**POGOs** 263**POJO** 263**PoorMansTagBuilder.methodMissing()** 208**PoorMansTagBuilder class** 206**prepareClosures() method** 282**pretended methods, Groovy**

about 200, 206

invokeMethod 206, 207

methodMissing 207

PoorMansTagBuilder class 206

prettyPrint() method 141**programming languages**

3GL systems (3rd Generation Languages)

10

4GLs 10, 11

4GL systems (4th Generation Languages)

10

evolution 9
general-purpose languages 9
high-level programming language 10
spreadsheets 10

Q

querying
about 178
dynamic finders 179
get() method 178
getAll() method 178
GORM, as DSL 180
list method 178

R

ranges 53, 54
regex find operator =~ 43
regex match operator ==~ 44
regex pattern operator ~String 43
regular expressions, Groovy language
about 43-46
regex find operator =~ 43
regex match operator ==~ 44
regex pattern operator ~String 43
regular expression support, Groovy 20
resolve strategy
about 150
DELEGATE_FIRST 150
DELEGATE_ONLY 150
OWNER_FIRST 150
OWNER_ONLY 150
TO_SELF 150
Reward
DSL 242, 243
types 242
RewardScript class 282
Rewards DSL
building 240
RewardService, event hook methods
applyRewardsOnConsume 252
applyRewardsOnPurchase 252
applyRewardsOnUpgrade 252
RewardService class 251 269
root() method 205

S

SalesOrder classes 171
save method 164
SAXBuilder 129
Script.evaluate() method 107
Script.getBinding() method 107
search method 105
setChild method 221
setClasspath method 283
setCurrent(Object current) hook 212
setDebug method 283
setFirstName() method 263
setOutput method 283
**setParent(Object parent, Object child),
BuilderSupport hook methods** 209
setParent call 213
setParent hook 217
setParent method 219-223
setProperty() method 191
setUp method 257
setVariable 229
shebang scripts 29
shell command 106
shorthand, Groovy language
assumed imports 36
parenthesis, optional 37, 38
properties and GroovyBeans 40, 41
return keyword, optional 39
semicolon, optional 37
visibility, default 37
should_be property accessor 191
should_equal method 190
should_not_be property accessor 191
should_not_equal method 190
Spaz 86
specify() method 190
specify block 190
Spock
@Speck annotation 194
@Sputnik 194
about 193
Abstract Syntax Tree (AST) 196
block 195
EmptyStack class 194
feature methods 194
fields 194

fixture methods 194
specification, parts 194
Spock Stack example, code snippet 193, 194
spread-dot operator 57, 58
spread operator 57, 58
spreadsheet macros 10
Spring
 using, for dependency injections 273-276
stack property 190
static typing, Groovy 18
String blanked method 154
strings, Groovy language 43
SwingBuilder
 about 129, 132
 Griffon 135
 pack() method 134
 show()method 134
 Twitter searching 134
switch statement 51

T

target() call 185
TDD 187
Test Driven Development. *See* TDD
the.stack.isEmpty() == false 190
The Apache Software Foundation 180
this keyword
 about 146
 example 147, 148
 resolving rules 148
Tweets 85
Twitter
 about 85, 86
 APIs, working with 86
 using 85, 86

Twitter.createFriendship method 93
Twitter.directMessages method 90
Twitter.getUserDetail method 89
Twitter.sendDirectMessage method 90
Twitter.updateStatus method 89
Twitter4J
 about 88
 direct messages 89
 downloading 88
 following concept 92
 searching 90, 91
 tweeting 88, 89
 using 88
Twitter API
 Twitter4J, using 88
 working with 86, 87

U

use block 143
use keyword 142

V

Virtual Machine interface. *See* VM interface
VM interface 17

W

wrappedObject.isEmpty() 191
X

xmlDeclaration() method 123
xUnit style testing
 issues 187



Thank you for buying **Groovy for Domain-Specific Languages**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

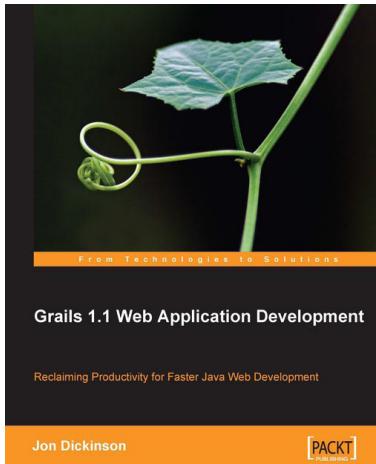
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



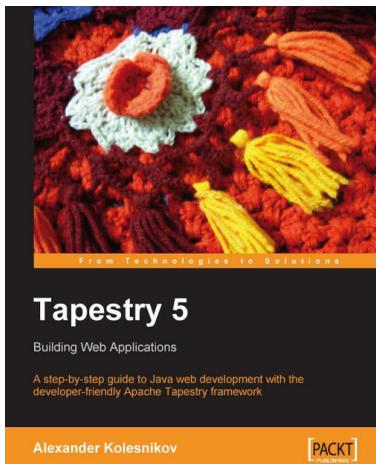
Grails 1.1 Web Application Development

ISBN: 978-1-847196-68-2

Paperback: 328 pages

Reclaiming Productivity for faster Java Web Development

1. Ideal for Java developers new to Groovy and Grails – this book will teach you all you need to create web applications with Grails
2. Create, develop, test, and deploy a web application in Grails
3. Take a step further into Web 2.0 using AJAX and the RichUI plug-in in Grails
4. Packed with examples and clear instructions to lead you through the development and deployment of a Grails web application



Tapestry 5: Building Web Applications

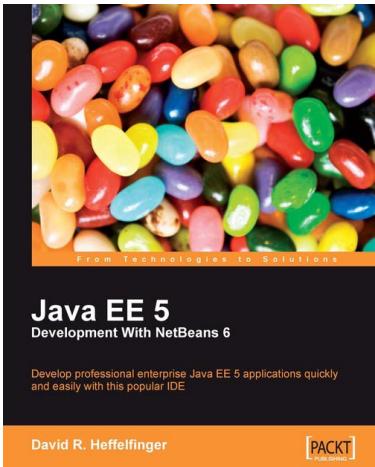
ISBN: 978-1-847193-07-0

Paperback: 280 pages

A step-by-step guide to Java Web development with the developer-friendly Apache Tapestry framework

1. Latest version of Tapestry web development framework
2. Get working with Tapestry components
3. Gain hands-on experience developing an example site
4. Practical step-by-step tutorial

Please check www.PacktPub.com for information on our titles



Java EE 5 Development with NetBeans 6

ISBN: 978-1-847195-46-3

Paperback: 400 pages

Develop professional enterprise Java EE applications quickly and easily with this popular IDE

1. Use features of the popular NetBeans IDE to improve Java EE development
2. Careful instructions and screenshots lead you through the options available
3. Covers the major Java EE APIs such as JSF, EJB 3 and JPA, and how to work with them in NetBeans
4. Covers the NetBeans Visual Web designer in detail



Flex 3 with Java

ISBN: 978-1-847195-34-0

Paperback: 304 pages

Develop rich internet applications quickly and easily using Adobe Flex 3, ActionScript 3.0 and integrate with a Java backend using BlazeDS 3.2

1. A step-by-step tutorial for developing web applications using Flex 3, ActionScript 3.0, BlazeDS 3.2, and Java
2. Build efficient and seamless data-rich interactive applications in Flex using a combination of MXML and ActionScript 3.0
3. Create custom UIs, Components, Events, and Item Renders to develop user friendly applications

Please check www.PacktPub.com for information on our titles