

A comprehensive step-by-step guide

Programming in Scala



artima

Martin Odersky
Lex Spoon
Bill Venners

Programming in Scala

PrePrintTM Edition

Thank you for purchasing the PrePrint™ Edition of *Programming in Scala*.

A PrePrint™ is a work-in-progress, a book that has not yet been fully written, reviewed, edited, or formatted. We are publishing this book as a PrePrint™ for two main reasons. First, even though this book is not quite finished, the information contained in its pages can already provide value to many readers. Second, we hope to get reports of errata and suggestions for improvement from those readers while we still have time incorporate them into the first printing.

As a PrePrint™ customer, you'll be able to download new PrePrint™ versions from Artima as the book evolves, as well as the final PDF of the book once finished. You'll have access to the book's content prior to its print publication, and can participate in its creation by submitting feedback. Please submit by clicking on the *Suggest* link at the bottom of each page.

Thanks for your participation. We hope you find the book useful and enjoyable.

Bill Venners
President, Artima, Inc.

Programming in Scala

PrePrintTM Edition

Martin Odersky, Lex Spoon, Bill Venners

artima

ARTIMA PRESS
MOUNTAIN VIEW, CALIFORNIA

Programming in Scala
PrePrint™ Edition Version 2

Martin Odersky is the creator of the Scala language and a professor at EPFL in Lausanne, Switzerland. Lex Spoon worked on Scala for two years as a post-doc with Martin Odersky. Bill Venners is president of Artima, Inc.

Artima Press is an imprint of Artima, Inc.
P.O. Box 390122, Mountain View, California 94039

Copyright © 2007, 2008 Martin Odersky, Lex Spoon, and Bill Venners.
All rights reserved.

PrePrint™ Edition first published 2007
Version 2 published February 18, 2008
Produced in the United States of America

12 11 10 09 08 2 3 4 5 6

No part of this publication may be reproduced, modified, distributed, stored in a retrieval system, republished, displayed, or performed, for commercial or noncommercial purposes or for compensation of any kind without prior written permission from Artima, Inc.

All information and materials in this book are provided "as is" and without warranty of any kind.

The term "Artima" and the Artima logo are trademarks or registered trademarks of Artima, Inc. All other company and/or product names may be trademarks or registered trademarks of their owners.

to Nastaran - M.O.

to Fay - L.S.

to Siew - B.V.

Overview

Contents	viii
Preface	xvi
Acknowledgments	xvii
Introduction	xx
1. A Scalable Language	27
2. First Steps in Scala	47
3. Next Steps in Scala	63
4. Classes and Objects	90
5. Basic Types and Operations	114
6. Functional Objects	136
7. Built-in Control Structures	151
8. Functions and Closures	169
9. Control Abstraction	190
10. Composition and Inheritance	205
11. Traits and Mixins	233
12. Case Classes and Pattern Matching	249
13. Packages and Imports	279
14. Working with Lists	292
15. Collections	322
16. Stateful Objects	342
17. Type Parameterization	362
18. Abstract Members and Properties	379
19. Implicit Conversions and Parameters	396
20. Implementing Lists	413
21. Object Equality	424
22. Working with XML	438
23. Actors and Concurrency	450
24. Extractors	461
25. Objects As Modules	473
26. Annotations	484
27. Combining Scala and Java	490
28. Combinator Parsing	502
Glossary	530
Bibliography	544
About the Authors	546

Contents

Contents	viii
Preface	xvi
Acknowledgments	xvii
Introduction	xx
1 A Scalable Language	27
1.1 A language that grows on you	28
1.2 What makes Scala scalable?	33
1.3 Why Scala?	36
1.4 Scala's roots	44
1.5 Conclusion	45
2 First Steps in Scala	47
Step 1. Learn to use the Scala interpreter	47
Step 2. Define some variables	49
Step 3. Define some functions	51
Step 4. Write some Scala scripts	55
Step 5. Loop with <code>while</code> , decide with <code>if</code>	57
Step 6. Iterate with <code>foreach</code> and <code>for</code>	59
Conclusion	62
3 Next Steps in Scala	63
Step 7. Understand the importance of <code>vals</code>	63
Step 8. Parameterize Arrays with types	65

Step 9. Use Lists and Tuples	69
Step 10. Use Sets and Maps	74
Step 11. Understand classes and singleton objects	79
Step 12. Understand traits and mixins	86
Conclusion	89
4 Classes and Objects	90
4.1 Objects and variables	91
4.2 Mapping to Java	93
4.3 Classes and types	96
4.4 Fields and methods	97
4.5 Class documentation	102
4.6 Variable scope	104
4.7 Semicolon inference	108
4.8 Singleton objects	110
4.9 A Scala application	111
4.10 Conclusion	113
5 Basic Types and Operations	114
5.1 Some basic types	115
5.2 Literals	116
5.3 Operators are methods	121
5.4 Arithmetic operations	125
5.5 Relational and logical operations	126
5.6 Object equality	128
5.7 Bitwise operations	130
5.8 Operator precedence and associativity	131
5.9 Rich wrappers	133
5.10 Conclusion	134
6 Functional Objects	136
6.1 A class for rational numbers	136
6.2 Choosing between <code>val</code> and <code>var</code>	138
6.3 Class parameters and constructors	139
6.4 Multiple constructors	140
6.5 Reimplementing the <code>toString</code> method	141
6.6 Private methods and fields	142

6.7	Self references	143
6.8	Defining operators	143
6.9	Identifiers in Scala	145
6.10	Method overloading	147
6.11	Going further	148
6.12	A word of caution	149
6.13	Conclusion	150
7	Built-in Control Structures	151
7.1	If expressions	152
7.2	While loops	154
7.3	For expressions	156
7.4	Try expressions	161
7.5	Match expressions	164
7.6	Living without break and continue	166
7.7	Conclusion	168
8	Functions and Closures	169
8.1	Methods	169
8.2	Nested functions	171
8.3	First-class functions	172
8.4	Short forms of function literals	175
8.5	Placeholder syntax	175
8.6	Partially applied functions	177
8.7	Closures	181
8.8	Repeated parameters	184
8.9	Tail recursion	185
8.10	Conclusion	189
9	Control Abstraction	190
9.1	Reducing code duplication	190
9.2	Simplifying client code	194
9.3	Currying	196
9.4	Writing new control structures	198
9.5	By-name parameters	201
9.6	Conclusion	204

10 Composition and Inheritance	205
10.1 Introduction	205
10.2 Abstract classes	206
10.3 The Uniform Access Principle	207
10.4 Assertions and assumptions	209
10.5 Subclasses	210
10.6 Two name spaces, not four	211
10.7 Class parameter fields	213
10.8 More method implementations	214
10.9 Private helper methods	215
10.10 Imperative or functional?	216
10.11 Adding other subclasses	218
10.12 Override modifiers and the fragile base class problem	219
10.13 Factories	221
10.14 Putting it all together	222
10.15 Scala's class hierarchy	225
10.16 Implementing primitives	229
10.17 Bottom types	231
10.18 Conclusion	232
11 Traits and Mixins	233
11.1 Syntax	233
11.2 Thin versus thick interfaces	235
11.3 The standard Ordered trait	236
11.4 Traits for modifying interfaces	238
11.5 Stacking modifications	241
11.6 Locking and logging queues	242
11.7 Traits versus multiple inheritance	245
11.8 To trait, or not to trait?	247
12 Case Classes and Pattern Matching	249
12.1 A simple example	249
12.2 Kinds of patterns	253
12.3 Pattern guards	261
12.4 Pattern overlaps	263
12.5 Sealed classes	264
12.6 The Option type	266

12.7 Patterns everywhere	267
12.8 A larger example	271
12.9 Conclusion	278
13 Packages and Imports	279
13.1 Packages	280
13.2 Imports	283
13.3 Access modifiers	286
14 Working with Lists	292
14.1 List literals	292
14.2 The List type	292
14.3 Constructing lists	293
14.4 Basic operations on lists	294
14.5 List patterns	295
14.6 Operations on lists Part I: First-order methods	297
14.7 Operations on lists Part II: Higher-order methods	307
14.8 Operations on lists Part III: Methods of the List object	315
14.9 Understanding Scala's type inference algorithm	318
15 Collections	322
15.1 Overview of the library	322
15.2 Sequences	324
15.3 Tuples	328
15.4 Sets and maps	331
15.5 Initializing collections	335
15.6 Immutable collections	336
15.7 Conclusion	341
16 Stateful Objects	342
16.1 What makes an object stateful?	342
16.2 Reassignable variables and properties	344
16.3 Case study: discrete event simulation	347
16.4 A language for digital circuits	348
16.5 The Simulation API	352
16.6 Circuit Simulation	355
16.7 Conclusion	361

17 Type Parameterization	362
17.1 Functional queues	362
17.2 Information hiding	365
17.3 Variance annotations	367
17.4 Lower bounds	374
17.5 Contravariance	375
17.6 Object-local data	376
17.7 Conclusion	378
18 Abstract Members and Properties	379
18.1 Abstract vals	380
18.2 Abstract vars	381
18.3 Abstract types	382
18.4 Case study: Currencies	385
18.5 Conclusion	395
19 Implicit Conversions and Parameters	396
19.1 Implicit conversions	396
19.2 The fine print	399
19.3 Implicit conversion to an expected type	403
19.4 Converting the receiver	403
19.5 Implicit parameters	406
19.6 View bounds	408
19.7 Debugging implicits	410
20 Implementing Lists	413
20.1 The List class in principle	413
20.2 The ListBuffer class	418
20.3 The List class in practice	420
20.4 Conclusion	422
21 Object Equality	424
21.1 Writing an equality method	425
22 Working with XML	438
22.1 Semi-structured data	438
22.2 Creating XML	439

22.3	Taking XML apart	442
22.4	Loading and saving	444
22.5	Pattern matching	446
22.6	Conclusion	449
23	Actors and Concurrency	450
23.1	Overview	450
23.2	Locks considered harmful	450
23.3	Actors and message passing	451
23.4	Treating native threads as actors	454
23.5	Tips for better actors	455
24	Extractors	461
24.1	An Example	461
24.2	Extractors	463
24.3	Patterns with zero or one variables	465
24.4	Variable argument extractors	467
24.5	Extractors and sequence patterns	469
24.6	Extractors vs Case Classes	470
24.7	Conclusion	472
25	Objects As Modules	473
25.1	A basic database	473
25.2	Abstraction	476
25.3	Splitting modules into traits	478
25.4	Runtime linking	480
25.5	Tracking module instances	481
25.6	Conclusion	483
26	Annotations	484
26.1	Why have annotations?	484
26.2	Syntax of annotations	485
26.3	Standard annotations	487
26.4	Conclusion	489
27	Combining Scala and Java	490
27.1	Translation details	490

27.2 Annotations	494
27.3 Existential types	498
27.4 Conclusion	501
28 Combinator Parsing	502
28.1 Example: Arithmetic Expressions	503
28.2 Running Your Parser	505
28.3 Another Example: JSON	507
28.4 Parser Output	509
28.5 Implementing Combinator Parsers	513
28.6 Lexing and Parsing	522
28.7 Standard Token Parsers	523
28.8 Error reporting	524
28.9 Backtracking vs LL(1)	526
28.10 Conclusion	528
Glossary	530
Bibliography	544
About the Authors	546

Preface

I'd like to thank you for purchasing the PrePrint™ Edition of *Programming in Scala*. Even though the book is still somewhat rough, we believe you can already use this book very effectively to learn Scala.

We released this PrePrint™ quite early in the book-writing process in part because so little documentation has up to now existed for Scala, but also because we want feedback to help us make the book better. At the bottom of each page is a *Suggest* link, which will take you to a small web application where you can submit comments about that page. We'll know which version and page you're on, so all you need to do is type your comment.

At this point we're interested in feedback on all aspects of the book except formatting. Please report any misspelled words, typos, or grammar errors. Let us know if you find something confusing, and be specific. Point out places where we appear to assume you know something that you don't. In short, we're interested in hearing about your reading experience, but please don't report formatting errors.

Formatting is something we plan to fix shortly before we send the book to the printer. At this point, we are aware that some lines are too long, the words are spaced out a bit too much here and there, the figures are surrounded by inconsistently sized white space, etc. Please forgive us for this for the time being, and don't report such problems.

By purchasing Version 2 of *Programming in Scala*, PrePrint™ Edition, you are entitled to download all updates until we publish the final version of the book. We thank you again for getting in on the ground floor and look forward to your feedback.

Bill Venners
Sunnyvale, California
February 18, 2008

Acknowledgments

Many people have contributed to this book and to the material it covers. We are grateful to all of them.

Scala itself has been a collective effort of many people. The design and the implementation of version 1.0 was helped by Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. Iulian Dragos, Gilles Dubochet, Philipp Haller, Sean McDermid, Ingo Maier, and Adriaan Moors joined in the effort to develop the second and current version of the language and tools.

Gilad Bracha, Craig Chambers, Erik Ernst, Matthias Felleisen, Shriram Krishnamurti, Gary Leavens, Sebastian Maneth, Erik Meijer, David Pollak, Jon Pretty, Klaus Ostermann, Didier Rémy, Vijay Saraswat, Don Syme, Mads Torgersen, Philip Wadler, Jamie Webb, and John Williams have shaped the design of the language by graciously sharing their ideas with us in lively and inspiring discussions, as well as through comments on previous versions of this document. The contributors to the Scala mailing list have also given very useful feedback that helped us improve the language and its tools.

George Berger has worked tremendously to make the build process and the web presence for the book work smoothly. As a result this project has been delightfully free of technical snafus.

Many people have given us feedback on early versions of the text. Thank you to Eric Armstrong, George Berger, Gilad Bracha, William Cook, Bruce Eckel, Stéphane Micheloud, Todd Millstein, David Pollak, Frank Sommers, Philip Wadler, and Matthias Zenger. We'd also like to thank Dewayne Johnson and Kim Leedy for their help with the cover art.

Bill would also like to thank Gary Cornell, Greg Doench, Andy Hunt, Mike Leonard, Tyler Ortman, and Dave Thomas for providing insight and

advice on book publishing.

And we'd like to extend a very special thanks to all of our readers who contributed comments. Your comments have been very helpful to us in shaping this into an even better book. Here's the names of everyone who submitted comments on eBook version 1 by clicking on the *Suggest* link, sorted first by the total number of comments submitted (higher numbers first), then alphabetically. Thanks goes to: Blair Zajac, Tony Sloane, Javier Diaz Soto, Mats Henricson, Justin Forder, David Biesack, Nigel Harrison, Donn Stephan, Mark Hayes, Calum MacLean, Les Pruszynski, Martin Elwin, Dmitry Grigoriev, Ervin Varga, Marius Scurtescu, Jeff Ervin, Jamie Webb, Howard Lovatt, Shanks Surana, Alexandre Patry, Eric Willigers, Filip Moens, Peter McLain, Arkadiusz Stryjski, Boris Lorbeer, Fred Janon, Jim Menard, George Berger, Martin Smith, Richard Dallaway, Thomas Jung, Andrew Tolopko, Joshua Cough, Craig Bordelon, Juan Miguel Garcia Lopez, Matt Russell, Michel Schinz, Peter Moore, Randolph Kahle, Bjarte S. Karlsen, Colin Perkins, Hiroaki Nakamura, Martin Smith, Bhaskar Maddala, George Kollias, Kristian Nordal, Maarten Hazewinkel, Marcus Schulte, Normen Mueller, Ole Hougaard, Rafael Ferreira, Vadim Gerassimov, Cameron Taggart, Lars Westergren, Sam Owen, Silvestre Zabala, Will McQueen, Christopher Rodrigues Macias, Jeroen Dijkmeijer, Jorge Ortiz, Michel Salim, Scot McSweeney-Roberts, Tim Azzopardi, Benjamin Smith, Eelco Hillenius, F Isphording, Frederic Jean, James Iry, Jonathan Feinberg, Marcus Dubois, Paolo Losi, Robert Christie, Trustin Lee, Alx Barker, Bjorn Zetterstrom, Chee Seng Chua, Chris Hagan, Claudi Paniagua, David Bernard, Deklan Dieterly, Hans Dockter, JC Zulian, Jay Lawrence, Johannes Rudolph, John D. Heintz, John Franey, Martin Scheffler, Matthew Passell, Michael Campbell, Robert Alexander, Rupert Key, Simon Epstein, Thomas Boam, Tomoyasu Kobayashi, Tyler Perkins, Aashutosh Vijayant, Amir Michail, Daniel Wellman, Dianne Marsh, Dirk Meister, Edgar Honing, Guy Oliver, Hanson Char, Jan Lohre, Jeff Heon, Johan Karlberg, John Tyler, Jon-Anders Teigen, Jonathan O'Connor, Jozef Saniga, Jörn Zaefferer, Kean Heng Lim, Kegan Gan, Marc Boehret, Maurice Walton, Narayan Iyer, Oliver Goodman, Paul Jackson, Roberto Chiaretti, Ryan Boyer, Sergey Novgorodsky, Sultan Rehman, Tom Davies, Tom Duffey, Tristan Woerth, Ulrik Rasmussen, Will Hays, William Heelan, and Wim Stolker.

Now, we can't promise this long list will make it into the printed book, but we'll see what we can do. We wanted to let you know at least in this

PrePrint that we find your feedback very helpful and appreciate it. Sorry if we missed anyone, as some people have submitted feedback via forums or email. It is a bit more convenient for us to process feedback submitted via the *Suggest* link, so we encourage that, but we appreciate your suggestions no matter how they come in.

Introduction

This book is a tutorial for the Scala programming language, written by people directly involved in the development of Scala. Our goal is that by reading this book, you can learn everything you need to be a productive Scala programmer.

Who should read this book

The main target audience is programmers who want to learn to program in Scala. If you want to do your next software project in Scala, then this is the book for you. In addition, the book should be interesting to programmers wishing to expand their horizons by learning new concepts. If you're a Java programmer, for example, reading this book will expose you to many concepts from functional programming as well as advanced object oriented ideas. We believe learning Scala can help you become a better programmer in general.

General programming knowledge is assumed. While Scala is a fine first programming language, this is not the book to use to learn programming.

On the other hand, no specific knowledge of programming languages is required. Even though most people use Scala on the Java platform, this book does not presume you know anything about Java. However, we expect many readers to be familiar with Java, and so we sometimes compare Scala to Java to help such readers understand the differences.

How to use this book

The main purpose of this book is to serve as a tutorial to help you learn to program in Scala. Thus, the recommended way to read this book is in chapter

order, from front to back. We have tried hard to introduce one topic at a time, and only explain new topics in terms of topics we've already introduced. Thus, if you skip to the back to get an early peak at something, you may find it explained in terms of things you don't quite understand. To the extent you read the chapters in order, we think you'll find it quite straightforward to gain competency in Scala, one step at a time.

If you see a term you do not know, be sure to check the glossary and the index. Many readers will skim parts of the book, and that is just fine. The glossary and index can help you backtrack whenever you skim over something too quickly.

After you have read the book once, it should also serve as a language reference. There is a formal specification of the Scala language, but the language specification tries for precision at the expense of readability. Although this book doesn't cover every detail of Scala, it is quite comprehensive and should serve as an approachable language reference as you become more adept at programming in Scala.

How to learn Scala

You will learn a lot about Scala simply by reading this book from cover to cover. You can learn Scala faster and more thoroughly, though, if you do a few extra things.

First of all, you can take advantage of the many program examples included in the book. Typing them in yourself is a way to force your mind through each line of code. Trying variations is a way to make them more fun and to make sure you really understand how they work.

Second, keep in touch with the numerous online forums. That way, you and other Scala enthusiasts can help each other. There are numerous mailing lists, there is a wiki, and there are multiple Scala-specific article feeds. Take some time to find ones that fit your information needs. You will spend a lot less time stuck on little problems, so you can spend your time on deeper, more important questions.

Finally, once you have read enough, take on a programming project of your own. Work on a small program from scratch, or develop an add-in to a larger program. You can only go so far by reading.

Ebook features

The eBook is not simply a printable version of the paper version of the book. While the content is the same as in the paper version, the eBook has been carefully prepared for reading on a computer screen.

The first thing to notice is that most references within the book are hyperlinked. If you select a reference to a chapter, figure, or glossary entry, your browser should take you immediately to the selected item so that you do not have to flip around to find it.

Additionally, at the bottom of each page are a number of navigation links. The “Cover,” “Overview,” and “Contents” links take you to major portions of the book. The “Glossary” and “Index” links take you to reference parts of the book. Finally, the “Discuss” link takes you to an online forum where you discuss questions with other readers, the authors, and the larger Scala community. If you find a typo, or something you think could be explained better, please click on the “Suggest” link, which will take you to an online web application with which you can give the authors feedback.

Although the same pages appear in the eBook as the printed book, blank pages are removed and the remaining pages renumbered. The pages are numbered differently so that it is easier for you to determine PDF page numbers when printing only a portion of the eBook. The pages in the eBook are, therefore, numbered exactly as in your PDF reader will number them.

Typographic conventions

The first time a *term* is used, it is italicized. Small code examples, such as `x + 1`, are written inline with a mono-spaced font. Larger code examples are put into mono-spaced quotation blocks like this:

```
def hello() {  
    println("Hello, world!")  
}
```

When interactive shells are shown, responses from the shell are shown in a lighter font.

```
scala> 3 + 4  
res0: Int = 7
```

Content overview

- [Chapter 1](#), “A Scalable Language,” describes the history of Scala and why you should care about the language.
- [Chapter 2](#), “First Steps in Scala,” rapidly shows you how to do a number of basic programming tasks in Scala, without going into detail about how they work.
- [Chapter 3](#), “Next Steps in Scala,” continues the previous chapter and rapidly shows you several more basic programming tasks.
- [Chapter 4](#), “Classes and Objects,” starts the in-depth coverage of Scala with a description of the basic building blocks of object-oriented languages.
- [Chapter 5](#), “Basic Types and Operations,” shows how to work with common types like integers and common operations like addition.
- [Chapter 6](#), “Functional Objects,” goes into more depth on object-oriented structures, using immutable rational numbers as a case study.
- [Chapter 7](#), “Basic Control Structures,” shows how to use familiar control structures like `if` and `while`.
- [Chapter 8](#), “Functions,” discusses in depth functions, the basic building block of functional languages.
- [Chapter 10](#), “Composition and Inheritance,” discusses more of Scala’s support object-oriented programming. The topics are not as fundamental as those in [Chapter 4](#), but they frequently arise in practice.
- [Chapter 11](#), “Traits and Mixins,” shows Scala’s *trait* mechanism for mixin composition.
- [Chapter 12](#), “Case Classes and Pattern Matching,” introduces *case classes* and *pattern matching*, twin constructs that support you when writing regular, non-encapsulated data structures. The two constructs are particularly helpful for tree-like recursive data.

- [Chapter 13](#), “Packages and Imports,” is the first chapter to discuss issues with programming in the large. It discusses top-level packages, import statements, and access control modifiers like `public` and `private`.
- [Chapter 14](#), “Working with Lists,” explains lists in detail. Lists are probably the most commonly used data structure in Scala programs.
- [Chapter 15](#), “Collections,” shows you how to use Scala’s collection library to organize large amounts of data.
- [Chapter 16](#), “Stateful Objects,” explains what stateful objects are, and what Scala provides in term of syntax to express them. The second part of this chapter also introduces a larger case study on discrete event simulation, an application area where stateful objects arise naturally.
- [Chapter 17](#), “Type Parameterization,” explains some of the techniques for information hiding introduced in the [Chapter 13](#) by means of a concrete example: the design of a class for purely functional queues. The chapter builds up to a description of variance of type parameters and how it interacts with information hiding.
- [Chapter 18](#), “Abstract Members and Properties,” describes all kinds of abstract members that Scala supports. Not only methods, but also fields and types can be declared abstract.
- [Chapter 19](#), “Implicit Conversions and Parameters,” describes implicit conversions and implicit parameters, two constructs which help programmers omit tedious details from source code and let the compiler infer them.
- [Chapter 20](#), “Implementing Lists,” describes the implementation of class `List`. It is important to understand how lists work in Scala, and furthermore the implementation demonstrates the use of several Scala features.
- [Chapter 21](#), “Object Equality,” points out several issues to consider when writing an `equals` method. There are several pitfalls to avoid.

- [Chapter 22](#), “Working with XML,” shows you how to process XML in Scala. It shows you idioms for generating XML, parsing it, and processing it once it is parsed.
- [Chapter 23](#), “Actors and Concurrency,” shows you how to use Scala’s *actors* support for concurrency. You can also use the threads and locks of the native platform, which work in Scala, but actors help you avoid the deadlocks and race conditions that plague that concurrency approach.
- [Chapter 24](#), “Extractors,” shows how to pattern match against arbitrary classes, not just case classes.
- [Chapter 25](#), “Objects As Modules,” shows how Scala’s objects are rich enough to remove the need for a separate modules system.
- [Chapter 26](#), “Annotations,” shows how to work with language extension via annotation. The chapter shows several standard annotations and shows you how to make your own.
- [Chapter 27](#), “Combining Scala and Java,” discusses several issues that arise when programming in Scala on the Java platform.

Programming in Scala

PrePrintTM Edition

Chapter 1

A Scalable Language

The name Scala stands for “scalable language.” The language is so named because it was designed to grow with the demands of its users. You can apply Scala to a wide range of programming tasks, from writing small scripts to building large systems.¹

Scala is easy to get into. It runs on the standard Java platform and it interoperates seamlessly with all Java libraries. It’s a great language for writing scripts that pull together Java components. But it can play out its strengths even more for building large systems and frameworks of reusable components.

Technically, Scala is a blend of object-oriented and functional programming concepts in a statically typed language. The fusion of object-oriented and functional programming shows up in many different aspects of Scala; it is probably more pervasive than in any other widely used language. The two programming styles have complementary strengths when it comes to scalability. Scala’s functional programming constructs make it easy to build interesting things quickly from simple parts. Its object-oriented constructs make it easy to structure larger systems and to adapt them to new demands. The combination of both styles in Scala makes it possible to express new kinds of programming patterns and component abstractions. It also leads to a legible and concise programming style. Because it is so malleable, programming in Scala can be a lot of fun.

This initial chapter answers the question, “Why Scala?” It gives a high-level view of Scala’s design and the reasoning behind it. After reading the

¹ Scala is pronounced *skah-la*.

chapter you should have a basic feel for what Scala is and what kinds of tasks it might help you accomplish. Although this book is a Scala tutorial, this chapter isn't really part of the tutorial. If you're anxious to start writing some Scala code, you should jump ahead to [Chapter 2](#).

1.1 A language that grows on you

Programs of different sizes tend to require different programming constructs. Consider, for example, the following small Scala program:

```
var capital = Map("US" -> "Washington",
                  "France" -> "Paris")

capital += ("Japan" -> "Tokyo")

println(capital("France"))
```

This program sets up a map from countries to their capitals, modifies the map by adding a new binding ("Japan" → "Tokyo"), and prints the capital associated with the country France.² The notation in this example is high-level, to the point, and uncluttered with extraneous semicolons or type annotations. Indeed, the feel is that of a modern “scripting” language like Perl, Python or Ruby. One common characteristic of these languages, which is relevant for the example above, is that they each support an “associative map” construct in the syntax of the language.

Associative maps are very useful because they help keep programs legible and concise. However, sometimes you might not agree with their “one size fits all” philosophy, because you need to control the properties of the maps you use in your program in a more fine-grained way. Scala gives you this fine-grained control if you need it, because maps in Scala are not language syntax. They are library abstractions that you can extend and adapt.

In the above program, you'll get a default `Map` implementation, but you can easily change that. You could for example specify a particular implementation, such as a `HashMap` or a `TreeMap`, or you could specify that the map should be thread-safe, “mixing in” a `SynchronizedMap` “trait.” You could specify a default value for the map, or you could override any other

²Please bear with us if you don't understand all details of this program. They will be explained in the next two chapters.

method of the map you create. In each case, you can use the same easy access syntax for maps as in the example above.

This example shows that Scala can give you both convenience and flexibility. Scala has a set of convenient constructs that help you get started quickly and let you program in a pleasantly concise style. At the same time, you have the assurance that you will not “outgrow” the language. You can always tailor the program to your requirements, because everything is based on library modules that you can select and adapt as needed.

Growing new types

Eric Raymond introduced the cathedral and bazaar as two metaphors of software development.³ The cathedral is a near-perfect building that takes a long time to build. Once built, it stays unchanged for a long time. The bazaar, by contrast, is adapted and extended each day by the people working in it. In Raymond’s work the bazaar is a metaphor for open-source software development. Guy Steele noted in a talk on “growing a language” that the same distinction can be applied to language design.⁴ Scala is much more like a bazaar than a cathedral, in the sense that it is designed to be extended and adapted by the people programming in it. Instead of providing all constructs you might ever need in one “perfectly complete” language, Scala puts the tools for building such constructs into your hands.

Here’s an example. Many applications need a type of integer that can become arbitrarily large without overflow or “wrap-around” of arithmetic operations. Scala defines such a type in a library class `scala.BigInt`. Here is the definition of a method using that type, which calculates the factorial of a passed integer value:⁵

```
def factorial(x: BigInt): BigInt =  
    if (x == 0) 1 else x * factorial(x - 1)
```

Now, if you call `factorial(30)` you would get:

```
265252859812191058636308480000000
```

³Raymond, *The Cathedral and the Bazaar* [Ray99]

⁴Steele, “Growing a language” [Ste99]

⁵`factorial(x)`, or $x!$ in mathematical notation, is the result of computing $1 * 2 * \dots * x$, with $0!$ defined to be 1.

BigInt looks like a built-in type, because you can use integer literals and operators such as `*` and `-` with values of that type. Yet it is just a class that happens to be defined in Scala’s standard library. If the class were missing, it would be straightforward for any Scala programmer to write an implementation, for instance by wrapping Java’s class `java.math.BigInteger` (in fact that’s how Scala’s BigInt class is implemented).

Of course, you could also use Java’s class directly. But the result is not nearly as pleasant, because although Java allows you to create new types, those types don’t feel much like native language support:

```
import java.math.BigInteger

def factorial(x: BigInteger): BigInteger =
  if (x == BigInteger.ZERO)
    BigInteger.ONE
  else
    x.multiply(factorial(x.subtract(BigInteger.ONE)))
```

BigInt is a representative of many other number-like types—big decimals, complex numbers, rational numbers, confidence intervals, polynomials—the list goes on. Some programming languages implement some of these types natively. For instance, Lisp, Haskell, and Python implement big integers; Fortran and Python implement complex numbers. But no sane language can implement all these abstractions at the same time. It would simply become too big to be manageable. What’s more, even if such a language were to exist, some applications would surely benefit from other number-like types that were not supplied. So the approach of attempting to provide everything in one language doesn’t scale very well. Instead, Scala allows users to grow and adapt the language in the directions they need by defining easy-to-use libraries that *feel* like native language support.

Growing new control constructs

The previous example demonstrates that Scala lets you add new types that can be used as conveniently as built-in types. The same extension principle also applies to control structures. This kind of extensibility is illustrated by Scala’s API for “actor-based” concurrent programming.

As multicore processors proliferate in the coming years, achieving acceptable performance will demand you use more parallelism in your ap-

plications. Often, this means rewriting your code so that computations are distributed over several concurrent threads. Unfortunately, creating dependable multi-threaded applications has proven challenging in practice. Java’s threading model is built around shared memory and locking, a model that is often difficult to reason about, especially as systems scale up in size and complexity. It is hard to be sure you don’t have a race condition or deadlock lurking—something that didn’t show up during testing, but might just show up in production. An arguably safer alternative is a message passing architecture such as the “actors” approach used by the Erlang programming language.

Java comes with a rich thread-based concurrency library. Scala programs can use it like any other Java API. However, Scala also offers an additional library that essentially implements Erlang’s actor model.

Actors are concurrency abstractions that can be implemented on top of threads. They communicate by sending messages to each other. An actor can perform two basic operations, message send and receive. The send operation, denoted by an exclamation point (!), sends a message to an actor. Here’s an example in which the actor is named `recipient`:

```
recipient ! msg
```

A send is asynchronous; that is, the sending actor can proceed immediately, without waiting for the message to be received and processed. Every actor has a *mailbox* in which incoming messages are queued. An actor handles messages that have arrived in its mailbox via a receive block:

```
receive {  
    case Msg1 => ... // handle Msg1  
    case Msg2 => ... // handle Msg2  
    // ...  
}
```

A receive block consists of a number of cases that each query the mailbox with a message pattern. The first message in the mailbox that matches any of the cases is selected, and the corresponding action is performed on it. If the mailbox does not contain any messages that match one of the given cases, the actor suspends and waits for further incoming messages.

As an example, here is a simple Scala actor implementing a checksum calculator service:

```
actor {
    var sum = 0
    loop {
        receive {
            case Data(bytes)      => sum += hash(bytes)
            case GetSum(requester) => requester ! sum
        }
    }
}
```

This actor first defines a local variable named `sum` with initial value zero. It then repeatedly waits in a loop for messages, using a `receive` statement. If it receives a `Data` message, it adds a hash of the sent `bytes` to the `sum` variable. If it receives a `GetSum` message, it sends the current value of `sum` back to the `requester` using the message send `requester ! sum`. The `requester` field is embedded in the `GetSum` message; it refers usually to the actor that made the request.

We don't expect you to understand fully the actor example at this point. Rather, what's significant about this example for the topic of scalability is that neither `actor`, `loop` nor `receive` nor message send (`!`) are built-in operations in Scala. Even though `actor`, `loop` and `receive` look and act very similar to control constructs like `while` or `for` loops, they are in fact methods defined in Scala's actors library. Likewise, even though (`!`) looks like a built-in operator, it too is just a method defined in the actors library. All four of these constructs are completely independent of the Scala language.

The `receive` block and send (`!`) syntax look in Scala much like they look in Erlang, but in Erlang, these constructs are built into the language. Scala also implements most of Erlang's other concurrent programming constructs, such as monitoring failed actors and time-outs. All in all, actors have turned out to be a very pleasant means for expressing concurrent and distributed computations. They feel like an integral part of Scala.

This example illustrates that you can “grow” the Scala language in new directions even as specialized as concurrent programming. To be sure, you need good architects and programmers to do this. But the crucial thing is that it is feasible—you can design and implement abstractions in Scala that address radically new application domains, yet still feel like native language support.

1.2 What makes Scala scalable?

Scalability is influenced by many factors, ranging from syntax details to component abstraction constructs. If we were forced to name just one aspect of Scala that helps scalability, we'd pick its combination of object-oriented and functional programming (well, we cheated, that's really two aspects, but they are intertwined).

Scala goes further than all other well-known languages in fusing object-oriented and functional programming into a uniform language design. For instance, where other languages might have objects and functions as two different concepts, in Scala a function value *is* an object. Function types are classes that can be inherited by subclasses. This might seem nothing more than an academic nicety, but it has deep consequences for scalability. In fact the actor concept shown previously could not have been implemented without this unification of functions and objects.

Scala is object-oriented

Object-oriented programming has been immensely successful. Starting from Simula in the mid-60's and Smalltalk in the 70's, it is now available in more languages than not. In some domains objects have taken over completely. While there is not a precise definition of what object-oriented means, there is clearly something about objects that appeals to programmers.

In principle, the motivation for object-oriented programming is very simple: all but the most trivial programs need some sort of structure. The most straightforward way to do this is to put data and operations into some form of containers. The great idea of object-oriented programming is to make these containers fully general, so that they can contain data as well as operations, and that they are themselves values that can be stored in other containers, or passed as parameters to operations. Such containers are called objects. Alan Kay, the inventor of Smalltalk, remarked that in this way the simplest object has the same construction principle as a full computer: it combines data with operations under a formalized interface.⁶ So objects have a lot to do with language scalability: the same techniques apply to the construction of small as well as large programs.

⁶Kay, "The Early History of Smalltalk" [Kay96]

Even though object-oriented programming has been mainstream for a long time, there are relatively few languages that have followed Smalltalk in pushing this construction principle to its logical conclusion. For instance, many languages admit values that are not objects, such as the primitive values in Java. Or they allow static fields and methods that are not members of any object. These deviations from the pure idea of object-oriented programming look quite harmless at first, but they have an annoying tendency to complicate things and limit scalability.

By contrast, Scala is an object-oriented language in pure form: every value is an object and every operation is a method call. For example, when you say `1 + 2` in Scala, you are actually invoking a method named `+` defined in class `Int`. You can define methods with operator-like names that clients of your API can then use in operator notation. This is how the designer of Scala's actors API enabled you to use expressions such as `requester ! sum` shown in the previous example: `(!)` is a method of the `Actor` class.

Scala is more advanced than most other languages when it comes to composing objects. An example is Scala's *traits*. Traits are like interfaces in Java, but they can also have method implementations and even fields. Objects are constructed by *mixin composition*, which takes the definitions of a class and adds the deltas of a number of traits to it. In this way, different aspects of classes can be encapsulated in different traits. This looks a bit like multiple inheritance, but is different when it comes to the details. Unlike a class, a trait can add a delta of functionality to an unspecified superclass. This makes traits more “pluggable” than classes. In particular, it avoids the classical “diamond inheritance” problems of multiple inheritance, which arise when the same class is inherited via several different paths.

Scala is functional

In addition to being a pure object-oriented language, Scala is also a full-blown functional language. The ideas of functional programming are older than (electronic) computers. Their foundation was laid in Alonzo Church's lambda calculus, which he developed in the 1930s. The first functional programming language was Lisp, which dates from the late 50s. Other popular functional languages are Scheme, SML, Erlang, Haskell, OCaml, and F#. For a long time, functional programming has been a bit on the sidelines, popular in academia, but not that widely used in industry. However, recent

years have seen an increased interest in functional programming languages and techniques.

Functional programming is guided by two main ideas. The first idea is that functions are first-class values. In a functional language, a function is a value of the same status as, say, an integer or a string. You can pass functions as arguments to other functions, return them as results from functions, or store them in variables. You can also define a function inside another function, just as you can define an integer value inside a function. And you can define functions without giving them a name, sprinkling your code with function literals as easily as you might write integer literals like 42.

Functions that are first-class values provide a convenient means for abstracting over operations and creating new control structures. This generalization of functions provides great expressiveness, which often leads to very legible and concise programs. It also plays an important role in Scala's scalability. As an example, the receive construct shown previously in the actor example is an invocation of a method that takes a function as argument. The code inside the receive construct is a function that is passed unexecuted into the receive method.

In most traditional languages, by contrast, functions are not values. Languages that do have function values often relegate them to second-class status. For example, the function pointers of C and C++ do not have the same status as non-functional values in those languages: function pointers can only refer to global functions, they do not give you the possibility to define first-class nested functions that refer to some values in their environment. Nor do they provide the possibility to define name-less function literals.

The second main idea of functional programming is that the operations of a program should map input values to output values rather than change data in place. To see the difference, consider the implementation of strings in Ruby and in Java. In Ruby, a string is an array of characters. Characters in a string can be changed individually. For instance you can change a semicolon character in a string to a period inside the same string object. In Java and Scala, on the other hand, a string is a sequence of characters in the mathematical sense. Replacing a character in a string using an expression like `s.replace(';', '.')` yields a new string object, which is different from `s`. Another way of expressing this is that strings are immutable in Java whereas they are mutable in Ruby. So looking at just strings, Java is a functional language, whereas Ruby is not. Immutable data structures are one

of the cornerstones of functional programming. The Scala libraries define many more immutable data types on top of those found in the Java APIs. For instance, Scala has immutable lists, tuples, maps, and sets.

Another way of stating this second idea of functional programming is that methods should not have any *side effects*. They should communicate with their environment only by taking arguments and returning results. For instance, the `replace` method in Java's `String` class fits this description. It takes a string and two characters and yields a new string where all occurrences of one character are replaced by the other. There is no other effect of calling `replace`. Methods like `replace` are called *referentially transparent*.

Functional languages encourage immutable data structures and referentially transparent methods. Some functional languages even require them. Scala gives you a choice. When you want to, you can write in an *imperative* style, which is what programming with mutable data and side effects is called. But Scala generally makes it easy to avoid imperative constructs when you want, because good functional alternatives exist.

1.3 Why Scala?

Is Scala for you? You will have to see and decide for yourself. We have found that there are actually many reasons besides scalability to like programming in Scala. Four of the most important aspects will be discussed in the following. They are: compatibility, brevity, high-level abstractions, and advanced static typing.

Scala is compatible

Scala's doesn't require you to leap backwards off the Java platform to step forward from the Java language. It allows you to add value to existing code—to build on what you already have, because it was designed for seamless interoperability with Java.⁷ Scala programs compile to JVM bytecodes. Their run-time performance is usually on par with Java programs. Scala code can call Java methods, access Java fields, inherit from Java classes, and implement Java interfaces. None of this requires special syntax, explicit interface

⁷There is also a Scala variant that runs on the .NET platform, but the JVM variant currently has better support.

descriptions, or glue code. In fact, almost all Scala code makes heavy use of Java libraries, often without programmers being aware of this fact.

Another aspect of full interoperability is that Scala heavily re-uses Java types. Scala's `Ints` are represented as Java primitive integers of type `int`, `Floats` are represented as `floats`, `Booleans` as `booleans`, and so on. Scala arrays are mapped to Java arrays. Scala also re-uses many of the standard Java library types. For instance, the type of a string literal "abc" in Scala is `java.lang.String`, and a thrown exception must be a subclass of `java.lang.Throwable`.

Scala not only re-uses Java's types, but also "dresses them up" to make them nicer. For instance, Scala's strings support methods like `toInt` or `toFloat`, which convert the string to an integer or floating point number. So you can write `str.toInt` as a shorter alternative for `Integer.parseInt(str)`. How can this be achieved without breaking interoperability? Java's `String` class certainly has no `toInt` method! In fact, Scala has a very general solution to solve this tension between advanced library design and interoperability. Scala lets you define *implicit conversions*, which are always applied when types would not normally match up, or when non-existing members are selected. In the case above, when looking for a `toInt` method on a string, the Scala compiler will find no such member of class `String`, but it will find an implicit conversion that converts a Java `String` to an instance of the Scala class `RichString`, which does define such a member. The conversion will then be applied implicitly before performing the `toInt` operation.

Scala code can also be invoked from Java code. This is sometimes a bit more subtle, because Scala is a richer language than Java, so some of Scala's more advanced features need to be encoded before they can be mapped to Java. [Chapter 27](#) explains the details.

Scala is concise

Scala programs tend to be short. Scala programmers have reported reductions in number of lines of up to a factor of ten compared to Java. These might be extreme cases. A more conservative estimate would be that a typical Scala program should have about half the number of lines of the same program written in Java. Fewer lines of code mean not only less typing, but also less effort at reading and understanding programs and fewer possibili-

ties of defects. There are several factors that contribute to this reduction in lines of code.

First, Scala's syntax avoids some of the boilerplate that burdens Java programs. For instance, semicolons are optional in Scala and are usually left out. There are also several other areas where Scala's syntax is less noisy. As an example, compare how you write classes and constructors in Java and Scala. In Java, a class with a constructor often looks like this:

```
// this is Java
class MyClass {

    private int index;
    private String name;

    public MyClass(int index, String name) {
        this.index = index;
        this.name = name;
    }
}
```

In Scala, you would likely write this instead:

```
class MyClass(index: Int, name: String) {}
```

Given this code, the Scala compiler will produce a class that has two private instance variables, an `Int` named `index` and a `String` named `name`, and a constructor that takes initial values for those variables as parameters. The code of this constructor will initialize the two instance variables with the values passed as parameters. In short, you get essentially the same functionality as the more verbose Java version.⁸ The Scala class is quicker to write, easier to read, and most importantly, less error prone than the Java class.

Scala's type inference is another factor that contributes to its conciseness. Repetitive type information can be left out, so programs become less cluttered and more readable.

But probably the most important key to compact code is code you don't have to write because it is done in a library for you. Scala gives you many tools to define powerful libraries that let you capture and factor out common

⁸The only real difference is that the instance variables produced in the Scala case will be final. You'll learn how to make them non-final in [Chapter 10](#), Composition and Inheritance.

behavior. For instance, different aspects of library classes can be separated out into traits, which can then be mixed together in flexible ways. Or library methods can be parameterized with operations, which lets you define constructs that are, in effect, your own control structures. Together, these constructs allow the definition of libraries that are both high-level and flexible to use.

Scala is high-level

Programmers are constantly grappling with complexity. To program productively, you must understand the code on which you are working. Overly complex code has been the downfall of many a software project. Unfortunately, important software usually has complex requirements. Such complexity can't be avoided; it must instead be managed.

Scala helps you manage complexity by letting you raise the level of abstraction in the interfaces you design and use. As an example, imagine you have a `String` variable `name`, and you want to find out whether or not that `String` contains an upper case character. In Java, you might write this:

```
// this is Java
boolean nameHasUpperCase = false;
for (int i = 0; i < name.length(); ++i) {
    if (Character.isUpperCase(name.charAt(i))) {
        nameHasUpperCase = true;
        break;
    }
}
```

Whereas in Scala, you could write this:

```
val nameHasUpperCase = name.exists(_.isUpperCase)
```

The Java code treats strings as low-level entities that are stepped through character by character in a loop. The Scala code treats the same strings as higher-level sequences of characters that can be queried with *predicates*. Clearly the Scala code is much shorter and—for trained eyes—easier to understand than the Java code. So the Scala code weighs less heavily on the total complexity budget. It also gives you less opportunity to make mistakes.

The predicate `(_.isUpperCase)` is an example of a function literal in Scala.⁹ It describes a function that takes a character argument (represented by the underscore character), and tests whether it is an upper case letter.¹⁰

In principle, such control abstractions are possible in Java as well. You'd need to define an interface that contains a method with the abstracted functionality. For instance, if you wanted to support querying over strings, you might invent an interface `CharacterProperty` with a single method `hasProperty`:

```
// this is Java
interface CharacterProperty {
    boolean hasProperty(char ch);
}
```

With that interface you can formulate a method `exists` in Java: It takes a string and a `CharacterProperty` and returns true if there is a character in the string that satisfies the property. You could then invoke `exists` as follows:

```
// this is Java
exists(name, new CharacterProperty {
    boolean hasProperty(char ch) {
        return Character.isUpperCase(ch);
    }
});
```

However, all this feels rather heavy. So heavy, in fact, that most Java programmers would not bother. They would just live with the increased complexity in their code. On the other hand, function literals in Scala are really lightweight, so they are used frequently. As you get to know Scala better you'll find more and more opportunities to define and use your own control abstractions. You'll find that this helps avoid code duplication and thus keeps your programs shorter and clearer.

⁹A function literal can be called a *predicate* if its result type is `Boolean`.

¹⁰This use of the underscore as a placeholder for arguments is described in [Section 8.5](#) on page 175

Scala is statically typed

A static type system classifies variables and expressions according to the kinds of values they hold and compute. Scala stands out as a language with a very advanced static type system. Starting from a system of nested class types much like Java's, it allows you to parameterize types with *generics*, to combine types using *intersections*, and to hide details of types using *abstract types*.¹¹ These give a strong foundation for building and composing your own types, so that you can design interfaces that are at the same time safe and flexible to use.

If you like dynamic languages such as Perl, Python, Ruby, or Groovy, you might find it a bit strange that Scala's static type system is listed as one of its strong points. After all, the absence of a static type system has been cited by some as a major advantage of dynamic languages. The most common counter-arguments against static types are that they make programs too verbose, that they prevent programmers from expressing themselves as they wish, and that they prevent certain patterns of dynamic modifications of software systems. However, often these counter-arguments do not go against the idea of static types in general, but against specific type systems, which are perceived to be too verbose or too inflexible. For instance, Alan Kay, the inventor of the Smalltalk language, once remarked: "I'm not against types, but I don't know of any type systems that aren't a complete pain, so I still like dynamic typing."

We'll hope to convince you in this book that Scala's type system is far from being a "complete pain." In fact, it addresses nicely two of the usual concerns about static typing: verbosity is avoided through type inference and flexibility is gained through pattern matching and several new ways to write and compose types. With these impediments out of the way, the classical benefits of static type systems can be better appreciated. Among the most important of these benefits are verifiable properties of program abstractions, safe refactorings, and better documentation.

Verifiable properties. Static type systems can prove the absence of certain run-time errors. For instance, they can prove properties like: booleans are never added to integers, private variables are not accessed from outside their

¹¹Generics are discussed in [Chapter 17](#), intersections in [Chapter 11](#), and abstract types in [Chapter 18](#).

class, functions are applied to the right number of arguments, only strings are ever added to a set of strings.

Other kinds of errors are not detected by today's static type systems. For instance, they will usually not detect array bounds violations, non-terminating functions, or divisions by zero. They will also not detect that your program does not conform to its specification (assuming there is a spec, that is!). Static type systems have therefore been dismissed by some as not being very useful. The argument goes that since such type systems can only detect simple errors, whereas unit tests provide more extensive coverage, why bother with static types at all? We believe that these arguments miss the point. Certainly a static type system cannot replace unit testing, even though it can reduce the number of unit tests that are necessary, because it takes care of some of the properties that would need to be tested otherwise. However, unit testing can not replace static typing either. After all, as Edsger Dijkstra said, testing can only prove the presence of errors, never their absence. So the guarantees that static typing gives may be simple but they are real guarantees, of the form no amount of testing can deliver.

Safe refactorings. A static type system provides a safety net that lets you make changes to a codebase with a high degree of confidence. Consider for instance a refactoring that adds an additional parameter to a method. In a statically typed language you can do the change, re-compile your system and simply fix all lines that cause a type error. Once you have finished with this, you are sure to have found all places that needed to be changed. The same holds for many other simple refactorings like changing a method name, or moving methods from one class to another. In all cases a static type check will provide enough assurance that the new system works just like the old one.

Documentation. Static types are program documentation that is checked by the compiler for correctness. Unlike a normal comment, a type annotation can never be out of date (at least not if the source file that contains it has recently passed a compiler). Furthermore, compilers and integrated development environments can make use of type annotations to provide better context help. For instance, an integrated development environment can display all the members available for a selection by determining the static type

of the expression on which the selection is made and looking up all members of that type.

Even though static types are generally useful for program documentation, they can sometimes be annoying when they clutter the program. Typically, useful documentation is what readers of a program cannot easily derive themselves. In a method definition like

```
def f(x: String) = ...
```

it's useful to know that `f`'s argument should be a `String`. On the other hand, at least one of the two annotations in the following example is annoying:

```
val x: HashMap[Int, String] = new HashMap[Int, String]()
```

Clearly, it should be enough to say just once that `x` is a `HashMap` with `Ints` as keys and `Strings` as values; there is no need to repeat the same phrase twice.

Scala has a very sophisticated type inference system that lets you omit almost all type information that's usually considered as annoying. In the example above, the following two less annoying alternatives would work as well.

```
val x = new HashMap[Int, String]()
val x: Map[Int, String] = new HashMap()
```

Type inference in Scala can go quite far. In fact, it's not uncommon for user code to have no explicit types at all. Therefore, Scala programs often look a bit like programs written in a dynamically typed scripting language. This holds particularly for client application code, which glues together pre-written library components. It's less true for the library components themselves, because these often employ fairly sophisticated types to allow flexible usage patterns. This is only natural. After all, the type signatures of the members that make up the interface of a re-usable component should be explicitly given, because they constitute an essential part of the contract between the component and its clients.

1.4 Scala’s roots

Scala’s design has been influenced by many programming languages and ideas in programming language research. In fact, only a few features of Scala are genuinely new; most have been already applied in some form in other languages. Scala’s innovations come primarily from how its constructs are put together. In this section, we list the main influences on Scala’s design. The list cannot be exhaustive—there are simply too many smart ideas around in programming language design to enumerate them all here.

At the surface level, Scala adopts a large part of the syntax of Java and C#, which in turn borrowed most of their syntactic conventions from C and C++. Expressions, statements and blocks are mostly as in Java, as is the syntax of classes, packages and imports.¹² Besides syntax, Scala adopts other elements of Java, such as its basic types, its class libraries, and its execution model.

Scala also owes much to other languages. Its uniform object model was pioneered by Smalltalk and taken up subsequently by Ruby. Its idea of universal nesting (almost every construct in Scala can be nested inside any other construct) is also present in Algol, Simula, and, more recently in Beta and gbeta. Its uniform access principle for method invocation and field selection comes from Eiffel. Its approach to functional programming is quite similar in spirit to the ML family of languages, which has SML, OCaml, and F# as prominent members. Many higher-order functions in Scala’s standard library are also present in ML or Haskell. Scala’s implicit parameters were motivated by Haskell’s type classes; they achieve analogous results in a more classical object-oriented setting. Scala’s actor-based concurrency library was heavily inspired by Erlang.

Scala is not the first language to emphasize scalability and extensibility. The historic root of extensible languages that can span different appli-

¹² The major deviation from Java concerns the syntax for type annotations—it’s “variable: Type” instead of “Type variable” in Java. Scala’s postfix type syntax resembles Pascal, Modula-2, or Eiffel. The main reason for this deviation has to do with type inference, which often lets you omit the type of a variable or the return type of a method. Using the “variable: Type” syntax this is easy—just leave out the colon and the type. But in C-style “Type variable” syntax you cannot simply leave off the type—there would be no marker to start the definition anymore. You’d need some alternative keyword to be a placeholder for a missing type (C# 3.0, which does some type inference, uses var for this purpose). Such an alternative keyword feels more ad-hoc and less regular than Scala’s approach.

cation areas is Peter Landin's 1966 paper "The Next 700 Programming Languages."¹³ (The language described in this paper, Iswim, stands beside Lisp as one of the pioneering functional languages.) The specific idea of treating an infix operator as a function can be traced back to Iswim and Smalltalk. Another important idea is to permit a function literal (or block) as a parameter, which enables libraries to define control structures. Again, this goes back to Iswim and Smalltalk. Smalltalk and Lisp have both a flexible syntax that has been applied extensively for building embedded domain-specific languages. C++ is another scalable language that can be adapted and extended through operator overloading and its template system; compared to Scala it is built on a lower-level, more systems-oriented core.

Scala is also not the first language to integrate functional and object-oriented programming, although it probably goes furthest in this direction. Other languages that have integrated some elements of functional programming into OOP include Ruby, Smalltalk, and Python. On the Java platform, Pizza, Nice, and Multi-Java have all extended a Java-like core with functional ideas. There are also primarily functional languages that have acquired an object system; examples are OCaml, F#, and PLT-Scheme.

Scala has also contributed some innovations to the field of programming languages. For instance, its abstract types provide a more object-oriented alternative to generic types, its traits allow for flexible component assembly, and its extractors provide a representation-independent way to do pattern matching. These innovations have been presented in papers at programming language conferences in recent years.¹⁴

1.5 Conclusion

In this chapter, we gave you a glimpse of what Scala is and how it might help you in your programming. To be sure, Scala is not a silver bullet that will magically make you more productive. To advance, you will need to apply Scala artfully, and that will require some learning and practice. If you're coming to Scala from Java, the most challenging aspects of learning Scala may involve Scala's type system (which is richer than Java's) and its support for functional programming. The goal of this book is to guide you gently up

¹³Landin, "The Next 700 Programming Languages" [Lan66]

¹⁴For more information, see [Ode03], [Ode05], and [Emi07] in the bibliography.

Scala's learning curve, one step at a time. We think you'll find it a rewarding intellectual experience that will expand your horizons and make you think differently about program design. Hopefully, you will also gain pleasure and inspiration from programming in Scala.

In the next chapter, we'll get you started writing some Scala code.

Chapter 2

First Steps in Scala

It's time to write some Scala code. Before we start on the in-depth Scala tutorial, we put in two chapters that will give you the big picture of Scala, and most importantly, get you writing code. We encourage you to actually try out all the code examples presented in this chapter and the next as you go. The best way to get started learning Scala is to program in it.

To run the examples in this chapter, you should have a standard Scala installation. To get one, go to <http://www.scala-lang.org/downloads> and follow the directions for your platform. You can also use a Scala plug-in for Eclipse, but for the steps in this chapter, we'll assume you're using the Scala distribution from scala-lang.org.

If you are a veteran programmer new to Scala, the next two chapters should give you enough understanding to enable you to start writing useful programs in Scala. If you are less experienced, some of the material may seem a bit mysterious to you. But don't worry. Everything will be explained in greater detail in later chapters.

Step 1. Learn to use the Scala interpreter

The easiest way to get started with Scala is by using the Scala interpreter, which is an interactive “shell” for writing Scala expressions and programs. Simply type an expression into the interpreter and it will evaluate the expression and print the resulting value. The interactive shell for Scala is simply

called `scala`. You use it by typing `scala` at a command prompt:¹

```
$ scala  
Welcome to Scala version 2.6.1-final.  
Type in expressions to have them evaluated.  
Type :help for more information.  
scala>
```

After you type an expression, such as `1 + 2`, and hit return:

```
scala> 1 + 2
```

The interpreter will print:

```
res0: Int = 3
```

This line includes:

- an automatically generated or user-defined name to refer to the computed value (`res0`, which means result 0)
- a colon (`:`)
- the type of the expression and its resulting value (`Int`)
- an equals sign (`=`)
- the value resulting from evaluating the expression (3)

The type `Int` names the class `Int` in the package `scala`. Packages in Scala are similar to packages in Java: they partition the global namespace and provide a mechanism for information hiding.² Values of class `Int` correspond to Java's `int` values. More generally, all of Java's primitive types have corresponding classes in the `scala` package. For example, `scala.Boolean` corresponds to Java's `boolean` primitive type.

¹If you're using Windows, you'll need to type the `scala` command into the "Command Prompt" DOS box. If you're on Unix, you'll need to say either "edit `scala`" or "`rlwrap scala`" to get line editing functionality in the interpreter.

²If you're not familiar with Java packages, you can think of them as providing a full name for classes. Because `Int` is a member of package `scala`, "Int" is the class's simple name, and "`scala.Int`" is its full name. The details of packages are explained in [Chapter 13](#).

`scala.Float` corresponds to Java's `float`. When you compile your Scala code to Java bytecodes, the Scala compiler will use Java's primitive types where possible to give you the performance benefits of the primitive types.

The `resX` identifier may be used in later lines. For instance, since `res0` was set to 3 previously, `res0 * 3` will be 9:

```
scala> res0 * 3
res1: Int = 9
```

To print the necessary, but not sufficient, `Hello, world!` greeting, type:

```
scala> println("Hello, world!")
Hello, world!
```

The `println` function prints the passed string to the standard output, similar to `System.out.println` in Java.

Step 2. Define some variables

Scala has two kinds of variables, `vals` and `vars`. `vals` are similar to final variables in Java. Once initialized, a `val` can never be reassigned. `vars`, by contrast, are similar to non-final variables in Java. A `var` can be reassigned throughout its lifetime. Here's a `val` definition:

```
scala> val msg = "Hello, world!"
msg: java.lang.String = Hello, world!
```

This statement introduces `msg` as a name for the `String` `"Hello world!"`. The type of `msg` is `java.lang.String`, because Scala strings are implemented by Java's `String` class.

If you're used to declaring variables in Java, you'll notice one striking difference here: neither `java.lang.String` or `String` appear anywhere in the `val` definition. This example illustrates *type inference*, Scala's ability to figure out types from context. In this case, because you initialized `msg` with a `String`, Scala inferred the type of `msg` to be `String`. When the Scala interpreter (or compiler) can infer a type, it is usually best to let it do so rather than fill the code with unnecessary, explicit type annotations. You can, however, specify a type explicitly if you wish, and sometimes you probably should. An explicit type annotation can both ensure that the Scala compiler infers

the type you intend, as well as serve as useful documentation for future readers of the code. In contrast to Java, where you specify a variable's type before its name, in Scala you specify a variable's type after its name, separated by a colon. For example:

```
scala> val msg2: java.lang.String = "Hello again, world!"  
msg2: java.lang.String = Hello again, world!
```

Or, since `java.lang` types are visible with their simple names³ in Scala programs, simply:

```
scala> val msg3: String = "Hello yet again, world!"  
msg3: String = Hello yet again, world!
```

Going back to the original `msg`, now that it is defined, you can use it as you'd expect, for example:

```
scala> println(msg)  
Hello, world!
```

What you can't do with `msg`, given that it is a `val`, not a `var`, is reassign it.⁴ For example, see how the interpreter complains when you attempt the following:

```
scala> msg = "Goodbye cruel world!"  
<console>:7: error: assignment to immutable value  
      msg = "Goodbye cruel world!"  
           ^
```

If reassignment is what you want, you'll need to use a `var`, as in:

```
scala> var greeting = "Hello, world!"  
greeting: java.lang.String = Hello, world!
```

Since `greeting` is a `var` not a `val`, you can reassign it later. If you are feeling grouchy later, for example, you could change your `greeting` to:

³The simple name of `java.lang.String` is `String`.

⁴In the interpreter, however, you can *define* a new `val` with a name that was already used before. This mechanism is explained in [Section 4.6](#).

```
scala> greeting = "Leave me alone, world!"  
greeting: java.lang.String = Leave me alone, world!
```

To enter something into the interpreter that spans multiple lines, just keep typing after the first line. If the code you typed so far is not complete, the interpreter will respond with a vertical bar on the next line.

```
scala> val multiLine =  
|   "This is the next line."  
multiLine: java.lang.String = This is the next line.
```

If you realize you have typed something wrong, but the interpreter is still waiting for more input, you can escape by pressing enter twice:

```
scala> val oops =  
|  
|  
| You typed two blank lines. Starting a new command.  
scala>
```

Step 3. Define some functions

Now that you've worked with Scala variables, you'll probably want to write some functions. Here's how you do that in Scala:

```
scala> def max(x: Int, y: Int): Int = {  
|   if (x > y)  
|     x  
|   else  
|     y  
| }  
max: (Int,Int)Int
```

Function definitions start with `def`. The function's name, in this case `max`, is followed by a comma-separated list of parameters in parentheses. A type annotation must follow every function parameter, preceded by a colon, because the Scala compiler (and interpreter, but from now on we'll just say compiler) does not infer function parameter types. In this example, the function named

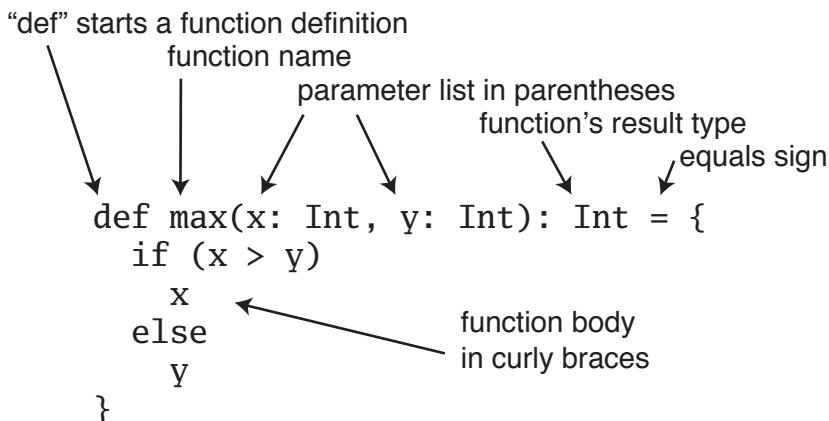


Figure 2.1: The basic form of a function definition in Scala.

`max` takes two parameters, `x` and `y`, both of type `Int`. After the close parenthesis of `max`'s parameter list you'll find another “`: Int`” type annotation. This one defines the *result type* of the `max` function itself.⁵ Following the function's result type is an equals sign and a pair of curly braces that contain the body of the function. In this case, the body contains a single `if` expression, which selects either `x` or `y`, whichever is greater, as the result of the `max` function.⁶ The equals sign that precedes the body of a function hints that in the functional world view, a function defines an expression that results in a value. The basic structure of a function is illustrated in Figure 2.1.

Sometimes the Scala compiler will require you to specify the result type of a function. If the function is *recursive*,⁷ for example, you must explicitly specify the function's result type. In the case of `max` however, you may leave the result type off and the compiler will infer it.⁸ Also, if a function consists

⁵In Java, the type of the value returned from a method is its return type. In Scala, that same concept is called *result type*.

⁶As demonstrated here, Scala's `if` expression can result in a value, similar to Java's ternary operator. For example, the Scala expression “`if (x > y) x else y`” behaves similarly to “`(x > y) ? x : y`” in Java.

⁷A function is recursive if it calls itself.

⁸Function result types is one of the cases where it is sometimes better to provide an explicit type annotation. Such type annotations can make the code easier to read, because the reader need not study the function body to figure out the inferred result type.

of just one statement, you can optionally leave off the curly braces. Thus, you could alternatively write the `max` function like this:

```
scala> def max2(x: Int, y: Int) = if (x > y) x else y
      max2: (Int,Int)Int
```

Once you have defined a function, you can call it by name, as in:

```
scala> max(3, 5)
      res6: Int = 5
```

Here's the definition of a function that takes no parameters and returns no usable result:

```
scala> def greet() = println("Hello, world!")
      greet: ()Unit
```

When you define the `greet()` function, the interpreter will respond with `greet: ()Unit`. “`greet`” is, of course, the name of the function. The empty parentheses indicates the function takes no parameters. And `Unit` is `greet`'s result type. A result type of `Unit` indicates the function returns no usable value. Scala's `Unit` type is similar to Java's `void` type, and in fact every `void`-returning method in Java is mapped to a `Unit`-returning method in Scala. Methods with the result type of `Unit`, therefore, are only executed for their side effects. In the case of `greet()`, the side effect is a friendly greeting printed to the standard output.

If a function takes no parameters, as is the case with `greet()`, you can call it with or without parentheses:

```
scala> greet()
      Hello, world!
      scala> greet // This is bad style
      Hello, world!
```

The recommended style for such function invocations is that if the function performs an operation, invoke it with empty parentheses. If the function returns a conceptual property, leave the empty parentheses off. Since the `greet` function prints a greeting to the standard output, it performs an operation and you should invoke it with parentheses, as in `greet()`. By contrast,

since the `length` method⁹ of `java.lang.String` returns a conceptual property (the length of a String), you should invoke it without parentheses, as in `msg.length`.

Moreover, when you define a function that takes no parameters and returns a conceptual property, you should leave the empty parentheses off, like this:

```
scala> def greeting = "Hello, world!"  
greeting: java.lang.String
```

Such functions are called *parameterless functions*. This style of function definition supports the *uniform access principle*, which states that client code should look the same whether it is accessing a `val` or a parameterless function.¹⁰ Although `greeting` is a function, it can only be invoked without parentheses:

```
scala> println(greeting)  
Hello, world!
```

If you attempt to invoke `greeting` with an empty parameter list, your program will not compile, even though `greeting` is a function:

```
scala> println(greeting())  
<console>:6: error: wrong number of arguments for method  
      apply: (Int)Char in class RichString  
          println(greeting())  
                  ^
```

The uniform access principle allows you to change a parameterless function definition into a `val` without needing to change client code. One benefit of requiring an equals sign before the body of a function is that you can change a parameterless function definition into a `val` definition simply by changing the `def` into a `val`, for example:

```
scala> val greeting = "Hello, world!"  
greeting: java.lang.String = Hello, world!
```

⁹A *method* is a function that is a member of a class, Java interface, Scala *trait*, etc. In other words, a method is a “member function.”

¹⁰The uniform access principle will be discussed in [Section 10.3 on page 207](#).

Even though `greeting` is a `val` now instead of a `def`, client code still looks the same:

```
scala> println(greeting)
Hello, world!
```

In the next step, you'll place Scala code in a file and run it as a script. If you wish to exit the interpreter, you can do so by entering `:quit` or `:q`.

```
scala> :quit
$
```

Step 4. Write some Scala scripts

Although Scala is designed to help programmers build very large-scale systems, it also scales down nicely to scripting. A script is just a sequence of statements in a file that will be executed sequentially. Put this into a file named `hello.scala`:

```
println("Hello, world, from a script!")
```

then run:

```
$ scala hello.scala
```

And you should get yet another greeting:

```
Hello, world, from a script!
```

Command line arguments to a Scala script are available via a Scala array named `args`. In Scala, arrays are zero based, as in Java, but you access an element by specifying an index in parentheses rather than square brackets. So the first element in a Scala array named `steps` is `steps(0)`, not `steps[0]`, as in Java. To try this out, type the following into a new file named `helloarg.scala`:

```
// Say hello to the first argument
println("Hello, " + args(0) + "!")
```

then run:

```
$ scala helloarg.scala planet
```

In this command, "planet" is passed as a command line argument, which is accessed in the script as `args(0)`. Thus, you should see:

```
Hello, planet!
```

Note also that this script included a comment. As with Java, the Scala compiler will ignore characters between `//` and the next end of line, as well as any characters between `/*` and `*/`. This example also shows `Strings` being concatenated with the `+` operator. This works as you'd expect. The expression `"Hello, " + "world!"` will result in the string `"Hello, world!"`.

By the way, if you're on some flavor of Unix, you can run a Scala script as a shell script by prepending a "pound bang" directive at the top of the file. For example, type the following into a file named `helloarg`:

```
#!/bin/sh
exec scala "$0" "$@"
!#
// Say hello to the first argument
println("Hello, " + args(0) + "!")
```

The initial `#!/bin/sh` must be the very first line in the file. Once you set its execute permission:

```
$ chmod +x helloarg
```

You can run the Scala script as a shell script by simply saying:

```
$ ./helloarg globe
```

Which should yield:

```
Hello, globe!
```

If you're on Windows, you can achieve the same effect by placing this at the top of your script:

```
:::\#!  
@echo off  
call scala \%0 \%\*  
goto :eof  
:::\#
```

Step 5. Loop with while, decide with if

You write while loops in Scala in much the same way as in Java. Try out a while by typing the following into a file name `printargs.scala`:

```
var i = 0  
while (i < args.length) {  
    println(args(i))  
    i += 1  
}
```

This script starts with a variable definition, `var i = 0`. Type inference gives `i` the type `scala.Int`, because that is the type of its initial value, 0. The `while` construct on the next line causes the *block* (the code between the curly braces) to be repeatedly executed until the boolean expression `i < args.length` is false. `args.length` gives the length of the `args` array, similar to the way you get the length of an array in Java. The block contains two statements, each indented two spaces, the recommended indentation style for Scala. The first statement, `println(args(i))`, prints out the `i`th command line argument. The second statement, `i += 1`, increments `i` by one. Note that Java's `++i` and `i++` don't work in Scala. To increment in Scala, you need to say either `i = i + 1` or `i += 1`. Run this script with the following command:

```
$ scala printargs.scala Scala is fun
```

And you should see:

```
Scala  
is  
fun
```

For even more fun, type the following code into a new file named `echoargs.scala`:

```
var i = 0
while (i < args.length) {
    if (i != 0)
        print(" ")
    print(args(i))
    i += 1
}
println()
```

In this version, you've replaced the `println` call with a `print` call, so that all the arguments will be printed out on the same line. To make this readable, you've inserted a single space before each argument except the first via the `if (i != 0)` construct. Since `i != 0` will be `false` the first time through the `while` loop, no space will get printed before the initial argument. Lastly, you've added one more `println` to the end, to get a line return after printing out all the arguments. Your output will be very pretty indeed. If you run this script with the following command:

```
$ scala echoargs.scala Scala is even more fun
```

You'll get:

```
Scala is even more fun
```

Note that in Scala, as in Java, you must put the boolean expression for a `while` or an `if` in parentheses. (In other words, you can't say in Scala things like `if i < 10` as you can in a language such as Ruby. You must say `if (i < 10)` in Scala.) Another similarity to Java is that if a block has only one statement, you can optionally leave off the curly braces, as demonstrated by the `if` statement in `echoargs.scala`. And although you haven't seen any of them, Scala does use semi-colons to separate statements as in Java, except that in Scala the semi-colons are very often optional, giving some welcome relief to your right little finger. If you had been in a more verbose mood, therefore, you could have written the `echoargs.scala` script as follows:

```
var i = 0;
```

```
while (i < args.length) {  
    if (i != 0) {  
        print(" ");  
    }  
    print(args(i));  
    i += 1;  
}  
println();
```

If you type the previous code into a new file named `echoargsverbosely.scala`, and run it with the command:

```
$ scala echoargsverbosely.scala In Scala semicolons are often optional
```

You should see the output:

```
In Scala semicolons are often optional
```

Note that because you had no parameters to pass to the `println` function, you could have left off the parentheses and the compiler would have been perfectly happy. But given the style guideline that you should always use parentheses when calling functions that may have side effects—coupled with the fact that by printing to the standard output, `println` will indeed have side effects—you specified the parentheses even in the concise `echoargs.scala` version.

Step 6. Iterate with `foreach` and `for`

Although you may not have realized it, when you wrote the while loops in the previous step, you were programming in an *imperative* style. In the imperative style, which is the style you would ordinarily use with languages like Java, C++, and C, you give one imperative command at a time, iterate with loops, and often mutate state shared between different functions. Scala enables you to program imperatively, but as you get to know Scala better, you'll likely often find yourself programming in a more *functional* style. In fact, one of the main aims of this book is to help you become as comfortable with the functional style as you are with imperative style.

One of the main characteristics of a functional language is that functions are first class constructs, and that's very true in Scala. For example, another (far more concise) way to print each command line argument is:

```
args.foreach(arg => println(arg))
```

In this code, you call the `foreach` method on `args`, and pass in a function. In this case, you're passing in a *function literal* that takes one parameter named `arg`. The body of the function is `println(arg)`. If you type the above code into a new file named `pa.scala`, and execute with the command:

```
$ scala pa.scala Concise is nice
```

You should see:

```
Concise  
is  
nice
```

In the previous example, the Scala interpreter infers the type of `arg` to be `String`, since `String` is the element type of the array on which you're calling `foreach`. If you'd prefer to be more explicit, you can mention the type name, but when you do you'll need to wrap the argument portion in parentheses (which is the normal form of the syntax anyway). Try typing this into a file named `epa.scala`.

```
args.foreach((arg: String) => println(arg))
```

Running this script has the same behavior as the previous one. With the command:

```
$ scala epa.scala Explicit can be nice too
```

You'll get:

```
Explicit  
can  
be  
nice  
too
```

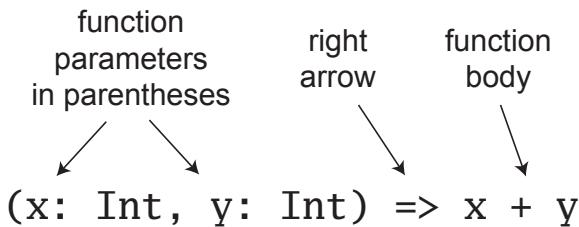


Figure 2.2: The syntax of a function literal in Scala.

If instead of an explicit mood, you're in the mood for even more conciseness, you can take advantage of a special shorthand in Scala. If a function literal consists of one statement that takes a single argument, you need not explicitly name and specify the argument.¹¹ Thus, the following code also works:

```
args.foreach(println)
```

To summarize, the syntax for a function literal is a list of named parameters, in parentheses, a right arrow, and then the body of the function. This syntax is illustrated in [Figure 2.2](#).

Now, by this point you may be wondering what happened to those trusty for loops you have been accustomed to using in imperative languages such as Java. In an effort to guide you in a functional direction, only a functional relative of the imperative for (called a *for expression*) is available in Scala. While you won't see their full power and expressiveness until you reach (or peek ahead to) [Section 7.3 on page 156](#), we'll give you a glimpse here. In a new file named `forprintargs.scala`, type the following:

```
for (arg <- args)
    println(arg)
```

The parentheses after the for in this for expression contain

¹¹This shorthand is called a *partially applied function*, and is described in [Section 8.6 on page 177](#).

arg <- args.¹² To the right of <- is the familiar args array. To the left of the <- symbol is “arg,” the name of a val (not a var). For each element of the args array, a new arg val will be created and initialized to the element value, and the body of the for will be executed. Scala’s for expressions can do much more than this, but this simple form is similar in functionality to Java 5’s:

```
for (String arg : args) {      // This is Java
    System.out.println(arg);
}
```

When you run the forprintargs.scala script with the command:

```
$ scala forprintargs.scala for is functional
```

You should see:

```
for
is
functional
```

Conclusion

In this chapter, you learned some Scala basics and, hopefully, took advantage of the opportunity to write a bit of Scala code. In the next chapter, we’ll continue this introductory overview and get into more advanced topics.

¹²You can say “in” for the <- symbol. You’d read for (arg <- args), therefore, as “for arg in args.”

Chapter 3

Next Steps in Scala

This chapter continues the previous chapter’s introduction to Scala. In this chapter, we’ll introduce some more advanced features. When you complete this chapter, you should have enough knowledge to enable you to start writing useful scripts in Scala. As with the previous chapter, we recommend you try out these examples as you go. The best way to start getting a feel for Scala is to start writing in it.

Step 7. Understand the importance of `vals`

As mentioned in [Chapter 1](#), Scala allows you to program in an *imperative* style, but encourages you to adopt a more *functional* style. If you are coming to Scala from an imperative background—for example, if you are a Java programmer—one of the main challenges you will likely face when learning Scala is figuring out how to program in the functional style. We realize this transition can be difficult, and in this book we try hard to guide you through it. But it will require some work on your part, and we encourage you to make the effort. If you come from an imperative background, we believe that learning to program in a functional style will not only make you a better Scala programmer, it will expand your horizons and make you a better programmer in general.

The first step along the path to a more functional style is to recognize the difference between the two styles in code. One way to think about this is that if code contains any vars, variables that can be reassigned, it is probably in the imperative style. If the code contains no vars at all—i.e., it contains *only*

vals—it is probably in the functional style. Thus one way to move towards a functional style is to try to program without any vars.

For example, the following while loop example, taken from [Chapter 2](#), uses a var and is therefore in the imperative style:

```
var i = 0
while (i < args.length) {
    println(args(i))
    i += 1
}
```

You can transform this bit of code into a functional style by getting rid of the var, for example, like this:

```
for (arg <- args)
    println(arg)
```

or this:

```
args.foreach(println)
```

This example illustrates the main benefit of programming in a functional style. The functional code is more concise, more clear, and less error-prone than the corresponding imperative code. The reason Scala encourages a functional style, in fact, is that the functional style can help you write more understandable, less error prone code.

That said, bear in mind that vars are not evil. Scala is not a pure functional language that forces you to program everything in the functional style. Scala is a hybrid imperative/functional language. You may find that in some situations an imperative style is a better fit for the problem at hand, and in such cases you should not hesitate to use it. In fact, the for expression and foreach examples above are not *purely* functional, because they call `println`, a method that has side effects. They are simply more functional than the while loop that uses a var.

The attitude we suggest you adopt is to be suspicious of vars in your code. Challenge them. If there isn't a good justification for a particular var, try and find a way to do the same thing without any vars.

In short, you should prefer vals over vars in your code. For someone coming from an imperative background, this can be easier said than done.

To help you learn how, we'll show you many specific examples of code with vars and how to transform those vars to vals in [Chapter 7](#).

Step 8. Parameterize Arrays with types

In addition to being functional, Scala is object-oriented. In Scala, as in Java, you define a blueprint for objects with classes. From a class blueprint, you can instantiate objects, or class instances, by using new. For example, the following Scala code instantiates a new String and prints it out:

```
val s = new String("Hello, world!")
println(s)
```

In the previous example, you *parameterize* the String instance with the initial value "Hello, world!". Parameterization means configuring an instance at the point in your program that you create that instance. You configure an instance with values by passing objects to a constructor of the instance in parentheses, just like you do when you create an instance in Java. If you place the previous code in a new file named paramwithvalues.scala and run it with scala paramwithvalues.scala, you'll see the familiar Hello, world! greeting printed out.

In addition to parameterizing instances with values at the point of instantiation, you can also parameterize them with types. This kind of parameterization is akin to specifying a type in angle brackets when instantiating a generic type in Java 5 and beyond. The main difference is that instead of the angle brackets used for this purpose in Java, in Scala you use square brackets. Here's an example:

```
val greetStrings = new Array[String](3)
greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"
for (i <- 0 to 2)
  print(greetStrings(i))
```

In this example, greetStrings is a value of type Array[String] (say this as, “an array of string”) that is initialized to length 3 by passing the

value 3 to a constructor in parentheses in the first line of code. Type this code into a new file called `paramwithtypes.scala` and execute it with `scala paramwithtypes.scala`, and you'll see yet another `Hello, world!` greeting. Note that when you parameterize an instance with both a type and a value, the type comes first in its square brackets, followed by the value in parentheses.

Had you been in a more explicit mood, you could have specified the type of `greetStrings` explicitly like this:

```
val greetStrings: Array[String] = new Array[String](3)
```

Given Scala's type inference, this line of code is semantically equivalent to the actual first line of code in `paramwithtypes.scala`. But this form demonstrates that while the type parameterization portion (the type names in square brackets) forms part of the type of the instance, the value parameterization part (the values in parentheses) does not. The type of `greetStrings` is `Array[String]`, not `Array[String](3)`.

The next three lines of code in `paramwithtypes.scala` initialize each element of the `greetStrings` array:

```
greetStrings(0) = "Hello"  
greetStrings(1) = ", "  
greetStrings(2) = "world!\n"
```

As mentioned previously, arrays in Scala are accessed by placing the index inside parentheses, not square brackets as in Java. Thus the zeroth element of the array is `greetStrings(0)`, not `greetStrings[0]` as in Java.

These three lines of code illustrate an important concept to understand about Scala concerning the meaning of `val`. When you define a variable with `val`, the variable can't be reassigned, but the object to which it refers could potentially still be mutated. So in this case, you couldn't reassign `greetStrings` to a different array; `greetStrings` will always point to the same `Array[String]` instance with which it was initialized. But you *can* change the elements of that `Array[String]` over time, so the array itself is mutable.

The final two lines in `paramwithtypes.scala` contain a `for` expression that prints out each `greetStrings` array element in turn.

```
for (i <- 0 to 2)
```

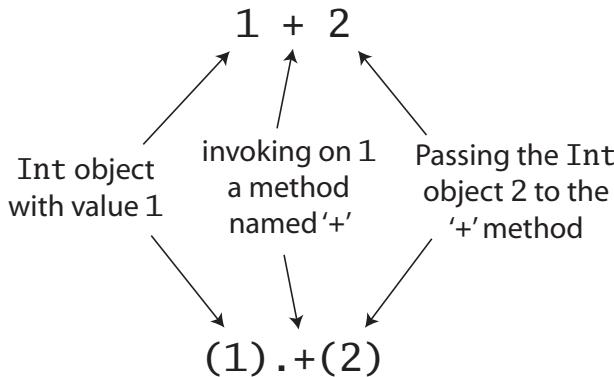


Figure 3.1: All operations are method calls in Scala.

```
print(greetStrings(i))
```

The first line of code in this for expression illustrates another general rule of Scala: if a method takes only one parameter, you can call it without a dot or parentheses. `to` is actually a method that takes one `Int` argument. The code `0 to 2` is transformed into the method call `(0) .to(2)`. (This `to` method actually returns not an `Array` but a Scala *iterator* containing the values 0, 1, and 2, which the for expression iterates over. Iterators will be described in Chapter 15.) Scala doesn't technically have operator overloading, because it doesn't actually have operators in the traditional sense. Characters such as `+`, `-`, `*`, and `/`, have no special meaning in Scala, but they can be used in method names. Thus, when you typed `1 + 2` into the Scala interpreter in Step 1, you were actually invoking a method named `+` on the `Int` object `1`, passing in `2` as a parameter. As illustrated in Figure 3.1, you could alternatively have written `1 + 2` using traditional method invocation syntax, `(1) .+(2)`.

Another important idea illustrated by this example will give you insight into why arrays are accessed with parentheses in Scala. Scala has fewer special cases than Java. Arrays are simply instances of classes like any other class in Scala. When you apply parentheses to a variable (a `val` or a `var`) and pass in a single argument, Scala will transform that into an invocation of a method named `apply`. So `greetStrings(i)` gets transformed into `greetStrings.apply(i)`. Thus accessing the element of an array in

Scala is simply a method call like any other method call. What's more, the compiler will transform *any* application of parentheses with a single argument on any type into an apply method call, not just arrays. Of course it will compile only if that type actually defines an apply method. So it's not a special case; it's a general rule.

Similarly, when an assignment is made to a variable to which parentheses and a single argument have been applied, the compiler will transform that into an invocation of an update method that takes two parameters. For example,

```
greetStrings(0) = "Hello"
```

will be transformed into

```
greetStrings.update(0, "Hello")
```

Thus, the following Scala code is semantically equivalent to the code you typed into `paramwithtypes.scala`:

```
val greetStrings = new Array[String](3)  
greetStrings.update(0, "Hello")  
greetStrings.update(1, ", ")  
greetStrings.update(2, "world!\n")  
  
for (i <- 0.to(2))  
    print(greetStrings.apply(i))
```

Scala achieves a conceptual simplicity by treating everything, from arrays to expressions, as objects with methods. You as the programmer don't have to remember lots of special cases, such as the differences in Java between primitive and their corresponding wrapper types, or between arrays and regular objects. However, it is significant to note that in Scala this uniformity does not incur the significant performance cost that it often has in other languages that have aimed to be pure in their object orientation. The Scala compiler uses Java arrays, primitive types, and native arithmetic where possible in the compiled code.

Step 9. Use Lists and Tuples

One of the big ideas of the functional style of programming is that methods should not have side effects. The only effect of a method should be to compute the value or values that are returned by the method. Some benefits gained when you take this approach are that methods become less entangled, and therefore more reliable and reusable. Another benefit of the functional style in a statically typed language is that everything that goes into and out of a method is checked by a type checker, so logic errors are more likely to manifest themselves as type errors. To apply this functional philosophy to the world of objects, you would make objects immutable. A simple example of an immutable object in Java is `String`. If you create a `String` with the value "Hello, ", it will keep that value for the rest of its lifetime. If you later call `concat("world!")` on that `String`, it will not add "world!" to itself. Instead, it will create and return a brand new `String` with the value "Hello, world!".

As you've seen, a Scala `Array` is a mutable sequence of objects that all share the same type. An `Array[String]` contains only `Strings`, for example. Although you can't change the length of an `Array` after it is instantiated, you can change its element values. Thus, `Arrays` are mutable objects. An immutable, and therefore more functional-oriented, sequence of objects that share the same type is Scala's `List`. As with `Arrays`, a `List[String]` contains only `Strings`. Scala's `List`, `scala.List`, differs from Java's `java.util.List` type in that Scala `Lists` are always immutable (whereas Java `Lists` can be mutable). But more importantly, Scala's `List` is designed to enable a functional style of programming. Creating a `List` is easy, you just say:

```
val oneTwoThree = List(1, 2, 3)
```

This establishes a new `val` named `oneTwoThree`, which is initialized with a new `List[Int]` with the integer element values 1, 2 and 3.¹ Because `Lists` are immutable, they behave a bit like Java `Strings` in that when you call a method on one that might seem by its name to imply the `List` will be mutated, it instead creates a new `List` with the new value and returns it. For

¹You don't need to say `new List` because "`List.apply()`" is defined as a factory method on the `scala.List` *companion object*. You'll read more on companion objects in Step 11.

example, `List` has a method named `:::` that prepends a passed `List` to the `List` on which `:::` was invoked. Here's how you use it:

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo :: threeFour
println(oneTwo + " and " + threeFour + " were not mutated.")
println("Thus, " + oneTwoThreeFour + " is a new List.")
```

Type this code into a new file called `listcat.scala` and execute it with `scala listcat.scala`, and you should see:

```
List(1, 2) and List(3, 4) were not mutated.
Thus, List(1, 2, 3, 4) is a new List.
```

Enough said.² Perhaps the most common operator you'll use with `Lists` is `::`, which is pronounced “cons.” `Cons` prepends a new element to the beginning of an existing `List`, and returns the resulting `List`.³ For example, if you type the following code into a file named `consit.scala`:

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
```

And execute it with `scala consit.scala`, you should see:

```
List(1, 2, 3)
```

²Actually, you may have noticed something amiss with the associativity of the `:::` method (or suspected “prepend” was a typo), but it is actually a simple rule to remember. If a method is used in operator notation, as in `a * b` or `a :::: b`, the method is invoked on the left hand operand, as in `a.*(b)`, unless the method name ends in a colon. If the method name ends in a colon, then the method is invoked on the right hand operand. For example, in `a :::: b`, the `::::` method is invoked on `a`, passing in `b`, like this: `b ::::(a)`. Thus, `b` prepends `a` to itself and returns the result.

³Class `List` does not offer an append operation, because the time it takes to append to a `List` grows linearly with the size of the `List`, whereas prepending takes constant time. Your options if you want to build a list by appending elements is to prepend them, then when you're done call `reverse`, or use a `ListBuffer`, a mutable `List` that does offer an append operation, and call `toList` when you're done. `ListBuffer` will be described in Section 20.2 on page 418.

Given that a shorthand way to specify an empty List is `Nil`, one way to initialize new Lists is to string together elements with the `cons` operator, with `Nil` as the last element.⁴ For example, if you type the following code into a file named `consinit.scala`:

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
```

And execute it with `scala consinit.scala`, you should again see:

```
List(1, 2, 3)
```

Scala's List is packed with useful methods, many of which are shown in [Table 3.1](#).

Table 3.1: Some List methods and usages.

What it is	What it does
<code>List()</code>	Creates an empty List
<code>Nil</code>	An empty List
<code>List("Cool", "tools", "rule")</code>	Creates a new List[String] with the three values "Cool", "tools", and "rule"
<code>val thrill = "Will" :: "fill" :: "until" :: Nil</code>	Creates a new List[String] with the three values "Will", "fill", and "until"
<code>List("a", "b") ::: List("c", "d")</code>	Creates a new List[String] with values "a", "b", "c", and "d"
<code>thrill(2)</code>	Returns the 2 nd element (zero based) of the <code>thrill</code> List (returns "until")
<code>thrill.count(s => s.length == 4)</code>	Counts the number of String elements in <code>thrill</code> that have length 4 (returns 2)

⁴The reason you need `Nil` at the end is that `::` is defined on class `List`. If you try to just say `1 :: 2 :: 3`, it won't compile because 3 is an Int, which doesn't have a `::` method.

Table 3.1: continued

<code>thrill.drop(2)</code>	Returns the <code>thrill</code> List without its first 2 elements (returns <code>List("until")</code>)
<code>thrill.dropRight(2)</code>	Returns the <code>thrill</code> List without its rightmost 2 elements (returns <code>List("Will")</code>)
<code>thrill.exists(s => s == "until")</code>	Determines whether a String element exists in <code>thrill</code> that has the value "until" (returns true)
<code>thrill.filter(s => s.length == 4)</code>	Returns a List of all elements, in order, of the <code>thrill</code> List that have length 4 (returns <code>List("Will", "fill")</code>)
<code>thrill.forall(s => s.endsWith("l"))</code>	Indicates whether all elements in the <code>thrill</code> List end with the letter "l" (returns true)
<code>thrill.foreach(s => print(s))</code>	Executes the <code>print</code> statement on each of the Strings in the <code>thrill</code> List (prints "Willfilluntil")
<code>thrill.foreach(print)</code>	Same as the previous, but more concise (also prints "Willfilluntil")
<code>thrill.head</code>	Returns the first element in the <code>thrill</code> List (returns "Will")
<code>thrill.init</code>	Returns a List of all but the last element in the <code>thrill</code> List (returns <code>List("Will", "fill")</code>)
<code>thrill.isEmpty</code>	Indicates whether the <code>thrill</code> List is empty (returns false)
<code>thrill.last</code>	Returns the last element in the <code>thrill</code> List (returns "until")

Table 3.1: continued

<code>thrill.length</code>	Returns the number of elements in the <code>thrill</code> List (returns 3)
<code>thrill.map(s => s + "y")</code>	Returns a List resulting from adding a "y" to each String element in the <code>thrill</code> List (returns <code>List("Willy", "filly", "untily")</code>)
<code>thrill.remove(s => s.length == 4)</code>	Returns a List of all elements, in order, of the <code>thrill</code> List <i>except those</i> that have length 4 (returns <code>List("until")</code>)
<code>thrill.reverse</code>	Returns a List containing all elements of the <code>thrill</code> List in reverse order (returns <code>List("until", "fill", "Will")</code>)
<code>thrill.sort((s, t) => s.charAt(0).toLowerCase < t.charAt(0).toLowerCase)</code>	Returns a List containing all elements of the <code>thrill</code> List in alphabetical order of the first character lowercased (returns <code>List("fill", "until", "Will")</code>)
<code>thrill.tail</code>	Returns the <code>thrill</code> List minus its first element (returns <code>List("fill", "until")</code>)

Besides `List`, one other ordered collection of object elements that's very useful in Scala is the *tuple*. Like `Lists`, tuples are immutable, but unlike `Lists`, tuples can contain different types of elements. Thus whereas a `List` might be a `List[Int]` or a `List[String]`, a tuple could contain both an `Int` and a `String` at the same time. Tuples are very useful, for example, if you need to return multiple objects from a method. Whereas in Java, you would often create a JavaBean-like class to hold the multiple return values, in Scala you can simply return a tuple. And it is simple: to instantiate a new tuple that holds some objects, just place the objects in parentheses, separated by commas. Once you have a tuple instantiated, you can access its elements individually with a dot, underscore, and the one-based index of the element. For

example, type the following code into a file named `luftballons.scala`:

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

In the first line of this example, you create a new tuple that contains an `Int` with the value `99` as its first element, and a `String` with the value `"Luftballons"` as its second element. Scala infers the type of the tuple to be `Tuple2[Int, String]`, and gives that type to the variable `pair` as well. In the second line, you access the `_1` field, which will produce the first element, `99`. The `“.”` in the second line is the same dot you'd use to access a field or invoke a method. In this case you are accessing a field named `_1`.⁵ If you run this script with `scala luftballons.scala`, you'll see:

```
99
Luftballons
```

The actual type of a tuple depends on the number of elements it contains and the types of those elements. Thus, the type of `(99, "Luftballons")` is `Tuple2[Int, String]`. The type of `('u', 'r', "the", 1, 4, "me")` is `Tuple6[Char, Char, String, Int, Int, String]`.⁶

Step 10. Use Sets and Maps

Because Scala aims to help you take advantage of both functional and imperative styles, its collections libraries make a point to differentiate between mutable and immutable collection classes. For example, `Arrays` are always mutable, whereas `Lists` are always immutable. When it comes to `Sets` and `Maps`, Scala also provides mutable and immutable alternatives, but in a different way. For `Sets` and `Maps`, Scala models mutability in the class hierarchy.

⁵You may be wondering why you can't access the elements of a tuple like the elements of a `List`, such as `pair(0)`. The `apply` method always returns the same type for a `List`, but each element of a tuple may be a different type. `_1` can have one result type, `_2` another, and so on. These `_N` numbers are one-based, instead of zero-based, because starting with `1` is a tradition set by other languages with statically typed tuples, such as Haskell and ML.

⁶Although conceptually you could create tuples of any length, currently the Scala library only defines them up to `Tuple22`.

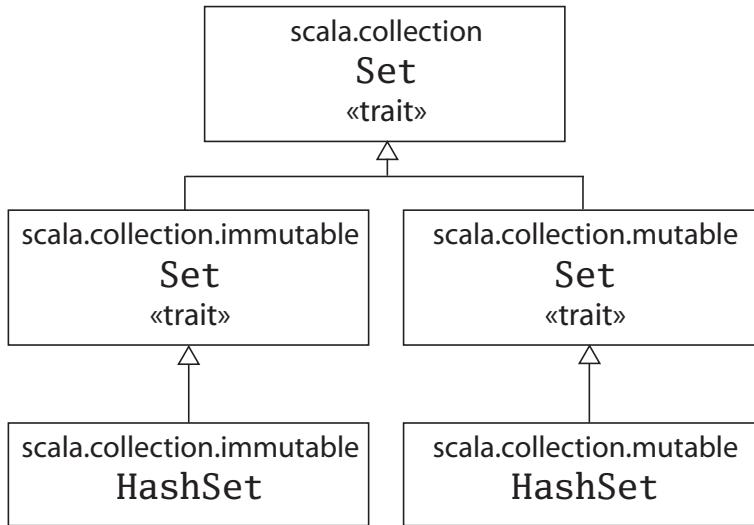


Figure 3.2: Class hierarchy for Scala Sets.

For example, the Scala API contains a base *trait* for Sets, where a trait is similar to a Java interface. (You’ll find out more about traits in Step 12.) Scala then provides two subtraits, one for mutable Sets and another for immutable Sets. As you can see in Figure 3.2, these three traits all share the same simple name, `Set`. Their fully qualified names differ, however, because each resides in a different package. Concrete Set classes in the Scala API, such as the `HashSet` classes shown in Figure 3.2, extend either the mutable or immutable Set trait. (Although in Java you “implement” interfaces, in Scala you “extend” traits.) Thus, if you want to use a `HashSet`, you can choose between mutable and immutable varieties depending upon your needs.

To try out Scala Sets, type the following code into file `jetset.scala`:

```
import scala.collection.mutable.HashSet  
val jetSet = new HashSet[String]  
jetSet += "Lear"  
jetSet += ("Boeing", "Airbus")  
println(jetSet.contains("Cessna"))
```

The first line of `jetSet.scala` imports the mutable `HashSet`. As with Java, the import allows you to use the simple name of the class, `HashSet`, in that source file. After a blank line, the third line initializes `jetSet` with a new `HashSet` that will contain only `Strings`. Note that just as with `Lists` and `Arrays`, when you create a `Set`, you need to parameterize it with a type (in this case, `String`), since every object in a `Set` must share the same type. The subsequent two lines add three objects to the mutable `Set` via the `+=` method. As with most other symbols you've seen that look like operators in Scala, `+=` is actually a method defined on class `HashSet`. Had you wanted to, instead of writing `jetSet += "Lear"`, you could have written `jetSet.+=("Lear")`. Because the `+=` method takes a variable number of arguments, you can pass one or more objects at a time to it. For example, `jetSet += "Lear"` adds one `String` to the `HashSet`, but `jetSet += ("Boeing", "Airbus")` adds two `Strings` to the `Set`.⁷ Finally, the last line prints out whether or not the `Set` contains a particular `String`. (As you'd expect, it prints `false`.)

If you want an immutable `Set`, you can take advantage of a factory method defined in `scala.collection.Set`'s companion object, which is imported automatically into every Scala source file. Just say:

```
val movieSet = Set("Hitch", "Poltergeist", "Shrek")
println(movieSet)
```

Another useful collection class in Scala is `Map`. As with `Sets`, Scala provides mutable and immutable versions of `Map`, using a class hierarchy. As you can see in [Figure 3.3](#), the class hierarchy for `Maps` looks a lot like the one for `Sets`. There's a base `Map` trait in package `scala.collection`, and two subtrait `Maps`: a mutable `Map` in `scala.collection.mutable` and an immutable one in `scala.collection.immutable`.

Implementations of `Map`, such as the `HashMaps` shown in the class hierarchy in [Figure 3.3](#), implement either the `mutable` or `immutable` trait. To see a `Map` in action, type the following code into a file named `treasure.scala`:

⁷Although `("Boeing", "Airbus")` by itself looks like a tuple containing two `Strings`, as used here it is a parameter list to `jetSet`'s `+=` method. In other words, the statement could have also been written like this: `jetSet.+=("Boeing", "Airbus")`. If you ever want to pass a tuple into a function that expects one, you'll need two sets of parentheses, as in `functionThatTakesATuple(("a", "b"))`. The outer parentheses enclose the parameter list; the inner ones define the tuple.

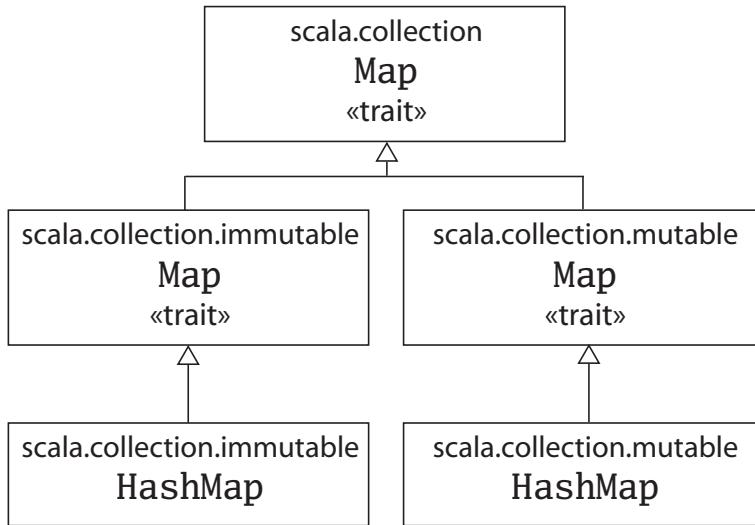


Figure 3.3: Class hierarchy for Scala Maps.

```
import scala.collection.mutable.HashMap  
  
val treasureMap = new HashMap[Int, String]  
treasureMap += (1 -> "Go to island.")  
treasureMap += (2 -> "Find big X on ground.")  
treasureMap += (3 -> "Dig.")  
println(treasureMap(2))
```

On the first line of `treasure.scala`, you import the mutable form of `HashMap`. After a blank line, you define a `val` named `treasureMap` and initialize it with a new mutable `HashMap` whose keys will be `Ints` and values `Strings`. On the next three lines you add key/value pairs to the `HashMap` using the `->` method. As illustrated in previous examples, the Scala compiler transforms a binary operation expression like `1 -> "Go to island."` into `(1).->("Go to island.")`. Thus, when you say `1 -> "Go to island."`, you are actually calling a method named `->` on an `Int` with the value `1`, and passing in a `String` with the value `"Go to island."` This `->` method, which you can invoke on any object in a Scala program, returns a two-element tuple

containing the key and value.⁸ You then pass this tuple to the `+=` method of the `HashMap` object to which `treasureMap` refers. Finally, the last line prints the value that corresponds to the key 2 in the `treasureMap`. If you run this code, it will print:

```
Find big X on ground.
```

Because maps are such a useful programming construct, Scala provides a factory method for Maps that is similar in spirit to the factory method shown in Step 9 that allows you to create Lists without using the `new` keyword. To try out this more concise way of constructing maps, type the following code into a file called `numerals.scala`:

```
val romanNumeral = Map(  
    1 -> "I",  
    2 -> "II",  
    3 -> "III",  
    4 -> "IV",  
    5 -> "V"  
)  
println(romanNumeral(4))
```

In `numerals.scala` you take advantage of the fact that the immutable `Map` trait is automatically imported into any Scala source file. Thus when you say `Map` in the first line of code, the Scala interpreter knows you mean `scala.collection.immutable.Map`. In this line, you call a factory method on the immutable `Map`'s companion object, passing in five key/value tuples as parameters.⁹ This factory method returns an instance of the immutable `HashMap` containing the passed key/value pairs. The name of the factory method is actually `apply`, but as mentioned in Step 8, if you say `Map(...)` it will be transformed by the compiler to `Map.apply(...)`. If you run the `numerals.scala` script, it will print IV.

⁸The Scala mechanism that allows you to invoke `->` on any object is called *implicit conversion*, which will be covered in [Chapter 19](#).

⁹Companion objects will be covered in Step 11.

Step 11. Understand classes and singleton objects

Up to this point you've written Scala scripts to try out the concepts presented in this chapter. For all but the simplest projects, however, you will likely want to partition your application code into classes. To give this a try, type the following code into a file called `greetSimply.scala`:

```
class SimpleGreeter {  
    val greeting = "Hello, world!"  
    def greet() = println(greeting)  
}  
  
val g = new SimpleGreeter  
g.greet()
```

`greetSimply.scala` is actually a Scala script, but one that contains a class definition. This first example, however, illustrates that as in Java, classes in Scala encapsulate fields and methods. Fields are defined with either `val` or `var`. Methods are defined with `def`. For example, in class `SimpleGreeter`, `greeting` is a field and `greet` is a method. To use the class, you initialize a `val` named `g` with a new instance of `SimpleGreeter`. You then invoke the `greet` instance method on `g`. If you run this script with `scala greetSimply.scala`, you will be dazzled with yet another `Hello, world!`.

Although classes in Scala are in many ways similar to Java, in several ways they are quite different. One difference between Java and Scala involves constructors. In Java, classes have constructors, which can take parameters, whereas in Scala, classes can take parameters directly. The Scala notation is more concise—class parameters can be used directly in the body of the class; there's no need to define fields and write assignments that copy constructor parameters into fields. This can yield substantial savings in boilerplate code, especially for small classes. To see this in action, type the following code into a file named `greetFancily.scala`:

```
class FancyGreeter(greeting: String) {  
    def greet() = println(greeting)  
}  
  
val g = new FancyGreeter("Salutations, world")
```

```
g.greet()
```

Instead of defining a constructor that takes a `String`, as you would do in Java, in `greetFancily.scala` you placed the `greeting` parameter of that constructor in parentheses placed directly after the name of the class itself, before the open curly brace of the body of class `FancyGreeter`. When defined in this way, `greeting` essentially becomes a private `val` (not a `var`—it can't be reassigned) field that's accessible anywhere inside the class body, but not accessible outside. In fact, you pass it to `println` in the body of the `greet` method. If you run this script with the command `scala greetFancily.scala`, it will inspire you with:

```
Salutations, world!
```

This is cool and concise, but what if you wanted to check the `String` used to parameterize a new `FancyGreeter` instance for `null`, and throw `NullPointerException` to abort the construction of the new instance? Fortunately, you can. Any code sitting inside the curly braces surrounding the class definition, but which isn't part of a method or variable definition, is compiled into the body of a constructor generated by the Scala compiler, which takes the class parameters as constructor parameters. This generated constructor is called the class's *primary constructor*. In essence, the primary constructor will first initialize a `val` field for each parameter in parentheses following the class name.¹⁰ It will then execute any top-level code contained in the class's body. For example, to check a passed parameter for `null`, type in the following code into a file named `greetCarefully.scala`:

```
class CarefulGreeter(greeting: String) {  
    if (greeting == null)  
        throw new NullPointerException("greeting was null")  
    def greet() = println(greeting)  
}  
  
new CarefulGreeter(null)
```

¹⁰Actually, if a class parameter is *never* used in the body of the class, the Scala compiler will optimize it away and not create a field for it.

In `greetCarefully.scala`, an `if` statement is sitting smack in the middle of the class body, something that wouldn't compile in Java. The Scala compiler places this `if` statement into the body of the primary constructor, just after code that initializes a `val` field named `greeting` with the passed value. Thus, if you pass in `null` to the primary constructor, as you do in the last line of the `greetCarefully.scala` script, the primary constructor will first initialize the `greeting` field to `null`. Then, it will execute the `if` statement that checks whether the `greeting` field is equal to `null`, which will throw a `NullPointerException`. If you run `greetCarefully.scala`, you will see a `NullPointerException` stack trace.

In Java, you sometimes give classes multiple constructors with overloaded parameter lists. You can do that in Scala as well, however you must pick one of them to be the primary constructor, and place those constructor parameters directly after the class name. You then place any additional *auxiliary constructors* in the body of the class. You define auxiliary constructors like methods named `this` that have no result type. The first statement in an auxiliary constructor must be an invocation of another constructor in the same class. To try this out, type the following code into a file named `greetRepeatedly.scala`:

```
class RepeatGreeter(greeting: String, count: Int) {  
    def this(greeting: String) = this(greeting, 1)  
  
    def greet() = {  
        for (i <- 1 to count)  
            println(greeting)  
    }  
}  
  
val g1 = new RepeatGreeter("Hello, world", 3)  
g1.greet()  
val g2 = new RepeatGreeter("Hi there!")  
g2.greet()
```

`RepeatGreeter`'s primary constructor takes not only a `String` `greeting` parameter, but also an `Int` `count` of the number of times to print the greeting. However, `RepeatGreeter` also contains a definition of an auxiliary constructor, the “`def this`” that takes a single `String` `greeting` pa-

parameter. The body of this constructor consists of a single statement: an invocation of the primary constructor parameterized with the passed greeting and a count of 1. In the final four lines of the `greetRepeatedly.scala` script, you create two `RepeatGreeter` instances, one using each constructor, and call `greet` on each. If you run `greetRepeatedly.scala`, it will print:

```
Hello, world  
Hello, world  
Hello, world  
Hi there!
```

Another area in which Scala departs from Java is that you can't have any static fields or methods in a Scala class. Instead, Scala allows you to create *singleton objects* using the keyword `object`. A singleton object cannot, and need not, be instantiated with `new`. It is essentially automatically instantiated the first time it is used, and as the "singleton" in its name implies, there is only ever one instance. A singleton object can share the same name with a class, and when it does, the singleton is called the class's *companion object*. To give this a try, type the following code into a file named `WorldlyGreeter.scala`:¹¹

```
// The WorldlyGreeter class  
class WorldlyGreeter(greeting: String) {  
    def greet() {  
        val worldlyGreeting = WorldlyGreeter.worldify(greeting)  
        println(worldyGreeting)  
    }  
}  
  
// The WorldlyGreeter companion object  
object WorldlyGreeter {  
    def worldify(s: String) = s + ", world!"  
}
```

¹¹Earlier files were scripts, and had all lower case names. This filename is in camel case because it is *not* a script, but rather will form part of an application.

In this file, you define both a class, with the `class` keyword, and a companion object, with the `object` keyword. Both types are named `WorldlyGreeter`. One way to think about this if you are coming from a Java programming perspective is that any static methods that you would have placed in class `WorldlyGreeter` in Java, you'd put in companion object `WorldlyGreeter` in Scala. Note also that in the first line of the `greet` method in class `WorldlyGreeter`, you invoke the companion object's `worldify` method using a syntax similar to the way you invoke static methods in Java: the companion object name, a dot, and the method name:

```
// Invoking a method on a singleton object from class WorldlyGreeter
// ...
val worldlyGreeting = WorldlyGreeter.worldify(greeting)
// ...
```

You could run this code in a script, but this time why not try running it in an application. Type the following code into a file named `WorldlyApp.scala`:

```
// A singleton object with a main method that allows
// this singleton object to be run as an application.
// This file can't be run from a script, because it
// ends in a definition. It must be compiled.
object WorldlyApp {
    def main(args: Array[String]) {
        val wg = new WorldlyGreeter("Hello")
        wg.greet()
    }
}
```

Because there's no class named `WorldlyApp`, this singleton object is *not* a companion object. It is instead called a *stand-alone* object. Thus, a singleton object is either a companion or a stand-alone object. The distinction is important because companion objects get a few special privileges, such as access to the private members of the like-named class. The full details of companion objects will be described in [Chapter 4](#).

One difference between Scala and Java is that whereas Java requires you to put a public class in a file named after the class—for example,

you'd put class `SpeedRacer` in file `SpeedRacer.java`—in Scala, you can name `.scala` files anything you want, no matter what Scala classes or code you put in them. In general in the case of non-scripts, however, it is recommended style to name files after the classes they contain as is done in Java, so that programmers can more easily locate classes by looking at file names. This is the approach we've taken with the two files in this example, `WorldlyGreeter.scala` and `WorldlyApp.scala`.

Neither `WorldlyGreeter.scala` nor `WorldlyApp.scala` are scripts, because they end in a definition. A script, by contrast, must end in a result expression. Thus if you try to run either of these files as a script, for example by typing:

```
scala WorldlyGreeter.scala # This won't work!
```

the Scala interpreter will complain that `WorldlyGreeter.scala` does not end in a result expression (assuming of course you didn't add any expression of your own after the `WorldlyGreeter` object definition). Instead, you'll need to actually compile these files with the Scala compiler, then run the resulting class files. One way to do this is to use `scalac`, which is the basic Scala compiler. Simply type:

```
scalac WorldlyApp.scala WorldlyGreeter.scala
```

Given that the `scalac` compiler starts up a new Java runtime instance each time it is invoked, and that the Java runtime often has a perceptible start-up delay, the Scala distribution also includes a Scala compiler *daemon* called `fsc` (for fast Scala compiler). You use it like this:

```
fsc WorldlyApp.scala WorldlyGreeter.scala
```

The first time you run `fsc`, it will create a local server daemon attached to a port on your computer. It will then send the list of files to compile to the daemon via the port, and the daemon will compile the files. The next time you run `fsc`, the daemon will already be running, so `fsc` will simply send the file list to the daemon, which will immediately compile the files. Using `fsc`, you only need to wait for the Java runtime to startup the first time. If you ever want to stop the `fsc` daemon, you can do so with `fsc -shutdown`.

Running either of these `scalac` or `fsc` commands will produce Java class files that you can then run via the `scala` command, the same command

you used to invoke the interpreter in previous examples. However, instead of giving it a filename with a `.scala` extension containing Scala code to interpret as you did in every previous example,¹² in this case you'll give it the name of a object containing a `main` method. Similar to Java, any Scala object with a `main` method that takes a single parameter of type `Array[String]` and returns `Unit` can serve as the entry point to an application.¹³ In this example, `WordlyApp` has a `main` method with the proper signature, so you can run this example by typing:

```
scala WordlyApp
```

At which point you should see:

```
Hello, world!
```

You may recall seeing this output previously, but this time it was generated in this interesting manner:

- The `scala` program fires up a Java runtime with the `WordlyApp`'s `main` method as the entry point.
- `WordlyApp`'s `main` method creates a new `WordlyGreeter` instance via `new`, passing in the string "Hello" as a parameter.
- Class `WordlyGreeter`'s primary constructor essentially initializes a final field named `greeting` with the passed value, "Hello" (this initialization code is automatically generated by the Scala compiler).
- `WordlyApp`'s `main` method initializes a local `val` named `wg` with the new `WordlyGreeter` instance.
- `WordlyApp`'s `main` method then invokes `greet` on the `WordlyGreeter` instance to which `wg` refers.
- Class `WordlyGreeter`'s `greet` method invokes `worldify` on companion object `WordlyGreeter`, passing along the value of the final field `greeting`, "Hello".

¹²The actual mechanism that the `scala` program uses to “interpret” a Scala source file is that it compiles the Scala source code to bytecodes, loads them immediately via a class loader, and executes them.

¹³As described in Step 2, `Unit` in Scala is similar to `void` in Java.

- Companion object `WorldlyGreeter`'s `worldify` method returns a `String` consisting of the value of a concatenation of the `s` parameter, which is "Hello", and the literal `String ", world!"`.
- Class `WorldlyGreeter`'s `greet` method then initializes a `val` named `worldlyGreeting` with the "Hello, world!" `String` returned from the `worldify` method.
- Class `WorldlyGreeter`'s `greet` method passes the "Hello, world!" `String` to which `worldlyGreeting` refers to `println`, which sends the cheerful greeting, via the standard output stream, to you.

Step 12. Understand traits and mixins

As first mentioned in Step 10, Scala includes a construct called a trait, which is similar in spirit to Java's interface. One main difference between Java interfaces and Scala traits is that whereas all methods in Java interfaces are by definition abstract, you can give methods real bodies with real code in Scala traits. Here's an example:

```
trait Friendly {  
    def greet() = "Hi"  
}
```

In this example, the `greet` method returns the `String` "Hi". If you are coming from Java, this `greet` method may look a little funny to you, as if `greet()` is somehow a field being initialized to the `String` value "Hi". What is actually going on is that lacking an explicit return statement, Scala methods will return the value of the last expression. In this case, the value of the last expression is "Hi", so that is returned. A more verbose way to say the same thing would be:

```
trait Friendly {  
    def greet(): String = {  
        return "Hi"  
    }  
}
```

Regardless of how you write the methods, however, the key point is that Scala traits can actually contain non-abstract methods. Another difference between Java interfaces and Scala traits is that whereas you *implement* Java interfaces, you *extend* Scala traits. Other than this implements/extends difference, however, inheritance when you are defining a new type works in Scala similarly to Java. In both Java and Scala, a class can extend one (and only one) other class. In Java, an interface can extend zero to many interfaces. Similarly in Scala, a trait can extend zero to many traits. In Java, a class can *implement* zero to many interfaces. Similarly in Scala, a class can *extend* zero to many traits; *implements* is not a keyword in Scala.

Here's an example:

```
class Dog extends Friendly {  
    override def greet() = "Woof"  
}
```

In this example, class Dog extends trait Friendly. This inheritance relationship implies much the same thing as interface implementation does in Java. You can assign a Dog instance to a variable of type Friendly. For example:

```
var pet: Friendly = new Dog  
println(pet.greet())
```

When you invoke the greet method on the Friendly pet variable, it will use dynamic binding, as in Java, to determine which implementation of the method to call. In this case, class Dog overrides the greet method, so Dog's implementation of greet will be invoked. Were you to execute the above code, you would get "Woof" (Dog's implementation of greet), not "Hi" (Friendly's implementation of greet). Note that one difference with Java is that to override a method in Scala, you must precede the method's def with override. If you attempt to override a method without specifying override, your Scala code won't compile.

Finally, one quite significant difference between Java's interfaces and Scala's traits is that in Scala, you can mix in traits at instantiation time. For example, consider the following trait:

```
trait ExclamatoryGreeter extends Friendly {
```

```
override def greet() = super.greet() + "!"  
}
```

Trait `ExclamatoryGreeter` extends trait `Friendly` and overrides the `greet` method. `ExclamatoryGreeter`'s `greet` method first invokes the superclass's `greet` method, appends an exclamation point to whatever the superclass's `greet` method returns, and returns the resulting `String`. With this trait, you can *mix in* its behavior at instantiation time using the `with` keyword. Here's an example:

```
val pup: Friendly = new Dog with ExclamatoryGreeter  
println(pup.greet())
```

Given the initial line of code, the Scala compiler will create a *synthetic class* that extends class `Dog` and trait `ExclamatoryGreeter` and instantiate it.¹⁴ When you invoke `greet` on the synthetic class instance, it will cause the correct implementation to be executed.

When you run this code, the `pup` variable will first be initialized with the new instance of the synthetic class, then when `greet` is invoked on `pup`, you'll see "Woof!". Note that had `pup` not been explicitly defined to be of type `Friendly`, the Scala compiler would have inferred the type of `pup` to be `Dog with ExclamatoryGreeter`.

To give all these concepts a try, type the following code into a file named `friendly.scala`:

```
trait Friendly {  
    def greet() = "Hi"  
}  
  
class Dog extends Friendly {  
    override def greet() = "Woof"  
}  
  
class HungryCat extends Friendly {  
    override def greet() = "Meow"  
}  
  
class HungryDog extends Dog {
```

¹⁴A class is “synthetic” if it is generated automatically by the compiler.

```
override def greet() = "I'd like to eat my own dog food"
}

trait ExclamatoryGreeter extends Friendly {
    override def greet() = super.greet() + "!"
}

var pet: Friendly = new Dog
println(pet.greet())

pet = new HungryCat
println(pet.greet())

pet = new HungryDog
println(pet.greet())

pet = new Dog with ExclamatoryGreeter
println(pet.greet())

pet = new HungryCat with ExclamatoryGreeter
println(pet.greet())

pet = new HungryDog with ExclamatoryGreeter
println(pet.greet())
```

When you run the `friendly.scala` script, it will print:

```
Woof
Meow
I'd like to eat my own dog food
Woof!
Meow!
I'd like to eat my own dog food!
```

Conclusion

With the knowledge you've gained in this chapter, you should already be able to get started using Scala for small tasks, especially scripts. In future chapters, we will dive into more detail in these topics, and introduce other topics that weren't even hinted at here.

Chapter 4

Classes and Objects

As computer hardware has become cheaper and more capable, the software that people want to run on it has become larger and more complex. Managing this complexity is one of the main challenges of modern programming. Two of the most important tools provided by Scala to help programmers deal with complexity are classes and objects.

When you are faced with writing a program with complex requirements, you divide those requirements into smaller, simpler pieces. You decompose your program into classes, each of which encompasses an amount of complexity that you can handle. In other words, a class should be responsible for an *understandable* amount of functionality. In the process of designing your classes, you also design interfaces to those classes that abstract away the details of their implementation. Finally, you instantiate those classes into objects and orchestrate the objects in ways that will solve the larger problem. The way classes and objects help you manage complexity at this stage is that as you orchestrate the objects, you can think primarily in terms of their abstract interfaces, and for the most part forget about the more complex details of their implementations.

In this chapter you will learn the basics of classes and objects in Scala. If you are familiar with Java, you'll find the concepts in Scala are similar, but not exactly the same. So even if you're a Java guru, it will pay to read on.

4.1 Objects and variables

When writing the code of a Scala program, you create and interact with *objects*. To interact with a particular object, you can use a *variable* that refers to the object. You can define a variable with either a `val` or `var`, and assign an object to it with `=`. For example, when you write:

```
val i = 1
```

You create an `Int` object with the value `1` and assign it to a variable (in this case, a `val`) named `i`. Similarly, when you write:

```
var s = "Happy"
```

You create a `String` object with the value `"Happy"` and assign it to a variable (in this case, a `var`) named `s`.

You must assign an object to a variable when you define one, a process called *initialization*.¹ You might say, for example, that in the previous two lines of code you initialized variable `i` with the `Int` `1` and variable `s` with the `String` `"Happy"`. Once an object has been assigned to a variable, you can say that the variable *refers* to the object. For example, after the assignments of the previous two lines of code, you could say that the variable named `i` refers to an `Int` with the value `1`, and the variable named `s` refers to a `String` with the value `"Happy"`.

As mentioned in the previous chapters, the difference between the two kinds of variable, `val` and `var`, is that a `val` will always refer to the object with which it is initialized, whereas a `var` can be made to refer to different objects over time. For example, even though the `var` named `s` from the previous example was initialized with the `String` `"Happy"`, you could later make `s` refer to a different `String`:

```
s = "Programming"
```

¹In Java, a variable that is a field of an object may be left unassigned, in which case it is initialized with a predefined default value. The default value depends on the type of the variable: it is `0` or `0.0` for numbers, `false` for booleans, and `null` for reference types. In Scala, you always need an explicit initializer. However, you can achieve the same default initialization effect for fields (but not local variables) by “initializing” with an underscore `_`, as in `var x: Int = _`.

The var `s` initially referred to "Happy", but now refers to "Programming". Were you to attempt such a reassignment on a `val`, you would get a compiler error.

To interact with an object, you must go through a variable that refers to that object. For example, to find the length of the String "Programming", to which variable `s` currently refers, you can invoke the `length` method² on `s`:

```
s.length
```

One way to think about variables is that you use `val` and `var` to attach name tags to objects. Each `val` or `var` defines one name tag. You can give an object a name tag at the beginning of an object's life, and you can later give it still more name tags. For example:

```
var tomato = 7
```

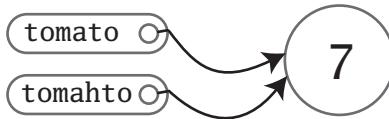
In this example, you create an `Int` object with the value 7, and give it the name tag `tomato`. Thus, you could picture it this way:



Here's how you can give it a second name tag:

```
var tomahto = tomato
```

In this line of code, you defined another variable, a `var` named `tomahto`, and initialized it with the object to which `tomato` refers (the `Int` with the value 7). Both `tomato` and `tomahto` now refer to the same object, thus you can think of this as a 7 with two name tags.



²Functions that are members of classes are called *methods* in Scala, as in Java.

The `tomahto` name could have been a `val`, and you'd still have two name tags for the 7 at this point. Given that `tomahto` is indeed a `var` not a `val`, however, you could later assign a different object to it like this:

```
tomahto = 8
```

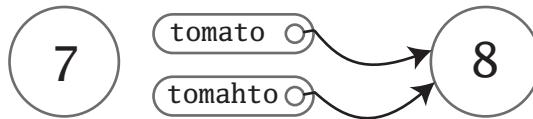
Now you have a 7 with the name tag `tomato` and an 8 with the name tag `tomahto`.



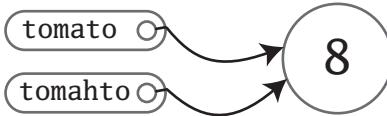
If you want to really shake things up, though, try this:

```
tomato = tomahto
```

Now the 8 has two names, as you'd expect, but what about poor 7?



Well, the old saying “nothing lasts forever” holds for objects in Scala programs too. At this point in your program, that 7 has become *unreachable*. No variable refers to it—it has no name tags—so there is no way your program can interact with it anymore. Therefore any memory it consumes has become available to be automatically reclaimed by the runtime. Thus, for all practical purposes, the picture in memory now looks like this:



4.2 Mapping to Java

If you're familiar with Java, you may be wondering how all this is implemented by the Scala compiler. The remainder of this section will be a quick

tour of how Scala's objects and variables map to Java's. If you're not familiar with Java, or simply aren't interested in peaking behind the curtain, you can safely skip to the beginning of the next section.

In [Chapter 1](#), we claimed that Scala enables programmers to work at a higher level. One of the ways Scala does that is by abstracting away some of the details of the underlying host platform. Just as the Java Platform itself provides an abstraction layer on top of the underlying operating system, Scala provides an abstraction layer on top of the Java Platform. Talking about how Scala programs are implemented can be tricky, because some words, such as “object,” have a different meaning depending upon whether you are talking at the Scala or the Java level of abstraction.

For example, in Java all objects are allocated in an area of the JVM's memory called the heap. If you create a `String` object in Java, it will reside on the heap, even if the `String` is only needed locally within a method. If you declare a local variable of type `String`, the memory used by the variable will reside on the stack, but any `String` to which it refers will reside on the heap. The variable itself will hold a *reference* to the `String`. The reference is conceptually a pointer to, or memory address of, the memory (on the heap) used by the referenced `String` object.

By contrast, in Java an `int` value is *not* an object. The `int` type is primitive in Java. Variables of type `int` contain the integer value itself, not a reference to an object on the heap. Because sometimes an actual integer object is needed, however, Java also provides an immutable integer wrapper type, `java.lang.Integer`. Class `java.lang.Integer` models the exact same concept as `int`, a 32-bit integer, but instances of `java.lang.Integer` are actual objects that reside on the heap. The main purpose of Java's bifurcation of the integer concept into separate primitive and wrapper types is to make possible certain kinds of performance optimization. The tradeoff is that Java programmers must understand and work with two different incarnations of the same integer concept, one of which, the primitive `int`, is not an object. The same bifurcation exists for all of Java's primitive types, `long`, `double`, `boolean`, etc.

To make working with these bifurcated concepts less cumbersome, Java 5 introduced *autoboxing*. Prior to Java 5, if you needed a `java.lang.Integer` but had a primitive `int`, you had to wrap the primitive `int` yourself, a process called *boxing*. Similarly, if you had a `java.lang.Integer` instance, but needed an `int`, you had to invoke

`intValue` on it, a process called *unboxing*.

In Scala, every value is an object, including `Ints` like 1 or 2—but only if you’re using the word *object* at the Scala level of abstraction. Nevertheless, for any Java objects other than the wrapper types, there exists a simple one to one mapping from Scala object to Java object. Here’s an example:

```
var s = "a Java object"
```

Given this source code, the Scala compiler will generate Java bytecodes that instantiate a `java.lang.String` object on the JVM’s heap and store a reference to that new object in a variable. Thus, the Scala `String` object maps to a Java `String` object, and the Scala variable `s` maps to a Java variable that holds a reference to the `String` object. The story is different for Scala objects that map to Java primitive types, however. For example:

```
val i = 1
```

Given this source code, the Scala compiler will essentially generate Java bytecodes that implement the Scala `Int` object as a primitive Java `int` or a `java.lang.Integer` instance, whichever is most advantageous. Throughout its lifetime, a Scala `Int` object’s representation may be boxed (to `java.lang.Integer`) and unboxed (to `int`) repeatedly. Thus a Scala `Int` object may at times map to a Java `java.lang.Integer` object, but at other times to a Java primitive `int` value.

To avoid confusion, in this book we will use the word “reference” only when a reference is certain to exist at runtime at the Java level of abstraction. For example, we might say that the `var s` holds a reference to a `String` with the value “a Java object”. But we wouldn’t say that the `val i` holds a reference to an `Int` with the value 1, because at runtime there may sometimes be a reference to a `java.lang.Integer`, but at other times just a primitive `int`. Instead, we’ll say that `i` *refers* to the `Int` with the value 1. Thus the word “refers” at the Scala level of abstraction does not necessarily imply a reference exists at runtime at the Java level of abstraction.

Similarly, we will not use the word “unreferenced” except to refer to objects implemented at runtime by Java objects that are actually becoming unreferenced and therefore available for garbage collection. Instead, we’ll use the more abstract term “unreachable.” Any Scala object can become unreachable at the Scala level of abstraction. A Scala `Int` implemented as

a primitive Java `int` in a local variable on the stack becomes unreachable when the method returns and the stack frame is popped, but that `Int` does not become unreferenced, or garbage collected, because no reference or Java object is involved.

4.3 Classes and types

The primary tool Scala gives you to organize your programs is the *class*. A class is a blueprint for objects. Once you define a class, you can create objects from the class blueprint with the keyword `new`. For example, given the class definition:

```
class House {  
    // class definition goes here  
}
```

You can create `House` objects (also called *instances* of class `House`) with:

```
new House
```

You may wish to assign the new `House` to a variable so you can interact with it later:

```
val h1 = new House
```

And just as a builder could construct many houses from one blueprint, you can construct many objects from one class:

```
val h2 = new House  
val h3 = new House
```

You now have three `House` objects. You have arrived!

When you define a class, you establish a new *type*. You can then define variables of that type. For example, when you defined class `House` previously, you established `House` as a type. You next create three variables: `h1`, `h2`, and `h3`. The Scala compiler gives the type `House` to each of these variables through type inference. Because you initialized `h1` with a new `House` instance, for example, the Scala compiler inferred the type of `h1` to be `House`. As mentioned in Chapter 2, you can also specify the type of a variable explicitly like this:

```
val h4: House = new House
```

The type of a variable determines what kind of objects can be assigned to it. If you attempt to assign a `String` to a variable of type `House`, for example, it won't compile, because a `String` is not a `House`:

```
var h5: House = "a String" // This won't compile
```

4.4 Fields and methods

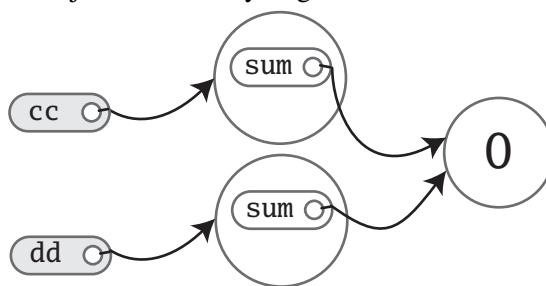
Inside a class definition, you place fields and methods, which are collectively called *members*. Fields are variables that refer to objects. Methods contain executable code. The fields hold the state, or data, of an object, whereas the methods use that data to do the computational work of the object. When you instantiate a class, the runtime sets aside some memory to hold the image of that object's state—*i.e.*, the content of its variables. For example, if you defined a `ChecksumCalculator` class and gave it a `var` field named `sum`:

```
class ChecksumCalculator {  
    var sum = 0  
}
```

and you instantiated it twice with:

```
val cc = new ChecksumCalculator  
val dd = new ChecksumCalculator
```

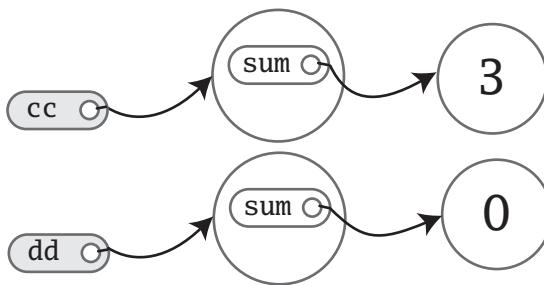
The image of the objects in memory might look like:



Since `sum`, a field declared inside class `ChecksumCalculator`, is a `var`, not a `val`, you can later reassign to `sum` a different `Int` value, like this:

```
cc.sum = 3
```

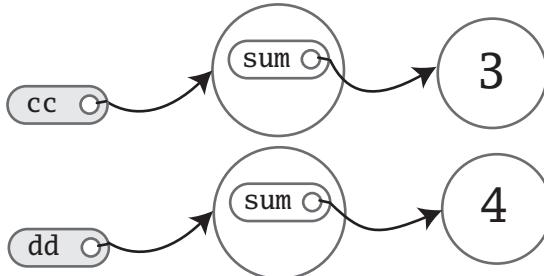
Now the picture would look like:



The first thing to notice about this picture is that there are two `sum` variables, one inside the `ChecksumCalculator` object referred to by `cc` and the other inside the `ChecksumCalculator` object referred to by `dd`. Fields are also known as *instance variables*, because every instance gets its own set of these variables. Collectively, an object's instance variables make up the memory image of the object. You can see this illustrated here not only in that you see two `sum` variables, but also that when you changed one, the other was unaffected. After you executed `cc.sum = 3`, for example, the `sum` inside the `ChecksumCalculator` referred to by `dd` remained at `0`. Similarly, if you reassigned the `sum` instance variable inside the `ChecksumCalculator` object referred to by `dd`:

```
dd.sum = 4
```

The state of the other `ChecksumCalculator` object would be unaffected:



Another thing to note in this example is that you were able to mutate the objects `cc` and `dd` referred to, even though both `cc` and `dd` are `vals`. What

you can't do with `cc` or `dd` given that they are `vals`, not `vars`, is reassign a different object to them. For example, the following attempt would fail:

```
// Won't compile, because cc is a val
cc = new ChecksumCalculator
```

What you can count on, then, is that `cc` will always refer to the same `ChecksumCalculator` object with which you initialize it, but the fields contained inside that object might change over time.

One important way to pursue robustness of an object is to attempt to ensure that the object's state—the values of its instance variables—remains valid during the entire lifetime of the object. The first step is to prevent outsiders from accessing the fields directly by making the fields *private*, so only the methods of the class can access the fields. This way, the code that will be updating that state is localized to just code inside methods defined in the class. To declare a field private, you place a *private* access modifier in front of the field, like this:

```
class ChecksumCalculator {
    private var sum = 0
}
```

Given this definition of `ChecksumCalculator`, any attempt to access `sum` from the outside of the class would fail:

```
val cc2 = new ChecksumCalculator
cc2.sum = 3 // Won't compile, because sum is private
```

Note that the way you make members public in Scala is by not explicitly specifying any access modifier. Put another way, where you'd say “public” in Java, you simply say nothing in Scala. Public is Scala's default access level.

Now that `sum` is private, the only code that can access `sum` is code defined inside the body of the class itself. Thus, `ChecksumCalculator` won't be of much use to anyone unless we define some methods in it:

```
class ChecksumCalculator {
    private var sum = 0
```

```
def add(b: Byte): Unit = {
    sum += b
}

def checksum: Int = {
    return ~(sum & 0xFF) + 1
}

}
```

The ChecksumCalculator now has two methods, one named `add` and the other `checksum`, both of which exhibit the basic form of a function definition, shown in [Figure 2.1 on page 52](#).³ Because `checksum` takes no parameters and returns a conceptual property, we left the parentheses off and made it, therefore, a parameterless method.⁴ In other words, we wrote “`def checksum: Int`,” not “`def checksum(): Int`.”

Any parameters to a method can be used inside the method. One important characteristic of method parameters in Scala is that they are `vals`, not `vars`. The reason parameters are `vals` is that `vals` are simpler to reason about. If you attempt to reassign a parameter inside a method in Scala, therefore, it won’t compile:

```
def add(b: Byte): Unit = {
    b += 1      // This won't compile, because b is a val
    sum += b
}
```

Although the `add` and `checksum` methods in this version of `ChecksumCalculator` correctly implement the desired functionality, in practice you would likely express them using a more concise style. First, the `return` statement at the end of the `checksum` method is superfluous and can be dropped. In the absence of any explicit `return` statement, a Scala method returns the last value computed by the method. The recommended style for methods is in fact to avoid having explicit, and especially multiple, `return` statements. Instead, think of each method as an expression that yields one value, which is returned. This philosophy will encourage you to make methods quite small, to factor larger methods into multiple smaller ones.

³As mentioned in [Chapter 2](#), a function defined as a member of a class is called a *method*.

⁴The reasoning behind parameterless methods was described in [Chapter 2 on page 51](#).

On the other hand, design choices depend on the design context, and Scala makes it easy to write methods in multiple return style if that's what you desire. Because all `checksum` does is calculate a value, it does not need a `return`. Another shorthand for methods is that you can leave off the curly braces if a method computes only a single result expression. If the result expression is short, it can even be placed on the same line as the `def` itself, as shown here:

```
class ChecksumCalculator {  
    private var sum = 0  
    def add(b: Byte): Unit = sum += b  
    def checksum: Int = ~(sum & 0xFF) + 1  
}
```

Methods with a result type of `Unit`, such as `ChecksumCalculator`'s `add` method, are executed for their side effects. A side effect is generally defined as mutating state somewhere external to the method or performing an I/O action. In `add`'s case, for example, the side effect is that `sum` is reassigned. Another way to express such methods is leaving out the result type and the equals sign, following the method with a block enclosed in curly braces. In this form, the method looks like a *procedure*, a method that is executed only for its side effects. An example is the `add` method in the following version of class `ChecksumCalculator`:

```
class ChecksumCalculator {  
    private var sum = 0  
    def add(b: Byte) { sum += b }  
    def checksum: Int = ~(sum & 0xFF) + 1  
}
```

One gotcha to watch out for is that whenever you leave off the equals sign before the body of a method, its result type will definitely be `Unit`. This is true no matter what the body contains, because the Scala compiler can convert any type to `Unit`. Here's an example in which a `String` is converted to `Unit`, because that's the declared result type of the method:

```
scala> def f: Unit = "this String gets lost"
```

```
f: Unit
```

The Scala compiler treats a method defined in the procedure style, *i.e.*, with curly braces but no equals sign, essentially the same as a method that explicitly declares its result type to be `Unit`. Here's an example:

```
scala> def g { "this String gets lost too" }
g: Unit
```

The gotcha occurs, therefore, if you intend to return a non-`Unit` value, but forget the equals sign. To get what you want, you'll need to insert the equals sign:

```
scala> def h = { "this String gets returned!" }
h: java.lang.String
scala> h
res3: java.lang.String = this String gets returned!
```

4.5 Class documentation

To make `ChecksumCalculator` usable as a library component, you should document it. The recommended way of doing this is to use Scaladoc comments. A Scaladoc comment is essentially the same as a Javadoc comment in Java: a multi-line comment that starts with `/**`. The comment always applies to the definition that comes right after it. With the help of the `scaladoc` tool you can produce HTML pages that contain the signatures of classes and their members together with their Scaladoc comments. The Eclipse IDE plugin also displays Scaladoc comments when it shows definitions in its context help feature. Here is a well-commented version of class `ChecksumCalculator`:

```
/** A class that calculates a checksum of bytes. This class
 *  is not thread-safe.
 */
class ChecksumCalculator {
    private var sum = 0
    /** Adds the passed <code>Byte</code> to the checksum
```

```
* calculation.  
*  
* @param b the <code>Byte</code> to add  
*/  
def add(b: Byte) { sum += b }  
  
/** Gets a checksum for all the <code>Byte</code>s passed  
* to <code>add</code>. The sum of the integer  
* returned by this method, when added to the  
* sum of all the passed bytes will yield zero.  
*/  
def checksum: Int = ~(sum & 0xFF) + 1  
}
```

An important point to note is that the purpose of such documentation is not simply to describe what the code is doing, but to abstract away some of the detail. When you design a class for others to use, it is important to design more than just the code. You should also think about the *abstraction*. The abstraction allows client programmers to use your class without thinking about, or even knowing about, all the details of your implementation. If you design the abstraction as carefully as you design the implementation, you can help your client programmers manage complexity, by enabling them to work at a higher level. You can also facilitate future changes, by clearly specifying the contract to which different implementations of your abstraction must adhere.

A tricky part of design can be figuring out just how much of the detail to abstract away. Given `ChecksumCalculator` is a contrived example, with no real design context, it is hard to say what level of abstraction is appropriate. In this example, we left out the details of the algorithm used to calculate the checksum, and simply explained how to use it: add the checksum to the sum of bytes, and make sure the result is zero. We also didn't indicate whether the checksum is calculated when you call `checksum`, or whether the calculation is done with each call to `add` and cached so it can be quickly returned by a call to `checksum`. All that detail is left to the implementation.

Should you always carefully design class documentation like this? No. If you're just whipping up a class to test a concept or use in a throw-away script, and many other cases, the documentation probably isn't worth the effort. Such care in design and documentation really pays off is when teams collaborate by offering each other APIs. Carefully defined and documented

interfaces help such teams work together productively.

For most of this book, we will leave off such Scaladoc documentation, primarily because it will help you see the points we are trying to illustrate. Nevertheless, we believe such documentation is a big help to the productivity of people collaborating on a software project.

4.6 Variable scope

Variable declarations in Scala programs have a *scope* that defines where you can use the name. Outside that scope, any use must be via inheritance, an import, or a field selection such as `cc.sum`.

The most common example is that curly braces generally introduce a new scope, so anything defined inside curly braces leaves scope after the final closing brace.⁵

For an illustration, consider the following script:

```
def printMultiTable() {  
    var i = 1  
    // only i in scope here  
    while (i <= 10) {  
        var j = 1  
        // both i and j in scope here  
        while (j <= 10) {  
            val prod = (i * j).toString  
            // i, j, and prod in scope here  
            var k = prod.length  
            // i, j, prod, and k in scope here  
            while (k < 4) {  
                print(" ")  
                k += 1  
            }  
        }  
    }  
}
```

⁵There are a few exceptions to this rule, because in Scala you can sometimes use curly braces in place of parentheses. One example of this kind of curly-brace use is the alternative for expression syntax described in Section 7.3 on page 156.

```
print(prod)
    j += 1
}
// i and j still in scope; prod and k out of scope
println()
i += 1
}
// i still in scope; j, prod, and k out of scope
}
printMultiTable()
```

The `printMultiTable` function prints out a multiplication table. The first statement of this function introduces a variable named `i` and initializes it to the integer 1. You can then use the name `i` for the remainder of the function.

The next statement in `printMultiTable` is a while loop:

```
while (i <= 10) {
    var j = 1
    ...
}
```

You can use `i` here because it is still in scope. In the first statement inside that while loop, you introduce another variable, this time named `j`, and again initialize it to 1. Because the variable `j` was defined inside the open curly brace of the while loop, it can be used only within that while loop. If you were to attempt to do something with `j` after the closing curly brace of this while loop, after where the comment says that `j`, `prod`, and `k` are out of scope, your program would not compile.

All of the variables defined in this example—`i`, `j`, `prod`, and `k`—are *local variables*. Local variables are “local” to the function in which they are defined. Each time a function is invoked, a new set of its local variables is created.

Once a variable is defined, you can’t define a new variable with the same name in the same scope. For example, the following script would not compile:

```
val a = 1
val a = 2 // Does not compile
println(a)
```

You can, on the other hand, define a variable in an inner scope that has the same name as a variable in an outer scope. The following script would compile and run:

```
val a = 1;
{
    val a = 2 // Compiles just fine
    a
}
println(a)
```

When executed, the script shown previously would print 1, because the a defined inside the curly braces is a different variable, which is in scope only until the closing curly brace.⁶ One difference to note between Scala and Java is that unlike Scala, Java will not let you create a variable in an inner scope that has the same name as a variable in an outer scope. In a Scala program, an inner variable is said to *shadow* a like-named outer variable, because the outer variable becomes invisible in the inner scope.

The main reason Scala differs from Java in this regard is that shadowing enables a more forgiving interpreter environment. Consider the following:

```
scala> val a = 1
a: Int = 1
scala> val a = 2
a: Int = 2
scala> println(a)
2
```

In the interpreter, unlike in a regular Scala program, you can reuse variable names to your heart's content. Among other things, this allows you to change your mind if you made a mistake when you define a variable the first

⁶By the way, the semicolon is required in this case after the first definition of a because Scala's semicolon inference mechanism will not place one there.

time in the interpreter. The reason you can do this is that implicitly, the interpreter creates a new nested scope for each new statement you type in. Thus, you could visualize the previous interpreted code like this:

```
val a = 1;
{
    val a = 2;
    {
        println(a)
    }
}
```

This code will compile and run as a Scala script, and like the code typed into the interpreter, will print 2. Keep in mind that such code can be very confusing to readers, because variable names adopt new meanings in nested scopes. In general, you should avoid shadowing variables explicitly in code you write outside of the interpreter.

Lastly, although the `printMultiTable` shown previously does a fine job of both printing a multiplication table and demonstrating the concept of scope, if you listen carefully you'll hear its many vars and indexes just beginning to be refactored into a more concise, less error-prone, more functional style. Here's one way you could do it:

```
def printMultiTable() {
    for (i <- 1 to 10) {
        for (j <- 1 to 10) {
            val prod = (i * j).toString
            print(String.format("%4s", Array(prod)))
        }
        println()
    }
}
printMultiTable()
```

This version of `printMultiTable` is written in a more functional style than the previous one, because this version avoids using any vars. All the variables—`i`, `j`, and `prod`—are `vals`. One nuance worth pointing out here is that even though `i` is a `val`, each time through the outermost `for` expression's

“loop”, `i` gets a different value. The first time `i` is 1, then 2, and so on all the way to 10. Although this may seem like behavior unbecoming a `val`, one way you can think of it is that for each iteration a brand new `val` named `i` is created and placed into scope inside the body of the `for` expression, where `i` remains true to its `val` nature. If you attempt to assign `i` a new value inside the body of the `for` expression, such as with the statement `i = -1`, your program will not compile.

Now, although this most recent version of `printMultiTable` is in more a functional style than the previous version, it still retains some of its imperative accent. That’s because invoking `printMultiTable` has the side effect of printing to the standard output. Because this is the only reason you invoke `printMultiTable`, its result type is `Unit`. An even more functional version would return the multiplication table as a `String`.

4.7 Semicolon inference

In a Scala program, a semicolon at the end of a statement is usually optional. You can type one if you want but you don’t have to if the statement appears by itself on a single line. Thus, code like the following does not need any semicolons:

```
val prod = 15
var k = prod.toString.length
while (k < 4) {
    print(" ")
    k += 1
}
print(prod)
```

A semicolon is only required if you write several statements on a single line:

```
val prod = 15; var k = prod.toString.length
while (k < 4) { print(" "); k += 1 }
print(prod)
```

If you want to enter a statement that spans multiple lines, most of the time you can simply enter it and Scala will separate the statements in the correct place. For example, the following is treated as one four-line statement:

```
if (x < 2)
    println("too small")
else
    println("ok")
```

Occasionally, however, Scala will split a statement into two parts against your wishes:

```
x
+ y
```

This parses as two statements `x` and `+y`. If you intend to parse it as one statement `x + y`, you can always wrap it in parentheses:

```
(x
+ y)
```

Alternatively, you can put the `+` at the end of a line. For just this reason, whenever you are chaining an infix operation such as `+`, it is a common Scala style to put the operators at the end of the line instead of the beginning:

```
x +
y +
z
```

The precise rules for statement separation are surprisingly simple for how well they work. In short, a line ending is treated as semicolon unless one of the following conditions is true.

1. The line in question ends in a word that would not be legal as the end of a statement, such as a period or an infix-operator.
2. The next line begins with a word that cannot start a statement.
3. The line ends while inside parentheses `(...)` or brackets `[...]`, because these cannot contain multiple statements anyway.

4.8 Singleton objects

As mentioned in [Chapter 1](#), one way in which Scala is more object-oriented than Java is that classes in Scala cannot have static members. Instead, Scala has [singleton objects](#). A singleton object definition looks like a class definition, except instead of the keyword `class` you use the keyword `object`. Here's an example:

```
// In file ChecksumCalculator.scala
object ChecksumCalculator {
    def calcChecksum(s: String): Int = {
        val cc = new ChecksumCalculator
        for (c <- s)
            cc.add(c.toByte)
        cc.checksum
    }
}
```

This singleton object is named `ChecksumCalculator`, which is the same name as the class in the previous example. When a singleton object shares the same name with a class, it is called that class's [companion object](#). You must define both the class and its companion object in the same source file. The class is called the [companion class](#) of the singleton object. A class and its companion object can access each other's private members.

The `ChecksumCalculator` singleton object has one method, `calcChecksum`, which takes a `String` and calculates a checksum for the characters in the `String`. The first line of the method defines a `val` named `cc` and initializes it with a new `ChecksumCalculator` instance. Because the keyword `new` is only used to instantiate classes, the new object created here is an instance of the `ChecksumCalculator` class. The next line is a `for` expression, which cycles through each character in the passed `String`, converts the character to a `Byte` by invoking `toByte` on it, and passing that to the `add` method of the `ChecksumCalculator` instances to which `cc` refers. After the `for` expression completes, the last line of the method invokes `checksum` on `cc`, which gets the checksum for the passed `String`. Because this is the last expression of the method, this checksum is the result returned from the method.

If you are a Java programmer, one way to think of singleton objects is as the home for any static methods you might have written in Java. You can invoke methods on singleton objects using a similar syntax: the name of the singleton object, a dot, and the name of the method. For example, you can invoke the `calcChecksum` method of singleton object `ChecksumCalculator` like this:

```
ChecksumCalculator.calcChecksum("Every value is an object.")
```

One difference between classes and singleton objects is that singleton objects cannot take parameters, whereas classes can via primary and auxiliary constructors. You never have to instantiate a singleton object with the `new` keyword. Singleton objects are implemented as static values, so they have the same initialization semantics as Java statics. In particular, a singleton object is initialized the first time someone accesses it.

A singleton object that does not share the same name with a companion class is called a *standalone object*. You can use standalone objects for many purposes, including collecting related utility methods together, or defining an entry point to a Scala application. This use case is shown in the next section.

4.9 A Scala application

To run a Scala program, you must supply the name of a standalone singleton object with a `main` method that takes one parameter, an `Array[String]`, and has a result type of `Unit`. Any singleton object with a `main` method of the proper signature can be used as the entry point into an application. Here's an example:

```
// In file Summer.scala
import ChecksumCalculator.calcChecksum

object Summer {
    def main(args: Array[String]) {
        for (arg <- args)
            println(arg + ": " + calcChecksum(arg))
    }
}
```

The name of this singleton object is `Summer`. Its `main` method has the proper signature, so you can use it as an application. The first statement in the file is an import of the `calcChecksum` method defined in the `ChecksumCalculator` object in the previous example. This import statement allows you to use the method's simple name in the rest of the file, like a static import feature introduced in Java 5. The body of the `main` method simply prints out each argument and the checksum for the argument, separated by a colon.

To run this application, place the code for object `Summer` in a file name `Summer.scala`. Because `Summer` uses `ChecksumCalculator`, place the code shown next in a file named `ChecksumCalculator.scala`:

```
// In file ChecksumCalculator.scala

class ChecksumCalculator {
    private var sum = 0
    def add(b: Byte) { sum += b }
    def checksum: Int = ~(sum & 0xFF) + 1
}

object ChecksumCalculator {
    def calcChecksum(s: String): Int = {
        val cc = new ChecksumCalculator
        for (c <- s)
            cc.add(c.toByte)
        cc.checksum
    }
}
```

As described in Step 11 on page 79, to compile these files you can use `scalac`, like this:

```
scalac ChecksumCalculator.scala Summer.scala
```

This will create binary Java `.class` files, which you can execute with:

```
scala Summer of love
```

You should see checksums printed for the two command line arguments:

```
of: -213  
love: -182
```

4.10 Conclusion

This chapter has given you the basics of classes and objects in Scala. In the next chapter, you'll learn about Scala's basic types and how to use them.

Chapter 5

Basic Types and Operations

Now that you've taken a tour of Scala and seen how basic classes and objects work, a good place to start understanding Scala in more depth is by looking at its basic types and operations. If you're familiar with Java, you'll be glad to find that Java's basic types and operators have the same meaning in Scala. However there are some interesting differences that will make this chapter worthwhile reading even if you're an experienced Java developer.

As object-oriented languages go, Scala is quite “pure” in the sense that every value in a Scala program is an object, and every operation on an object is a method call. This characteristic holds true even for the basic types such as integers and floating point numbers, and operations such as addition and multiplication. As mentioned in earlier chapters, this object-oriented purity gives rise to a conceptual simplicity that makes it easier to learn Scala and understand Scala programs. However, unlike attempts at purity in some other object-oriented languages,¹ it does not come with a significant performance cost, because the Scala compiler takes advantage of the efficiency of Java's primitive types and their operations when it compiles your Scala program down to Java bytecodes.

Given that all operations on objects in Scala are method calls, Scala doesn't have operators in the same sense as in most languages. Instead, each of the value types (such as `Int`, `Boolean`, and `Double`, *etc.*) have methods with names that act as operators in many other languages. For example, in Java, `*` is an operator that you can use to multiply two `ints`. In Scala, by

¹We don't want to name names here, but if you catch up with one of us at a conference, we may slip and reveal a name over small talk.

contrast, class `Int` has a method named `*`, which does multiplication. As mentioned in the previous chapter, when you say `2 * 3` in Scala, the compiler transforms your expression into `(2).*(3)`. It calls the method named `*` on the `Int` instance with the value `2`, passing in another `Int` instance with the value `3`.

In this chapter, you'll get an overview of Scala's basic types, including `Strings` and the value types `Int`, `Long`, `Short`, `Byte`, `Float`, `Double`, `Char`, and `Boolean`. You'll learn the operations you can perform on these types, including how operator precedence works in Scala expressions. And if that's not enough excitement for you, at the end of the chapter you will learn how implicit conversions to *rich* variants of these basic types can grant you access to more operations than those defined in the classes of these types.

5.1 Some basic types

Several fundamental types of Scala, along with the ranges of values instances may have, are shown in [Table 5.1](#).

Table 5.1: Some Basic Types

Value Type	Range
<code>Byte</code>	8-bit signed two's complement integer (- 2^7 to $2^7 - 1$, inclusive)
<code>Short</code>	16-bit signed two's complement integer (- 2^{15} to $2^{15} - 1$, inclusive)
<code>Int</code>	32-bit signed two's complement integer (- 2^{31} to $2^{31} - 1$, inclusive)
<code>Long</code>	64-bit signed two's complement integer (- 2^{63} to $2^{63} - 1$, inclusive)
<code>Char</code>	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
<code>String</code>	a sequence of <code>Chars</code>
<code>Float</code>	32-bit IEEE 754 single-precision float
<code>Double</code>	64-bit IEEE 754 double-precision float
<code>Boolean</code>	true or false

Other than `String`, which resides in package `java.lang`, all of these basic types are members of package `scala`.² For example, the full name of `Int` is `scala.Int`. However, given that all the members of package `scala` and `java.lang` are automatically imported into every Scala source file, you

²Packages will be covered in depth in [Chapter 13](#).

can just use the simple names (*i.e.*, names like Boolean, or Char, or String) everywhere.³

Savvy Java developers will note that these are the same exact ranges of the corresponding types in Java. This enables the Scala compiler to transform instances of Scala *value types*, such as Int or Double, down to Java primitive types in the bytecodes it produces.

5.2 Literals

All of the basic types listed in Table 5.1 can be written with *literals*. A literal is a way to write a constant value directly in code. The syntax of these literals is exactly the same as in Java, so if you’re a Java master, you may wish to skim most of this section. The one difference to note is Scala’s multi-line String literal, which is described on [page 120](#).

Integer literals

Integer literals for the types Int, Long, Short, and Byte come in three forms: decimal, hexadecimal, and octal. The way an integer literal begins indicates the base of the number. If the number begins with a 0x or 0X, it is hexadecimal (base 16), and may contain upper or lowercase digits A through F as well as 0 through 9. Some examples are:

```
scala> val hex = 0x5
hex: Int = 5

scala> val hex2 = 0x0OFF
hex2: Int = 255

scala> val magic = 0xcafebabe
magic: Int = -889275714
```

³You can in fact currently use lower case aliases for Scala value types, which correspond to Java’s primitive types. For example, you can say int instead of Int in a Scala program. But keep in mind they both mean exactly the same thing: `scala.Int`. The recommended style that arose from the experience of the Scala community is to always use the upper case form, which is what we attempt to do consistently in this book. In honor of this community-driven choice, the lower case variants may be deprecated or even removed in a future version of Scala, so you would be wise indeed to go with the community flow and say Int, not int, in your Scala code.

Note that the Scala shell always prints integer values in base 10, no matter what literal form you may have used to initialize it. Thus the interpreter displays the value of the hex2 variable you initialized with literal 0x00FF as decimal 255. (Of course, you don't need to take our word for it. A good way to start getting a feel for the language is to try these statements out in the interpreter as you read this chapter.) If the number begins with a zero, it is octal (base 8), and may only contain digits 0 through 7. Some examples are:

```
scala> val oct = 035 // (35 octal is 29 decimal)
oct: Int = 29

scala> val nov = 0777
nov: Int = 511

scala> val dec = 0321
dec: Int = 209
```

If the number begins with a non-zero digit, it is decimal (base 10). For example:

```
scala> val dec1 = 31
dec1: Int = 31

scala> val dec2 = 255
dec2: Int = 255

scala> val dec3 = 20
dec3: Int = 20
```

If an integer literal ends in an L or l, it is a Long, otherwise it is an Int. Some examples of Long integer literals are:

```
scala> val prog = 0XCAFEBABEL
prog: Long = 3405691582

scala> val tower = 35L
tower: Long = 35

scala> val of = 31l
of: Long = 31
```

If an Int literal is assigned to a variable of type Short or Byte, the literal is treated as if it were a Short or Byte type so long as the literal value is within the valid range for that type. For example:

```
scala> val little: Short = 367
little: Short = 367
scala> val littler: Byte = 38
littler: Byte = 38
```

Floating point literals

Floating point literals are made up of decimal digits, optionally containing a decimal point, and optionally followed by an E or e and an exponent. Some examples of floating point literals are:

```
scala> val big = 1.2345
big: Double = 1.2345
scala> val bigger = 1.2345e1
bigger: Double = 12.345
scala> val biggerStill = 123E45
biggerStill: Double = 1.23E47
```

Note that the exponent portion means the power of 10 by which the other portion is multiplied. Thus, 1.2345e1 is 1.2345 *times* 10^1 , which is 12.345. If a floating point literal ends in a F or f, it is a `Float`, otherwise it is a `Double`. Optionally, a `Double` floating point literal can end in D or d. Some examples of `Float` literals are:

```
scala> val little = 1.2345F
little: Float = 1.2345
scala> val littleBigger = 3e5f
littleBigger: Float = 300000.0
```

That last value expressed as a `Double` could take these (as well as other) forms:

```
scala> val anotherDouble = 3e5
anotherDouble: Double = 300000.0

scala> val yetAnother = 3e5D
yetAnother: Double = 300000.0
```

Character literals

Character literals can be any Unicode character between single quotes, such as:

```
scala> val a = 'A'
a: Char = A
```

In addition to providing an explicit character between the single quotes, you can provide an octal or hex number for the character code point preceded by a backslash. The octal number must be between '\0' and '\377'. For example, the Unicode character code point for the letter A is 101 octal. Thus:

```
scala> val c = '\101'
c: Char = A
```

A character literal in hex form must have four digits and be preceded by a \u, as in:

```
scala> val d = '\u0041'
d: Char = A

scala> val f = '\u005a'
f: Char = Z
```

There are also a few character literals represented by special escape sequences, shown in [Table 5.2](#).

For example:

```
scala> val backslash = '\\'
backslash: Char = \
```

Table 5.2: Special Character Literal Escape Sequences

Literal	Meaning
\n	line feed (\u000A)
\b	backspace (\u0008)
\t	tab (\u0009)
\f	form feed (\u000C)
\r	carriage return (\u000D)
\"	double quote (\u0022)
'	single quote (\u0027)
\\	backslash (\u005C)

String literals

The usual notation for a string literal is to surround a number of characters by double quotes (""):

```
scala> val hello = "hello"
hello: java.lang.String = hello
```

The syntax of the characters within the quotes is the same as with character literals. For example:

```
scala> val escapes = "\\\\"\\"
escapes: java.lang.String = \\"
```

This syntax is awkward for multi-line strings that contain a lot of escape sequences. For those, Scala includes a special syntax for multi-line strings. You start and end a multi-line string with three quotation marks in a row (""""). The interior of a multi-line string may contain any characters whatsoever, including newlines, quotation marks, and special characters, except of course three quotes in a row. For example, the following program prints out a message using a multi-line string:

```
println("""Welcome to Ultamix 3000.
Type "HELP" for help.""")
```

Running this code does not produce quite what is desired, however:

```
Welcome to Ultamix 3000.  
Type "HELP" for help.
```

The issue is that the leading spaces before the second line are included in the string! To help with this common situation, the String class includes a method call `stripMargin`. To use this method, put a pipe character (|) at the front of each line, and then call `stripMargin` on the whole string:

```
println("")|Welcome to Ultamix 3000.  
|Type "HELP" for help.""".stripMargin)
```

Now the code behaves as desired:

```
Welcome to Ultamix 3000.  
Type "HELP" for help.
```

Boolean literals

The Boolean type has two literals, `true` and `false`, which can be used like this:

```
scala> val bool = true  
bool: Boolean = true  
  
scala> val fool = false  
fool: Boolean = false
```

That's all there is to it. You are now literally⁴ an expert in Scala.

5.3 Operators are methods

Like most programming languages, Scala facilitates basic operations on its basic types, such as adding and subtracting numeric values and and-ing and or-ing boolean values. If you're familiar with Java, you'll find that the semantics of such expressions in Scala look the same as corresponding expressions in Java, even though they are arrived at in Scala in a slightly more object-oriented way.

⁴figuratively speaking

Scala provides a rich set of operators for its basic types. As mentioned in previous chapters, these operators are actually just a nice syntax for ordinary method calls. For example, `1 + 2` really means the same thing as `(1).+(2)`.⁵ In other words, class `Int` contains a method named `+` that takes an `Int` and returns an `Int` result. This `+` method is conceptually invoked when you add two `Ints`,⁶ as in:

```
scala> val sum = 1 + 2      // Scala invokes (1).+(2),  
sum: Int = 3                // which returns an Int
```

To prove this to yourself, write the expression explicitly as a method invocation:

```
scala> val sumMore = (1).+(2)  
sumMore: Int = 3
```

The whole truth, however, is that `Int` contains several “overloaded” `+` methods that take different parameter types.⁷ For example, `Int` has a different method also named `+` that takes a `Long` and returns a `Long`. If you add a `Long` to an `Int`, this alternate `+` method will be invoked, as in:

```
scala> val longSum = 1 + 2L     // Scala invokes (1).+(2L),  
longSum: Long = 3              // an overloaded + method on Int  
                                // that returns a Long
```

The upshot of all this is that all methods in Scala can be used in operator notation. In Java, for example, operators are a special language syntax. In Scala, an operator is a method—any method—invoked without the dot using one of three operator notations: prefix, postfix, or infix. In prefix notation, you put the method name before the object on which you are invoking the method, for example, the `“-”` in `“-7”`. In postfix notation, you put the method after the object, for example, the `toLong` in `“7 toLong”`. And in infix notation, you put the method between the object and the parameter or parameters you wish to pass to the method, for example, the `+` in `“7 + 2”`.

⁵By the way, the spaces around operators shown in this book usually just for style. `1+2` would compile just as fine as `1 + 2`.

⁶Conceptually invoked, because the Scala compiler will generally optimize this down to native Java bytecode addition on primitive Java `ints`.

⁷*Overloaded* methods have the same name but different argument types. More on method overloading in [Chapter 6](#).

So in Scala, `+` is not an operator. It's a method. But when you say `1 + 2`, you are *using* `+` as an operator—an infix operator to be specific. Moreover, this notation is not limited to things like `+` that look like operators in other languages. You can use any method in operator notation. For example, class `java.lang.String` has a method `indexOf` that takes one `Char` parameter. The `indexOf` method searches the `String` for the first occurrence of the specified character, and returns its index or `-1` if it doesn't find the character. You can use `indexOf` as an infix operator, like this:

```
scala> val s = "Hello, world!"  
s: java.lang.String = Hello, world!  
  
scala> s indexOf 'o'          // Scala invokes s.indexOf('o')  
res0: Int = 4
```

In addition, `java.lang.String` offers an overloaded `indexOf` method that takes two parameters, the character for which to search and an index at which to start. (The other `indexOf` method, shown previously, starts at index zero, the beginning of the `String`.) Even though this `indexOf` method takes two arguments, you can use it as an operator in infix notation. But whenever you call a method that takes multiple arguments using infix notation, you have to place those arguments in parentheses. For example, here is how you use this other form of `indexOf` as an operator (continuing from the previous example):

```
scala> s indexOf ('o', 5) // Scala invokes s.indexOf('o', 5)  
res1: Int = 8
```

Thus, in Scala operators are not special language syntax: any method can be an operator. What makes a method an operator is how you *use* it. When you say `s.indexOf('o')`, `indexOf` is not an operator. But when you say, `s indexOf 'o'`, `indexOf` is an operator.

In contrast to the infix operator notation—in which operators take two operands, one to the left and the other to the right—prefix and postfix operators are *unary*—they take just one operand. In prefix notation, the operand is to the right of the operator. Some examples of prefix operators are `-2.0`, `!found`, and `~0xFF`. As with the infix operators, these prefix operators are a shorthand way of invoking methods on value type objects. In this case,

however, the name of the method has “`unary_-`” prepended to the operator character. For instance, Scala will transform the expression `-2.0` into the method invocation “`(2.0).unary_-`”. You can demonstrate this to yourself by typing the method call both via operator notation and explicitly:

```
scala> -2.0                                // Scala invokes (2.0).unary_-
res2: Double = -2.0

scala> (2.0).unary_-
res3: Double = -2.0
```

The only identifiers that can be used as prefix operators are `+`, `-`, `!`, and `~`. Thus, if you define a method named `unary_!`, you could invoke that method on a value or variable of the appropriate type using prefix operator notation, such as `!p`. But if you define a method named `unary_*`, you wouldn’t be able to use prefix operator notation, because `*` isn’t one of the four identifiers that can be used as prefix operators. You could invoke the method normally, as in `p.unary_*`, but if you attempted to invoke it via `*p`, Scala will parse it as `*.p`, which is probably not what you had in mind!⁸

Postfix operators are methods that take no arguments, invoked without a dot or parentheses. As mentioned in the previous chapter, if a method takes no arguments, the convention is that you include parentheses if the method has side effects, such as `println()`, but leave them off if the method has no side effects, such as `toLowerCase` invoked on a `String`:

```
scala> val s = "Hello, world!"
s: java.lang.String = Hello, world!

scala> s.toLowerCase
res4: java.lang.String = hello, world!
```

In this latter case of a method that requires no arguments, you can alternatively leave off the dot and use postfix operator notation:

```
scala> s.toLowerCase
res5: java.lang.String = hello, world!
```

⁸All is not necessarily lost, however. There is an extremely slight chance your program with the `*p` might compile as C++.

In this case, `toLowerCase` is used as a postfix operator on the operand `s`.

To see what operators you can use with Scala's value types, therefore, all you really need to do is look at the methods declared in the value type's classes in the Scala API documentation. Given that this is a Scala tutorial, however, we'll give you a quick tour of most of these methods in the next few sections. If you're a Java guru in a rush, you can likely skip to [Section 5.8](#) on [page 131](#).

5.4 Arithmetic operations

You can invoke arithmetic methods via infix operator notation for addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and remainder (`%`), on any integer or floating point type. Here are some examples:

```
scala> 1.2 + 2.3
res6: Double = 3.5

scala> 3 - 1
res7: Int = 2

scala> 'b' - 'a'
res8: Int = 1

scala> 2L * 3L
res9: Long = 6

scala> 11 / 4
res10: Int = 2

scala> 11 % 4
res11: Int = 3

scala> 11.0f / 4.0f
res12: Float = 2.75

scala> 11.0 % 4.0
res13: Double = 3.0
```

When both the left and right operands are integer types (`Int`, `Long`, `Byte`, `Short`, or `Char`), the `/` operator will tell you the whole number portion of the quotient, excluding any remainder. The `%` operator indicates the remainder of an implied integer division.

Note that the floating point remainder you get with `%` is not the one defined by the IEEE 754 standard. The IEEE 754 remainder uses rounding division, not truncating division, in calculating the remainder, so it is quite different from the integer remainder operation. If you really want an IEEE 754 remainder, you can call `IEEEremainder` on `scala.Math`, as in:

```
scala> Math.IEEEremainder(11.0, 4.0)
res14: Double = -1.0
```

Numeric types also offer unary prefix `+` and `-` operators (methods `unary_+` and `unary_-`), which allow you to indicate a literal number is positive or negative, as in `-3` or `+4.0`. If you don't specify a unary `+` or `-`, a literal number is interpreted as positive. Unary `+` exists solely for symmetry with unary `-`, but it has no effect. The unary `-` can also be used to negate a variable. Here are some examples:

```
scala> val neg = 1 + -3
neg: Int = -2
scala> val y = +3
y: Int = 3
scala> -neg
res15: Int = 2
```

5.5 Relational and logical operations

You can compare two value types with the relational methods `greater than (>)`, `less than (<)`, `greater than or equal to (>=)`, and `less than or equal to (<=)`, which like the equality operators, yield a Boolean result. In addition, you can use the unary `!` operator (the `unary_!` method) to invert a Boolean value. Here are a few examples:

```
scala> 1 > 2
res16: Boolean = false
scala> 1 < 2
res17: Boolean = true
```

```
scala> 1.0 <= 1.0
res18: Boolean = true

scala> 3.5f >= 3.6f
res19: Boolean = false

scala> 'a' >= 'A'
res20: Boolean = true

scala> val thisIsBoring = !true
thisIsBoring: Boolean = false

scala> !thisIsBoring
res21: Boolean = true
```

The logical methods, logical-and (`&&`), and logical-or (`||`), take Boolean operands in infix notation, and yield a Boolean result. For example:

```
scala> val toBe = true
toBe: Boolean = true

scala> val question = toBe || !toBe
question: Boolean = true

scala> val paradox = toBe && !toBe
paradox: Boolean = false
```

The logical-and and logical-or operations are short-circuited as in Java. Expressions built from these operators are only evaluated as far as needed to determine the result. In other words, the right hand side of logical-and and logical-or expressions won't be evaluated if the left hand side determines the result. For example, if the left hand side of a logical-and expression evaluates to `false`, the result of the expression will definitely be `false`, so the right hand side is not evaluated. Likewise, if the left hand side of a logical-or expression evaluates to `true`, the result of the expression will definitely be `true`, so the right hand side is not evaluated. Here are some examples:

```
scala> def salt() = { println("salt"); false }
salt: ()Boolean

scala> def pepper() = { println("pepper"); true }
pepper: ()Boolean
```

```
scala> salt() && pepper()
salt
res22: Boolean = false

scala> pepper() && salt()
pepper
salt
res23: Boolean = false
```

Notice that in the second case, both `pepper` and `salt` run. In the first comparison, only `salt` runs. Because `salt` returns `false`, there is no need to check what `pepper` returns.

By the way, you may be wondering how short circuiting can work if operators are just methods. A normal method call evaluates all of the arguments before entering the method, so how can a method then choose not to evaluate its second argument? The answer is that all Scala methods have a facility for delaying the evaluation of their arguments, or even declining to evaluate them at all. The facility is called *by-name parameters*, and is discussed in [Chapter 9](#).

5.6 Object equality

If you want to compare two objects to see if they are equal, you should usually use either `==`, or its inverse `!=`. Here are a few simple examples:

```
scala> 1 == 2
res24: Boolean = false

scala> 1 != 2
res25: Boolean = true

scala> 2 == 2
res26: Boolean = true
```

These operations actually apply to all objects, not just basic types. For example, you can use it to compare lists:

```
scala> List(1,2,3) == List(1,2,3)
res27: Boolean = true

scala> List(1,2,3) == List(4,5,6)
```

```
res28: Boolean = false
```

Going further, you can compare two objects that have different types:

```
scala> 1 == 1.0
```

```
res29: Boolean = true
```

```
scala> List(1,2,3) == "hello"
```

```
res30: Boolean = false
```

You can even compare against null, or against things that might be null. No exception will be thrown:

```
scala> List(1,2,3) == null
```

```
res31: Boolean = false
```

```
scala> null == List(1,2,3)
```

```
res32: Boolean = false
```

As you see, == has been carefully crafted so that you get just the equality comparison you want in many cases. This is accomplished with a very simple rule: first check each side for null, and if neither side is null, call the equals method. Since equals is a method, the precise comparison you get depends on the type of the left-hand argument. Since there is an automatic null check, you do not have to do the check yourself.

This kind of comparison will yield true on different objects, so long as their contents are the same and their equals method is written to be based on contents. For example, here is a comparison between two strings that happen to have the same five letters in them:

```
scala> ("he" + "llo") == "hello"
```

```
res33: Boolean = true
```

Note that this is different from Java's == operator, which you can use to compare both primitive and reference types. On reference types, Java's == compares *reference equality*, which means the two variables point to the same object on the JVM's heap. Scala provides this facility, as well, under the name eq. However, eq and its opposite, ne, only apply to objects that directly map to Java objects. The full details about eq and ne are given in Sections 10.15 and 10.16.

5.7 Bitwise operations

Scala enables you to perform operations on individual bits of integer types with several bitwise methods. The bitwise methods are: bitwise-and (`&`), bitwise-or (`|`), and (or bitwise-xor) (`^`).⁹ The unary bitwise complement operator, `~` (the method `unary_~`), inverts each bit in its operand. For example:

```
scala> 1 & 2
res34: Int = 0

scala> 1 | 2
res35: Int = 3

scala> 1 ^ 3
res36: Int = 2

scala> ~1
res37: Int = -2
```

The first expression, `1 & 2`, bitwise-ands each bit in 1 (0001) and 2 (0010), which yields 0 (0000). The second expression, `1 | 2`, bitwise-ors each bit in the same operands, yielding 3 (0011). The third expression, `1 ^ 3`, bitwise-xors each bit in 1 (0001) and 3 (0011), yielding 2 (0010). The final expression, `~1`, inverts each bit in 1 (0001), yielding -2 (11111111111111111111111111111110).

Scala integer types also offer three shift methods: shift left (`<<`), shift right (`>>`), and unsigned shift right (`>>>`). The shift methods, when used in infix operator notation, shift the integer value on the left of the operator by the amount specified by the integer value on the right. Shift left and unsigned shift right fill with zeroes as they shift. Shift right fills with the highest bit (the sign bit) of the left hand value as it shifts. Here are some examples:

```
scala> -1 >> 31
res38: Int = -1

scala> -1 >>> 31
res39: Int = 1
```

⁹The bitwise-xor method performs an *exclusive or* on its operands. Identical bits yield a 0. Different bits yield a 1. Thus $0011 \wedge 0101$ yields 0110

```
scala> 1 << 2
res40: Int = 4
```

In the first example, $-1 \gg 31$, -1 (binary 11111111111111111111111111111111) is shifted to the right 31 bit positions. Since an Int consists of 32 bits, this operation effectively moves the leftmost bit over until it becomes the rightmost bit.¹⁰ Since the \gg method fills with ones as it shifts right, because the leftmost bit of -1 is 1, the result is identical to the original left operand, 32 one bits, or -1 . In the second example, $-1 \ggg 31$, the leftmost bit is again shifted right until it is in the rightmost position, but this time filling with zeros along the way. Thus the result this time is binary 00000000000000000000000000000001, or 1 . In the final example, $1 \ll 2$, the left operand, 1 , is shifted left two position (filling in with zeros), resulting in binary 00000000000000000000000000000000100, or 4 .

5.8 Operator precedence and associativity

Operator precedence determines which parts of an expression are evaluated before the other parts. For example, the expression $2 + 2 * 7$ evaluates to 16, not 28, because the $*$ operator has a higher precedence than the $+$ operator. Thus the $2 * 7$ part of the expression is evaluated before the $2 + 2$ part. You can of course use parentheses in expressions to clarify evaluation order or to override precedence. For example, if you really wanted the result of the expression above to be 28, you could write the expression like this:

```
(2 + 2) * 7
```

Given that Scala doesn't have operators, per se, just a way to use methods in operator notation, you may be wondering how operator precedence works. Scala decides precedence based on the first character of the methods used in operator notation. If the method name starts with a $*$, for example, it will have a higher precedence than a method that starts with a $+$. Thus $2 + 2 * 7$ will be evaluated as $2 + (2 * 7)$, and $a +++ b *** c$ (in which a , b , and

¹⁰The leftmost bit in an integer type is the sign bit. If the leftmost bit is 1, the number is negative. If 0, the number is positive.

c are values or variables, and `+++` and `***` are methods) will be evaluated as `+++(b *** c)`, because the `***` method has a higher precedence than the `+++` method.

Table 5.3 shows the precedence given to the first character of a method in decreasing order of precedence, with characters on the same line having the same precedence. The higher a character is in this table, the higher the precedence of methods that start with that character.

Table 5.3: Operator precedence

(all other special characters)
* / %
+ -
:
= !
< >
&
^
(all letters)

Here's an example:

```
scala> 2 << 2 + 2
res41: Int = 32
```

The `<<` method starts with the character `<`, which appears lower in Table 5.3 than the character `+`, which is the first and only character of the `+` method. Thus `<<` will have lower precedence than `+`, and the expression will be evaluated by first invoking the `+` method, then the `<<` method, as in $2 << (2 + 2)$. $2 + 2$ is 4, by our math, and $2 << 4$ yields 32. Here's another example:

```
scala> 2 + 2 << 2
res42: Int = 16
```

Since the first characters are the same, the methods will be invoked in the same order. First the `+` method will be invoked, then the `<<` method. So `2 + 2` will again yield `4`, and `4 << 2` is `16`.

When multiple operators of the same precedence appear side by side in an expression, the *associativity* of the operators determines the order of evaluation. The associativity of an operator in Scala is determined by its *last* character. As mentioned in the previous chapter in [footnote 2](#) on [page 70](#), any method that ends in a `:` character is invoked on its right operand, passing in the left operand. Methods that end in any other character are the other way around. They are invoked on their left operand, passing in the right operand. So `a * b` yields `a.*(b)`, but `a :: b` yields `b.:::(a)`. This associativity rule also plays a role when multiple operators of the same precedence appear side by side. If the methods end in `:`, they are evaluated right to left; otherwise, they are evaluated left to right. For example, `a :: b :: c` is evaluated `a :: (b :: c)`. But `a * b * c` is evaluated `(a * b) * c`.

Operator precedence is part of the Scala language. You needn't be afraid to use it. But on the other hand, if you find yourself attempting to show off your knowledge of precedence, consider using parentheses to clarify what operators are operating upon what expressions. Perhaps the only precedence you can truly count on other programmers knowing without looking up is that multiplicative operators, `*`, `/`, and `%`, have a higher precedence than the additive ones `+` and `-`. Thus even if `a + b << c` yields the result you want without parentheses, the extra clarity you get by writing `(a + b) << c` may reduce the frequency with which your peers utter your name in operator notation, for example, by shouting in disgust, “`bills !*&^%~ code!`”¹¹

5.9 Rich wrappers

You can invoke many more methods on Scala's basic types than were described in the previous sections. A few examples are shown in [Table 5.4](#). These methods are available via *implicit conversions*, a technique that will be described in detail in [Chapter 19](#). All you need to know for now is that for each basic type described in this chapter, there is also a “rich wrapper” that provides several additional methods. So, to see all the available methods

¹¹By now you should be able to figure out that given this code, the Scala compiler would invoke `(bills.!*&^%~(code)).!()`.

Table 5.4: Some Rich Operations

Code	Result
0 max 5	5
0 min 5	0
-2.7 abs	2.7
-2.7 round	-3L
1.5 isInfinity	false
(1.0 / 0) isInfinity	true
4 to 6	Range(4, 5, 6)
"bob" capitalize	"Bob"
"robert" drop 2	"bert"

Table 5.5: Rich Wrapper Classes

Basic Type	Rich Wrapper
Byte	scala.runtime.RichByte
Short	scala.runtime.RichShort
Int	scala.runtime.RichInt
Char	scala.runtime.RichChar
String	scala.runtime.RichString
Float	scala.runtime.RichFloat
Double	scala.runtime.RichDouble
Boolean	scala.runtime.RichBoolean

on the basic types, you should look at the API documentation on the rich wrapper for each basic type. Those classes are listed in [Table 5.5](#).

5.10 Conclusion

The main take-aways from this chapter are that operators in Scala are method calls, that implicit conversions to rich variants exist for Scala's basic types that add even more useful methods. In the next chapter, we'll show you what

it means to design objects in a functional style.

Chapter 6

Functional Objects

A common thing said about OO is that an object “encapsulates state, behavior and identity”. In [Chapter 4](#) you have seen a class `ChecksumCalculator` where the state was encapsulated in a variable. However, there are many useful classes that do not encapsulate mutable state, because they describe something that is immutable. Cases of immutable objects are already found in Java: Take `java.lang.String` or `java.lang.Integer`, for example. As a functional language, Scala puts great emphasis on such immutable objects. In this chapter, we present as a case study the design of a class for Rational numbers, which are immutable objects. On the way, you’ll learn more aspects of object-oriented programming in Scala: class parameters and constructors, methods and operators, private members, overloading, and self references.

6.1 A class for rational numbers

A simple example is rational numbers. A rational number is a fraction $\frac{x}{y}$ where x and y are whole numbers. x is called the *numerator* or the fraction and y is called the *denominator*. Examples of rational numbers are $\frac{1}{2}$, $\frac{2}{3}$, $\frac{112}{239}$, or $\frac{2}{1}$. Compared to floating point numbers, rational numbers have the advantage that fractions are represented exactly, without any rounding or approximation.

Computation on rational numbers follows the usual laws. For instance, to compute $\frac{1}{2} + \frac{2}{3}$, you first obtain the same denominator by multiplying both parts of the left operand with 3 and both parts of the right operand with 2.

This gives $\frac{3}{6} + \frac{4}{6}$. After that you add the two numerators, which gives $\frac{7}{6}$. Multiplying two rational numbers is done by multiplying their numerators and multiplying their denominators. For instance, $\frac{1}{2} * \frac{2}{5}$ gives $\frac{2}{10}$, which can be represented more compactly as $\frac{1}{5}$. Division is done by swapping numerator and denominator of the right operand and then multiplying. For instance $\frac{1}{2}/\frac{3}{5}$ is the same as $\frac{1}{2} * \frac{5}{3}$ or $\frac{5}{6}$.

One—maybe rather trivial—observation is that rational numbers do not have mutable state. You can add a constant to a rational number, but the result will be a new rational number. The original number will not have “changed”.

We’ll now design a class which implements rational numbers, and the arithmetic operations on them. Since rational numbers are by their nature unchangeable, the class itself will be purely functional. It will not contain any mutable fields.

Here is a first version of class Rational. It will be augmented and refined later on in this chapter.

```
class Rational(n: Int, d: Int) {
    val numer: Int = n
    val denom: Int = d
    def add(that: Rational): Rational =
        new Rational(numer * that.denom + that.numer * denom,
                     denom * that.denom)
    def sub(that: Rational): Rational =
        new Rational(numer * that.denom - that.numer * denom,
                     denom * that.denom)
    def mul(that: Rational): Rational =
        new Rational(numer * that.numer, denom * that.denom)
    def div(that: Rational): Rational =
        new Rational(numer * that.denom, denom * that.numer)
}
```

6.2 Choosing between val and var

Class Rational contains two fields, numer and denom, which are both defined as vals. This means that the fields are immutable; they won't change after their initial assignment. You could have also defined the fields as vars. This would have led to Rational number objects that can change their value over their lifetime. However, from a mathematical standpoint, this makes little sense: A rational number is defined by its value, so if you can change the value, you have a variable containing a rational number, not a rational number itself.

If you're coming from an imperative background, such as Java, C++, or C#, you may think of var as a regular variable and val as a special kind of variable. From the Java perspective, for example, a val is like a *final* variable. On the other hand, if you're coming from a functional background, such as Haskell, OCaml, or Erlang, you might think of val as a regular variable and var as akin to blasphemy. The Scala perspective, however, is that val and var are just two different tools in your toolbox, both useful, neither inherently evil. The ChecksumCalculator of Chapter 4 clearly required a mutable field, whereas the Rational class in this chapter equally clearly should be immutable. Scala encourages you to reach, without guilt, for the best tool for the job at hand. Nevertheless, even if you agree with this balanced philosophy, you might be wondering how to apply val and var most effectively.

If you're coming from an imperative background, you may find yourself using vars everywhere when you start programming in Scala. This is actually a scientifically recognized disease known as *varmonia*. Don't worry. There's a cure, which is simply this: challenge vars in your code, and where possible and appropriate, try and change them into vals. The reason we recommend you prefer vals over vars is that reassignable values are harder to reason about. An immutable object is just itself, whereas an object with mutable fields changes its value over time. You then need to worry whether the changes and observations of an object are all done in the right order. This task is hard enough for sequential programs, but it becomes a huge source of trouble as soon as your program has concurrent threads. By contrast, immutable objects can be easily shared between computations, whether they are sequential or concurrent.

6.3 Class parameters and constructors

Classes in Scala can take parameters. For instance, class Rational takes two parameters n and d which represent the numerator and denominator of the fraction. Assuming you have saved the definition of class Rational above in a file Rational.scala, you can create Rational numbers as follows:

```
scala> :load Rational.scala
Loading Rational.scala...
defined class Rational

scala> new Rational(1,2)
res0: Rational = Rational@11af7bb
```

One difference between Java and Scala concerns constructors. In Scala, classes can take parameters directly, whereas in Java, classes have constructors, which can take parameters. The Scala notation is more concise – class parameters can be used directly in the body of the class; there's no need to define fields and write assignments which copy constructor parameters into fields. This can yield substantial savings in boilerplate code; especially for small classes.

In fact, “under the covers”, a Scala class does have a constructor, even though it is not directly visible to user programs. This constructor is called the *primary constructor* of the class. It takes the class parameters and executes all statements of the class body. You can verify this by adding a print statement right into the body of Rational:

```
class Rational(n: Int, d: Int) {
    println("created: "+n+"/"+d)
    ... // rest of class is as before
}
```

If you re-load the changed class into the interpreter, you will get something like the following:

```
scala> :load Rational.scala
Loading Rational.scala...
defined class Rational

scala> new Rational(1,2)
```

```
created: 1/2
res1: Rational = Rational@29483
```

By the way, the Scala interpreter shell offers a shortcut here: you can “replay” a whole interpreter session using the :replay command. So a shorter way to try out the changed class Rational would be like this:

```
scala> :replay
Replaying: :load Rational.scala
Loading Rational.scala...
defined class Rational

Replaying: new Rational(1,2)
created: 1/2
res1: Rational = Rational@10c81a6
```

6.4 Multiple constructors

Sometimes one wants multiple constructors in a class. Scala supports this as well, through auxiliary constructors. An example of a auxiliary constructor is found in the following version of Rational:

```
class Rational(n: Int, d: Int) {
    def this(n: Int) = this(n, 1)
    println("created: "+n+"/"+d)
    // rest of class is as before
}
```

Secondary constructors in Scala start with `def this(...)`; In the code above, an auxiliary constructor is used to create an instance of Rational with a default value of 1 for the denominator. It does this by calling the primary constructor with the given parameter n and 1 as arguments. Now, if you feed the following to the scala shell:

```
scala> val y = new Rational(3)
```

you should see:

```
created: 3/1
y: Rational = Rational@de1520
```

Every auxiliary constructor must call another constructor of the same class as its first action. The called constructor is either the primary constructor (as in the example above), or else another auxiliary constructor that comes textually before the calling constructor. The net effect of this is that every constructor invocation in Scala will end up eventually calling the primary constructor of the class. The primary constructor is thus the single point of entry of a class.¹

6.5 Reimplementing the `toString` method

The current version of Rational does not display in a nice way. It's time to do something about this. When the interpreter prints out the value of a rational number, it invokes the number's `toString` method. This method has a default implementation in Scala's (and Java's) root class `Object`, where it just prints the class name and a hexadecimal number. This default implementation can be overridden by adding a method `toString` to class Rational:

```
override def toString() = numer+"/"+denom
```

The `override` modifier in front of a method definition signals that a previous method definition is overridden; more on this in [Chapter 10](#).

You can test the new behavior of `Rationals` in the interpreter (since now numbers display correctly, you can remove the `println` statement from the body of class Rational):

```
scala> val x = new Rational(1, 3)
x: Rational = 1/3

scala> val y = new Rational(5, 7)
y: Rational = 5/7

scala> val z = new Rational(3, 2)
z: Rational = 3/2

scala> x.add(y).mul(z)
res7: Rational = 66/42
```

¹This is a little bit more restrictive than in Java, in which any constructor can directly invoke a constructor of the superclass.

6.6 Private methods and fields

So far so good. But there is still something amiss: The numerator and denominator of a rational are unnecessarily large – it prints $66/42$ instead of $11/7$. It would be better to normalize the number by dividing both numerator and denominator with any common divisors they might have. You could do this normalization in every arithmetic operation, but that would lead to a lot of repetition. A more elegant technique is to normalize when a rational number is created. Here's how this is done:

```
class Rational(n: Int, d: Int) {  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
    private val g = gcd(n, d)  
    val numer: Int = n / g  
    val denom: Int = d / g  
    // rest of class as before  
}
```

The new version of `Rational` has a private method `gcd` and a private field `g`. As in Java, a **private** method can be accessed only from inside the class in which it is defined. There's also **protected**, which restricts access to a member to the class in which it is defined and all its subclasses. [Chapter 13](#) contains more material on **private** and **protected**, and other ways to control member visibility.

The `gcd` method computes the greatest common divisor of two numbers. For instance `gcd(12, 8)` is 4. The `g` field takes the result of computing the `gcd` of the two class parameters. The `numer` and `denom` fields then are initialized to the corresponding class parameters divided by `g`.

You can test the correct behavior of `Rational` by creating a non-normalized rational number. If you type

```
scala> val x = new Rational(12, 8)
```

you should get

```
x: Rational = 3/2
```

6.7 Self references

Just like in Java, the reserved word `this` refers to the currently executing object. As an example, consider adding a method, `lessThan`, which tests whether the given rational is smaller than a parameter:

```
def lessThan(that: Rational) =  
    this.numer * that.denom < that.numer * this.denom
```

Here, `this.numer` refers to the numerator of the object in which the `lessThan` method is executed. You can also leave off the `this`-prefix and write just `numer`; the two notations are equivalent.

As an example where you can't do without `this`, consider adding a `max` method to class `Rational` that returns the greater of the given rational number and an argument:

```
def max(that: Rational) =  
    if (this.lessThan(that)) that else this
```

Here, the first `this` is redundant, you could have equally well written `(lessThan(that))`. But the second `this` represents the result of the method in the case where the test returns false; if you omit it, there would be nothing left to return!

6.8 Defining operators

The current implementation of `Rationals` is OK as it stands, but it could be made more convenient to use. You might ask yourself why you can write

`x * y + z`

if `x`, `y`, and `z` are integers or floating point numbers, but you have to write

`x.mul(y).add(z)`

or at least

`x mul y add z`

if they are rational numbers. There's no convincing reason why this should be so. Rational numbers are numbers just like others (in a mathematical sense they are even more natural than, say floating point numbers). Why should you not use the natural arithmetic operators on them? In Scala you can do this. In the rest of this chapter, we walk you through the improvements of the `Rational` class that get you there.

The first step is to replace `add` and friends by the usual mathematical symbols. This is straightforward, as `+`, `-`, `*`, `/` are legal identifiers in Scala. So you simply define methods with these names:

```
class Rational(n: Int, d: Int) {  
    def this(n: Int) = this(n, 1)  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
    private val g = gcd(n, d)  
    val numer: Int = n / g  
    val denom: Int = d / g  
  
    def +(that: Rational): Rational =  
        new Rational(numer * that.denom + that.numer * denom,  
                     denom * that.denom)  
  
    def -(that: Rational): Rational =  
        new Rational(numer * that.denom - that.numer * denom,  
                     denom * that.denom)  
  
    def *(that: Rational): Rational =  
        new Rational(numer * that.numer, denom * that.denom)  
  
    def /(that: Rational): Rational =  
        new Rational(numer * that.denom, denom * that.numer)  
  
    override def toString() = numer+"/"+denom  
}
```

With class `Rational` changed as above, you can now write

```
scala> val x = new Rational(1, 2); val y = new Rational(3, 4)  
x: Rational = 1/2  
y: Rational = 3/4  
  
scala> (x + y) * x
```

```
res0: Rational = 5/8
```

As always, the operator syntax on the last input line is equivalent to two method calls. You could also have written

```
scala> (x.+(y)).*(x)
res1: Rational = 5/8
```

but this would be not as legible.

6.9 Identifiers in Scala

You have now seen the two most important ways to form an identifier in Scala: alphanumeric and operator. Scala has very flexible rules for forming identifiers. Besides the two forms you have seen there are also two others. All four forms of identifier formation are described below.

An *alphanumeric identifier* starts with a letter or an underscore character, which can be followed by further letters, digits, or underscore characters. So examples of alphanumeric identifiers are:

```
bob      x1      _out_      MAX_DECIMAL_NUMBER      CamelCase
```

The ‘\$’-character also counts as a letter, however it is reserved for identifiers generated by the Scala compiler. Identifiers in user programs should not contain ‘\$’ character, even though it will compile; if they do this might lead to name clashes with identifiers generated by the Scala compiler.

An *operator identifier* consists of one or more operator characters. Operator characters are printable ASCII characters such as +, :, ?, ~ or #. More precisely, an operator character belongs to the Unicode set of mathematical symbols(Sm) or other symbols(So), or to the 7 Bit ASCII characters that are not letters, digits, or one of the characters

```
_ ( ) [ ] { } . ; , " ' '
```

Examples of operator identifiers are:

```
+      ++      :::      <?>      :->
```

The Scala compiler will internally “mangle” operator identifiers to turn them into legal Java identifiers with embedded ‘\$’-characters. For instance:

```
scala> val :-> = "hi!"  
$colon$minus$greater: java.lang.String = hi!
```

Because operator identifiers in Scala can become arbitrarily long, there is a small incompatibility between Java and Scala. In Java, the input `x<-y` would be parsed as four lexical symbols, so it would be equivalent to `x < - y`. In Scala, `<-` would be parsed as a single identifier, giving `x <- y`. If you want the first interpretation, you need to separate the ‘<’ and the ‘-’ characters by a space. This is unlikely to be a problem in practice, as very few people would write `x<-y` in Java without inserting spaces or parentheses between the operators.

A *mixed identifier* consists of a alphanumeric identifier, which is followed by an underscore and an operator identifier. Examples of mixed identifiers are:

```
vector_+      success_?      myvar_=
```

Mixed identifiers are useful if you want to attach some explanation to an operator identifier, as in `vector_+`. The mixed identifier form `myvar_=` is generated by the Scala compiler to support *properties*; more on that in [Chapter 21](#).

A *literal identifier* is an arbitrary string enclosed in back-ticks ‘ ` ... ` ’. Examples of literal identifiers are

```
'x'      '<clinit>'      'yield'
```

The idea is that you can put any string that’s accepted by the runtime as an identifier between back ticks. The result is always a Scala identifier. This holds even if the name contained in the back ticks would be a Scala reserved word. A typical use case is accessing the static `yield` method in Java’s `Thread` class. You cannot write `Thread.yield()` because `yield` is a reserved word in Scala. However, you can still name the method in back ticks, e.g. `Thread.`yield`()`.

6.10 Method overloading

Back to class Rational. With the latest changes, you can now do arithmetic operations in the natural style on rational numbers. But one thing still missing is mixed arithmetic. For instance, you cannot multiply a rational number by an integer because the operands of ‘*’ always have to be rationals. So for a rational number r you can’t write `r * 2`, it must be `r * new Rational(2)`, which is less nice.

To make Rational even more convenient, you can add new methods to the class that perform mixed arithmetic on rationals and integers:

```
class Rational {  
    ... // as before  
  
    def +(that: Int): Rational = this + new Rational(that)  
    def -(that: Int): Rational = this - new Rational(that)  
    def *(that: Int): Rational = this * new Rational(that)  
    def /(that: Int): Rational = this / new Rational(that)  
}
```

There are now two versions of each arithmetic operator method: one that takes a rational as argument, the other which takes an integer. In other words, the methods are *overloaded*. In a method call, the correct version of an overloaded method is picked based on the types of the arguments. For instance, if the argument y in `x.*(y)` is a Rational, the method `*` which takes a Rational parameter is picked. But if the argument is an integer, the method `*` which takes a Int parameter is picked instead. You can try this out in the interpreter:

```
scala> val x = new Rational(2, 3)  
x: Rational = 2/3  
  
scala> val y = x * x  
y: Rational = 4/9  
  
scala> val z = y * 2  
z: Rational = 8/9
```

The process of overloading resolution is very similar to what Java does. In every case, the chosen overloaded version is the one which best matches the

static types of the arguments. Sometimes there is no unique best matching version; in that case the compiler will give you an “ambiguous reference” error. To experiment with this, you can enter the following nonsensical line in the interpreter:

```
val wrong = z * (throw new Error)
```

You should get:

```
<console>:5: error: ambiguous reference to overloaded definition,
both method * in class Rational of type (Int)Rational
and method * in class Rational of type (Rational)Rational
match argument types (Nothing)
val wrong = z * (throw new Error)
^
```

The input line above will probably look a bit mystifying to you right now. You’ll find out more background information about what goes on in [Section 7.4](#). In short, the argument here is a `throw`-expression whose type is `Nothing`, which is compatible with either `Int` or `Rational`. That’s why overloading resolution could not pick one of the two overloaded variants of the multiplication method.

6.11 Going further

There’s still room for improvement: Now that you can write `r * 2`, you might also want to swap the operands, as in `2 * r`. Unfortunately this does not work yet:

```
scala> 2 * r
<console>:5: error: overloaded method value * with alternatives
(Double)Double <and> (Float)Float <and> (Long)Long <and> (Int)Int
<and> (Char)Int <and> (Short)Int <and> (Byte)Int cannot be
applied to (Rational)
val res2 = 2 * r
^
```

The problem here is that `2 * r` is equivalent to `2.*(r)`, so it is a method call on the number 2, which is an integer. But the `Int` class contains no multiplication method which takes a `Rational` argument—it couldn't because class `Rational` has been written by you; it is not a standard class in the Scala library.

However, there is another way to solve this problem in Scala: You can create an implicit conversion which automatically converts integers to rationals when needed. Try to add this line in the interpreter:

```
scala> implicit def intToRational(x: Int) = new Rational(x)
```

This defines a conversion method from `Int` to `Rational`. The `implicit` modifier in front of the method tells the compiler to apply it automatically in a number of situations. With the conversion defined, you can now retry the example that failed before:

```
scala> val r = new Rational(2,3)
r: Rational = 2/3
scala> 2 * r
res0: Rational = 4/3
```

As you can glimpse from this example, implicit conversions are a very powerful technique for making libraries more flexible and more convenient to use. Because they are so powerful, they can also be easily misused. You'll find out more on implicit conversions in [Chapter 19](#).

6.12 A word of caution

As this chapter has demonstrated, creating methods with operator names and defining implicit conversions can help you design libraries for which client code is concise and easy to understand. This is even easier to see in [Section 1.1](#) on page 29, where you can compare client code for Java's `BigInteger` with code for Scala's `BigInt`. Scala gives you a great deal of power to design such easy-to-use libraries, but please bear in mind that with power comes responsibility.

If used unartfully, both operator methods and implicit conversions can give rise to client code that is hard to read and understand. Because implicit conversions are applied implicitly by the compiler, not explicitly writ-

ten down in the source code, it can be non-obvious to client programmers what implicit conversions are being applied. And although operator methods will usually make client code more concise, they will only make it more readable to the extent client programmers will be able to recognize and remember the meaning of each operator.

The goal you should keep in mind as you design libraries is not merely enabling concise client code, but readable, understandable client code. Ciseness will often be a bit part of that readability, but you can take it too far. By designing libraries that enable tastefully concise, readable, understandable client code, you can help those client programmers work productively.

6.13 Conclusion

In this section, you have seen more elements of classes in Scala. You have seen how to add parameters to a class, how to define several constructors, how to define operators as methods, and how to customize classes so that they are natural to use. Maybe most importantly, you should have taken away from the treatment in this chapter that objects without any mutable state manifested in `vars` are quite a natural way to code things in Scala.

Chapter 7

Built-in Control Structures

There are not many control structures built into Scala. The only control structures are `if`, `while`, `for`, `try`, `match`, and function calls. The reason Scala has so few is that it has included function literals since its inception. Instead of accumulating one higher-level control structure after another in the base syntax, Scala accumulates them in libraries. The next chapter will show precisely how that is done. This one will show those few control structures that are built in.

One thing you will notice is that almost all of Scala's control structures result in some value. This is the approach taken by functional languages, in which programs are viewed as computing a value, thus the components of a program should also compute values. You can also view this approach as the logical conclusion of a trend already present in imperative languages. In imperative languages, function calls can return a value, even though having the called function update an output variable passed as an argument would work just as well. In addition, imperative languages often have a ternary operator (such as the `?:` operator of C, C++, and Java), which behaves exactly like `if`, but results in a value. Scala adopts this ternary operator model, but calls it `if`. In other words, Scala's `if` can result in a value. Scala then continues this trend by having `for`, `try`, and `match` also result in values.

Programmers can use these result values to simplify their code, just as they use return values of functions. Without this facility, the programmer must create temporary variables just to hold results that are calculated inside a control structure. Removing these temporary variables makes the code a little simpler, and it also prevents many bugs where you set the variable in

one branch but forget to set it in another.

Overall, Scala's basic control structures, minimal as they are, are sufficient to provide all of the essentials from imperative languages. Further, they allow you to shorten your code by consistently having result values. To show you how all of this works, this chapter takes a closer look at each of Scala's basic control structures.

7.1 If expressions

Scala's if works just like in many other languages. It tests a condition and then executes one of two code branches depending on whether the condition holds true. Here is a common example, written in an imperative style:

```
var filename = "default.txt"
if (!args.isEmpty)
    filename = args(0)
```

This code declares a variable, `filename`, and initializes it to a default value. It then uses an `if` expression to check whether any arguments were supplied to the program. If so, it changes the variable to hold the value specified in the arguments list. If there are no arguments, it leaves the variable set to the default.

This code can be written more nicely, because Scala's `if` is an expression that returns a value. Here is the same example that uses an `if-else` expression and is written in a more functional style:

```
val filename =
  if (!args.isEmpty)
    args(0)
  else
    "default.txt"
```

This time, the `if` has two branches. If `args` is not empty, the initial element, `args(0)`, is chosen. Else, the default value is chosen. The `if` expression results in the chosen value, and the `filename` variable is initialized with that value.

This code is slightly shorter, but its real advantage is that it uses a `val` instead of a `var`. Using a `val` is the more functional style, and it helps you in

much the same way as a `final` variable in Java. It tells you that the variable will never change, saving you from scanning all code in the variable's scope to see if it ever changes.

A second advantage to using a `val` instead of a `var` is that it better supports *equational reasoning*. The introduced variable is *equal* to the expression that computes it, assuming that expression has no side effects. Thus, any time you are about to write the variable name, you could instead write the expression. Instead of `println(filename)`, for example, you could just as well write this:

```
println(if (!args.isEmpty) args(0) else "default.txt")
```

The choice is yours. You can write it either way. Using `vals` helps you safely make this kind of refactoring as your code evolves over time.

For both of these reasons, you should look for opportunities to use `vals` wherever possible. They make your code both easier to read and easier to refactor.

Lastly, you may have wondered if an `if` without an `else` returns a value. It does, just not a very useful one. The type of the result is `Unit`, which as mentioned previously, means the expression results in no value. It turns out that a value (and in fact, only one value) exists whose type is `Unit`. It is called the *unit value* and is written `()`. The existence of `()` is how Scala's `Unit` differs from Java's `void`. Try this in the interpreter:

```
scala> val a = if (false) "hi"
a: Unit = ()
scala> val b = if (true) "hi"
b: Unit = ()
scala> a == ()
res0: Boolean = true
```

Any type can be implicitly converted to `Unit` if need be, as illustrated here:

```
scala> val c: Unit = "hi"
c: Unit = ()
```

When a value's type is converted to `Unit`, all information is lost about that value. In essence, `()` is the value that means no value. Its only purpose is to let you use things like `if-without-else` expressions in contexts where an expression is expected. The two other build-in control constructs that result in `()`, while and do loops, are described next.

7.2 While loops

Scala's `while` loop behaves just like in other languages such as Java. The loop has a condition and a body, and the body is executed over and over as long as the condition holds true. Here's an example of a `while` loop used in a function that takes an imperative approach to compute the greatest common denominator of two `Long`s:

```
def gcdLoop(x: Long, y: Long): Long = {  
    var a = x  
    var b = y  
    while (a != 0) {  
        val temp = a  
        a = b % a  
        b = temp  
    }  
    b  
}
```

Scala also has a `do-while` loop. This is a variant of the `while` loop that simply tests the condition after the loop body instead of before. Here's an example:

```
var line = ""  
do {  
    line = readLine  
    println("Read: " + line)  
} while (!line.isEmpty)
```

As mentioned previously, both `while` and `do` loops result in `()`, the unit value.

Because the while loop results in no value, it is often left out of pure functional languages. Such languages have expressions, not loops. Scala includes the while loop nonetheless, because sometimes an imperative solution can be more readable, especially to programmers with a predominantly imperative background. For example, if you want to code an algorithm that repeats a process until some condition changes, a while loop can express it directly while the functional alternative, which likely uses recursion, may be a bit less obvious to some readers of the code.

For example, here's a more functional way to determine a greatest common denominator of two numbers:

```
def gcd(x: Long, y: Long): Long =  
  if (b == 0) a else gcd(b, a % b)
```

Given the same two values for x and y, the gcd function will return the same result as the gcdLoop function, shown earlier in this section. The difference between these two approaches is that gcdLoop is written in an imperative style, using vars and and a while loop, whereas gcd is written in a more functional style that involves recursion (gcd calls itself) and requires no vars. Although the functional gcd is clearly more concise than the imperative gcdLoop, is it more readable? Programmers with a predominantly imperative background may actually find the more verbose gcdLoop easier to understand.

In general, we recommend you challenge while loops in your code in the same way you challenge vars. In fact, while loops and vars often go hand in hand. Because while loops don't result in a value, to make any kind of difference to your program, the while loops will often need to update vars. You can see this in action in the gcdLoop example shown previously. As that while loop does its business, it updates vars a and b. Thus, we suggest you be a bit suspicious of while loops in your code. If there isn't a good justification for a particular while or do loop, try and find a way to do the same thing without it. That said, please keep in mind that your ultimate goal should not be to show off to your colleagues how smart you are, but to maximize the readability and understandability of your code for the people who will be reading it.

7.3 For expressions

Scala's for expression is a Swiss army knife of enumeration. It lets you combine a few simple ingredients in different ways to express a wide variety of enumerations. Simple uses allow common enumerations such as iterating through a sequence of integers. More advanced expressions can iterate over multiple collections of different kinds, can filter out elements based on arbitrary conditions, and can produce new collections.

Iteration through collections

The simplest thing you can do with for is to iterate through all the elements of an entire collection. For example, here is some code that prints out all files in the current directory.

```
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere)
    println(file)
```

The list of files is computed using methods from the Java API. The code creates a `File` on the current directory, and then it calls the standard `listFiles` method.

To print out all of the files, the for expression iterates through them and calls `println` on each one. The `file <- filesHere` syntax creates a new variable `file` and then causes that variable to be set to one element of `filesHere` at a time. For each setting of the variable, the body of the for expression, `println(file)`, is executed.

This syntax works for any kind of collection, not just arrays.¹ One convenient special case is the Range type, which you briefly saw in [Table 5.4](#) on [page 134](#). You can create Ranges using syntax like “1 to 5,” and you can iterate through them with a for. Here is a simple example:

```
scala> for (i <- 1 to 5)
    |   println("Iteration " + i)
Iteration 1
Iteration 2
```

¹To be precise, the expression to the right of the `<-` symbol must extend the `scala.Iterable` trait.

```
Iteration 3  
Iteration 4  
Iteration 5
```

Iterating through integers like this is common in Scala, but not nearly as much as in other languages. In other languages, you might use this facility to iterate through an array, like this:

```
// Not common in Scala...  
for (i <- 0 to filesHere.length - 1)  
    println(filesHere(i))
```

This for expression introduces a variable `i`, sets it in turn to each integer between 0 and `filesHere.length - 1`, and executes the body of the for expression for each setting of `i`. For each setting of `i`, the `i`'th element of `filesHere` is extracted and processed.

The reason this kind of iteration is less common in Scala is that you can just as well iterate over the collection directly. If you do, your code becomes shorter and you sidestep many of the off-by-one errors that arise so frequently when iterating through arrays. Should you start at 0 or 1? Should you add -1, +1, or nothing to the final index? Such questions are easily answered, but easily answered wrong. It is safer to avoid such questions entirely.

Filtering

Sometimes you do not want to iterate through a collection in its entirety. You want to filter it down to some subset. You can do this with a for expression by adding a semicolon plus an if clause to your for expression. For example, the following code lists only those files in the current directory whose names end with `.scala`:

```
for (file <- filesHere; if file.getName.endsWith(".scala"))  
    println(file)
```

You could alternatively accomplish the same goal with this code:

```
for (file <- filesHere)  
    if (file.getName.endsWith(".scala"))  
        println(file)
```

This code yields the same output as the previous code, and likely looks more familiar to programmers with an imperative background. The imperative form, however, is only an option because this particular `for` expression is executed for its printing side-effects and results in the unit value `()`. As will be demonstrated later in this section, the `for` expression is called an “expression” because it can result in an interesting value, a collection whose type is determined by the `for` expression’s `<-` clauses.

You can include more tests if you want. Just keep adding clauses. For example, to be extra defensive, the following code prints only files and not directories. It does so by adding an `if` clause that checks the standard `isFile` method.

```
for (
    file <- filesHere;
    if file.isFile;
    if file.getName.endsWith(".scala")
) println(file)
```

To make long `for` expressions easier to read, you can use curly braces instead of parentheses. As usual inside curly braces, it is not necessary to put semicolons at the ends of lines.

```
for {
    file <- filesHere
    if file.isFile
    if file.getName.endsWith(".scala")
} println(file)
```

Keep in mind that curly braces surrounding a `for` expression’s `<-` and `if` clauses serve the same purpose as parentheses. In particular, variables defined in these clauses, such as `file` in the previous example, are available to be used in the body of the `for` expression. In the previous example, for instance, `file` is passed to `println`. The sole advantage of curly braces in this case is that they allow you to leave off the semi-colons at the end of the clauses, which the parentheses require.

Nested iteration

If you add multiple `<-` clauses, you will get nested “loops.” For example, the following `for` expression has two nested loops. The outer loop iterates through `filesHere`, and the inner loop iterates through `fileLines(file)` for any file that ends with `.scala`.

```
def fileLines(file: java.io.File) =  
    scala.io.Source.fromFile(file).getLines  
  
def grep(pattern: String) =  
    for {  
        file <- filesHere  
        if file.getName.endsWith(".scala")  
        line <- fileLines(file)  
        if line.trim.matches(pattern)  
    } println(file + ": " + line.trim)  
  
grep(".*gcd.*")
```

Mid-stream assignment

Note that the previous code repeats the expression `line.trim`. This is a non-trivial computation, so you might want to only compute it once. You can do this by binding the result to a variable using an equals sign (`=`). The bound variable is introduced and used just like a `val`, only with the `val` keyword left out.

```
def grep(pattern: String) =  
    for {  
        file <- filesHere  
        if file.getName.endsWith(".scala")  
        line <- fileLines(file)  
        trimmed = line.trim  
        if trimmed.matches(pattern)  
    } println(file + ": " + trimmed)  
  
grep(".*gcd.*")
```

In this code, a variable named `trimmed` is introduced halfway through the `for` expression. That variable is initialized to the result of `line.trim`. The

rest of the for expression then uses the new variable in a couple of places, once in an if and once in a `println`.

Producing a new collection

While all of the examples so far have operated on the iterated values and then forgotten them, you can also generate a value to remember for each iteration. You simply prefix the body of the for expression by the keyword `yield`. For example, here is a function that identifies the `.scala` files and stores them in an array:

```
def scalaFiles =  
  for {  
    file <- filesHere  
    if file.getName.endsWith(".scala")  
  } yield file
```

Each time the body of the for expression executes it produces one value, in this case simply `file`. When the for expression completes, all of these values are returned in a single expression. The type of the resulting collection is based on the kind of collections processed in the iteration clauses. In this case the result is an array, because `filesHere` is an array.

Be careful, by the way, where you place the `yield` keyword. The syntax of a for-yield expression is like this:

```
for clauses yield body
```

The `yield` goes before the entire body. Even if the body is a block surrounded by curly braces, put the `yield` before the first curly brace, not before the last expression of the block. Avoid the temptation to write things like this:

```
for (file <- filesHere; if file.getName.endsWith(".scala")) {  
  yield file // Syntax error!  
}
```

Stepping back, for-yield is another way that Scala supports functional programming. If you compute a collection this way, you can ignore integer indexes and think less about the order things happen. You can focus on the essence of the code: how the iteration works, and what it should produce.

For example, in Section 4.6 you saw two versions of `printMultiTable`, a function that prints a multiplication table to the standard output. Although the second version, shown on [page 107](#), is in a more functional style than the first, shown on [page 104](#), it can be made even more functional by refactoring so that the function returns the multiplication table as a `String`. One way to do this would be to use for expressions with `yield` like this:

```
def multiTable = {  
    val table = for (i <- 1 to 10) yield {  
        val row = for (j <- 1 to 10) yield {  
            val prod = (i * j).toString  
            String.format("%4s", Array(prod))  
        }  
        row.mkString + '\n'  
    }  
    table.mkString  
}  
  
println(multiTable)
```

7.4 Try expressions

Scala's exceptions behave just like in many other languages. Instead of returning a value in the normal way, a method can terminate by throwing an exception. The method's caller can either catch and handle that exception, or it can itself simply terminate, in which case the exception propagates to the caller's caller. The exception propagates in this way, unwinding the call stack, until a method handles it or there are no more methods left.

Throwing exceptions

Throwing an exception looks the same as in Java. You create an exception object and then you throw it with the `throw` keyword:

```
throw new NullPointerException
```

One note of interest is that `throw` returns a value, too... sort of. Here is an example of “returning” a value from a `throw`:

```
val half =  
  if (n % 2 == 0)  
    n / 2  
  else  
    throw new Exception("n must be even")
```

What happens here is that if `n` is even, `half` will be initialized to half of `n`. If `n` is not even, then an exception will be thrown before `half` can be initialized to anything at all. Because of this, it is safe to treat a thrown exception as any kind of value whatsoever. Any context that tries to use the return from a `throw` will never get to do so, and thus no harm will come.

Technically, an exception throw returns type `Nothing`. You can use a `throw` as an expression even though it will never actually evaluate to anything. This little bit of technical gymnastics might sound weird, but is frequently useful in cases like the previous example. One branch of an `if` computes a value, while the other throws an exception and computes `Nothing`. The type of the whole `if` expression is then the type of that branch which does compute something. Type `Nothing` is discussed further in [Section 10.17](#).

Catching exceptions

You catch exceptions using the following syntax:

```
try {  
  doSomething()  
}  
catch {  
  case ex: IOException => println("Oops!")  
  case ex: NullPointerException => println("Oops!!")  
}
```

This unusual syntax is chosen for its consistency with an important part of Scala: *pattern matching*. Pattern matching, a powerful feature, is described briefly in this chapter and in more detail in [Chapter 12](#).

The behavior of this `try-catch` expression is exactly as in other languages with exceptions.² The body is executed, and if it throws an exception,

²One difference from Java that you'll quickly notice in Scala is that unlike Java, Scala

tion, each catch clause is tried in turn. In this example, if the exception is of type `IOException`, then the first clause will execute. If it is of type `NullPointerException`, the second clause will execute. If the exception is of neither type, then the try-catch will terminate and the exception will propagate further.

The finally clause

You can wrap an expression with a `finally` clause if you want to cause some code to execute even if a method is terminated early. For example, you might want to be sure an open file gets closed even if a method exits by throwing an exception.

```
val file = openFile()
try {
    // use the file
}
finally {
    file.close() // be sure to close the file
}
```

Yielding a value

As with most other Scala control structures, try-catch-finally results in a value. For example, here is how you can try to parse a URL but use a default value if the URL is badly formed:

```
val url =
try {
    new URL(path)
}
catch {
    case e: MalformedURLException =>
        new URL("http://www.scala-lang.org")
}
```

does not require you to catch checked exceptions, or declare them in a throws clause. You can declare a throws clause if you wish with the `@throws` annotation, but it is not required.

The result is that of the `try` clause if no exception is thrown, or the relevant `catch` clause if an exception is thrown and caught. If an exception is thrown but not caught, the expression has no result at all. The value computed in the `finally` clause, if there is one, is dropped. Usually `finally` clauses do some kind of clean up such as closing a file, and the programmer would prefer to hold onto the value computed in a different part of the `try`.

If you're familiar with Java, it's worth noting that Scala's behavior differs from Java only because Java's `try-finally` does not result in a value. As in Java, if a `finally` clause includes an explicit return statement, or throws an exception, that return value or exception will "overrule" any previous one that originated in the `try` block or one of its `catch` clauses. For example, given:

```
def f(): Int = try { return 1 } finally { return 2 }
```

calling `f()` results in 2. By contrast, given:

```
def g(): Int = try { 1 } finally { 2 }
```

calling `g()` results in 1.

7.5 Match expressions

The final build-in control structure you will want to know about is the `match` expression. Match expressions let you select from a number of alternatives, just like `switch` statements in other languages. In general a `match` expression lets you select using arbitrary *patterns*, as described in [Chapter 12](#). The general form can wait. For now, just consider using `match` to select among a number of alternatives.

As an example, the following code reads a food name from the argument list and prints a companion to that food.

```
val firstArg = if (args.length > 0) args(0) else ""  
firstArg match {  
    case "salt" => println("pepper")  
    case "chips" => println("salsa")  
    case "eggs" => println("bacon")  
    case _ => println("huh?")
```

```
}
```

This match expression examines `firstArg`, which has been set to the first argument out of the argument list. If it is the string “salt,” it prints “pepper,” while if it is the string “chips,” it prints “salsa,” and so on. The default case is specified with an underscore (`_`), a wildcard symbol frequently used in Scala as a placeholder for a completely unknown value.

There are a few important differences from Java’s `switch` statement. One is that any kind of constant, as well as other things, can be used in case clauses in Scala, not just the integer-type and enum constants of Java’s case statements. In this case, the alternatives are strings. Another difference is that there are no `breaks` at the end of each alternative. Instead the `break` is implicit, and there is no fall through from one alternative to the next. The common case—not falling through—becomes shorter, and a source of errors is avoided because programmers can no longer fall through by accident.

The most surprising difference, though maybe not so surprising by now, is that match expressions result in a value. In the previous example, each alternative in the match expression prints out a value. It would work just as well to return the value rather than printing it, as shown here:

```
val firstArg = if (!args.isEmpty) args(0) else ""  
val friend =  
    firstArg match {  
        case "salt" => "pepper"  
        case "chips" => "salsa"  
        case "eggs" => "bacon"  
        case _ => "huh?"  
    }  
    println(friend)
```

Here the value that results from the match expression is stored in the `friend` variable. Aside from the code getting shorter (in number of tokens, anyway), the code now disentangles two separate concerns: first it chooses a food, and then it prints it.

7.6 Living without break and continue

You may have noticed that there has been no mention of `break` or `continue`. Scala leaves out these commands because they do not mesh well with function literals, a feature described in the next chapter. It is clear what `continue` means inside a `while` loop, but what would it mean inside a function literal? While Scala supports both imperative and functional styles of programming, in this case it leans slightly towards functional programming in exchange for simplifying the language.

Do not worry, though. There are many ways to program without `break` and `continue`, and if you take advantage of function literals, those alternatives can often be shorter than the original code.

The simplest approach is to replace every `continue` by an `if` and every `break` by a boolean variable. The boolean variable indicates whether the enclosing `while` loop should continue. For example, suppose you are searching through an argument list for a string that ends with “`.scala`” but does not start with a hyphen. That is, you are looking for a Scala file but want to ignore any options. In Java you could—if you were quite fond of `while` loops, `break`, and `continue`—write the following:

```
// This is Java...
int i = 0;
boolean foundIt = false;

while (i < args.length) {
    if (args[i].startsWith("-")) {
        i = i + 1;
        continue;
    }
    if (args[i].endsWith(".scala")) {
        foundIt = true;
        break;
    }
    i = i + 1;
}
```

To transliterate this directly to Scala, instead of doing an `if` and then a `continue`, you could write an `if` that surrounds the entire remainder of the

while loop. To get rid of the break, you would normally add a boolean variable indicating whether to keep going, but in this case you can reuse foundIt. Using both of these tricks, the code ends up looking like this:

```
var i = 0
var foundIt = false

while (i < args.length && !foundIt) {
    if (!args(i).startsWith("-")) {
        if (args(i).endsWith(".scala"))
            foundIt = true
    }
    i = i + 1
}
```

This version is quite similar to the original. All the basic chunks of code are still there and in the same order. There is a test that `i < args.length`, a check for `"-"`, and then a check for `".scala"`.

To make the code more functional by getting rid of the var, one approach you can try is to rewrite the loop as a recursive function. Continuing the previous example, you could define a `searchFrom` function that takes an integer as an input, searches forward from there, and then returns the index of the desired argument. Using this technique the code would look like this:

```
def searchFrom(i: Int): Int =
    if (i >= args.length) // don't go past the end
        -1
    else if (args(i).startsWith("-")) // skip options
        searchFrom(i + 1)
    else if (args(i).endsWith(".scala")) // found it!
        i
    else
        searchFrom(i + 1) // keep looking

val i = searchFrom(0)
```

This version is longer, but it has the advantage that `searchFrom` is given a human-meaningful name.

Really, though, to write the cleanest, most concise code in Scala you must get familiar with function literals. There are many methods in the Scala

API that take advantage of function literals, and you just have to know they are there and be comfortable writing function literals. In this case, you could write the code as briefly as the following:

```
args.findIndexOf(  
    arg => !arg.startsWith("-") && arg.endsWith(".scala")  
)
```

This version uses `findIndex0f`, one of many methods in the standard library that help you with common looping patterns. This method finds the first element of a collection that matches some condition, and returns its index, or `-1` if no matching element is found. The interesting thing is that the condition is described right inline as a block of code. The function literal syntax may still feel a bit unfamiliar to you, but if you squint your eyes at it you'll see that the argument to `findIndex0f` first checks for `"-"` and then checks for `".scala"`. The entire block of code gets passed to the `findIndex0f` method, and so `findIndex0f` can run that block of code whenever it needs to do its job. Because `findIndex0f` knows just what you are trying to do, you can leave out several details and the code becomes quite short.

7.7 Conclusion

Scala's build-in control structures are minimal, but they do the job. They act much like their imperative equivalents, but because they tend to result in a value, they support a functional style too. Just as important, they are careful in what they omit, thus leaving room for one of Scala's most powerful features, the function literal.

Chapter 8

Functions and Closures

When programs get larger, you need some way to divide them into smaller, more manageable pieces. For dividing up control flow, Scala offers an approach familiar to all experienced programmers: divide the code into functions. In fact, Scala offers several ways to define functions that are not present in Java. Besides methods, which are functions that are members of some template (class, trait, or singleton object), there are also nested functions and function literals. Functions can also be values. This chapter shows how to write and use such functions in Scala.

8.1 Methods

The most common way to define a function is as a member of some object. Such a function is called a *method*. As an example, here are two methods that together read a file with a given name and print out all lines whose length exceeds a given width. Every printed line is prefixed with the name of the file it appears in:

```
import scala.io.Source
object LongLines {
    def processFile(filename: String, width: Int) {
        val source = Source.fromFile(filename)
        for (line <- source.getLines)
            processLine(filename, width, line)
    }
    def processLine(filename: String, width: Int, line: String) {
```

```
    if (line.length > width)
        println(filename+": "+line.trim)
    }
}
```

The `processFile` method takes a `filename` and `width` as parameters. It creates a `Source` object from the file name and processes all lines of that file by calling the *helper method* `processLine`. The `processLine` method takes three parameters: a `filename`, a `width`, and a `line`. It tests whether the length of the line is greater than the given width, and, if so, it prints the `filename`, followed by a colon, and the `line`.

To make a complete application, the two methods can be placed in an object `LongLines` as follows:

```
import scala.io.Source
object LongLines {
    def processFile ...
    private def processLine ...
    def main(args: Array[String]) {
        val width = args(0).toInt
        for (arg <- args.drop(1))
            processFile(arg, width)
    }
}
```

You could then use the `LongLines` application to find the long lines in a set of files. Here's an example:

```
scala LongLines 40 *.scala
LongLines.scala:  def processFile(filename: String, width: Int) {
LongLines.scala:      if (line.length > width) println(filename+": "+line)
LongLines.scala:      val source = Source.fromFile(filename)
Queues.scala: class Queue[T](leading: List[T], trailing: List[T]) {
    ...
}
```

So far, this is very similar to what you would do in any object-oriented language. However the concept of a function in Scala is more general than a method. In fact, you can define and use a function in roughly the same way

as, say, an integer value. These added capabilities will be explained in the following sections.

8.2 Nested functions

The construction of the `processFile` method in the previous section demonstrated an important design principle of the functional programming style: programs should be decomposed into many small functions that each do a well-defined thing. Individual functions are often quite small. The advantage of this style is that it gives a programmer many building blocks that can be flexibly composed to do more difficult things. Each building block should be simple enough to be understood individually. However, a problem with the “many small functions” approach is that all these helper function names have a tendency to pollute the program name space. In the interpreter shell this not so much of a problem. But once functions are packaged in reusable classes and objects, it’s desirable to hide the helper functions from clients of a class, because they might not make sense individually. In Java, you would use a `private` method for this purpose. This private-method approach works in Scala as well, but Scala offers an alternative approach: you can define functions inside other functions. Just like local variables, such nested functions are visible only in their enclosing block. Here’s how you can use this scheme to clean-up the `processFile` functions:

```
def processFile(filename: String, width: Int) {  
    def processLine(filename: String, width: Int, line: String) {  
        if (line.length > width) print(filename+": "+line)  
    }  
    val source = Source.fromFile(filename)  
    for (line <- source.getLines) {  
        processLine(filename, width, line)  
    }  
}
```

Once you have moved the helper function `processLine` inside `processFile`, another improvement becomes possible. Notice how `filename` and `width` are passed unchanged into the helper function? This

is not necessary, because you can just use the parameters of the outer `processLine` function:

```
def processFile(filename: String, width: Int) {  
    def processLine(line: String) {  
        if (line.length > width) print(filename+": "+line)  
    }  
    val source = Source.fromFile(filename)  
    for (line <- source.getLines) {  
        processLine(line)  
    }  
}
```

Regarding scoping, nested function definitions behave just like nested variable definitions: A nested definition can access everything that's defined around it, and the defined entity is visible only in the enclosing block. It's a simple principle, but very powerful, in particular in connection with first-class functions, which are described next.

8.3 First-class functions

In addition to methods and nested functions, Scala also offers *first-class functions*. At runtime, first-class functions are represented by objects called *function values*. Like other values, function values can be passed as parameters to other functions, returned as results, or assigned to variables. In the source code, you can express first-class functions in a shorthand form called *function literals*. We introduced function literals in [Chapter 2](#) and showed the basic syntax in [Figure 2.2 on page 61](#).

A function literal is compiled into a class that when instantiated at runtime is a function value.¹ Thus the distinction between function literals and values is that function literals exist in the source code, whereas function values exist as objects at runtime. You can use the term “first-class function” to refer to either a function literal or value, as first-class function means the

¹Every function value is an instance of some class that extends one of several `FunctionN` traits in package `scala`, such as `Function0` for functions with no parameters, `Function1` for functions with one parameter, and so on. Each `FunctionN` trait has an `apply` method used to invoke the function.

kind of function that's represented at runtime by an object. Each function in a Scala program, both in source code and at runtime, is either a method, a nested function, or a first-class function.

Here is a simple example of a function literal that adds one to a number:

```
(x: Int) => x + 1
```

The `=>` designates that this function converts the thing on the left (any integer `x`) to the thing on the right (`x + 1`). So, this is a function mapping any integer `x` to `x + 1`.

Function values are objects, so you can store them in variables if you like. They are functions, too, so you can invoke them using the function-call notation. Here is an example of both activities:

```
scala> var increase = (x: Int) => x + 1
increase: (Int) => Int = <function>
scala> increase(10)
res0: Int = 11
```

As with any other var, you can assign a different function value to `increase` whenever you like. For example:

```
scala> increase = (x: Int) => x + 9999
increase: (Int) => Int = <function>
scala> increase(10)
res2: Int = 10009
```

If you want to have more than one statement in the function literal, surround it by curly braces and put one statement per line. Just like a method, when the function value is invoked, all of the statements will be executed, and the value returned from the function is whatever the expression on the last line generates.

```
scala> increase = (x: Int) => {
    |   println("Line 1")
    |   println("Line 2")
    |   println("Line 3")
    |   x + 1
    | }
```

```
increase: (Int) => Int = <function>
scala> increase(10)
Line 1
Line 2
Line 3
res4: Int = 11
```

Careful library writers will give you a lot of opportunities to use first-class functions. For example, a `foreach` method is available for all collections.² It takes a function as an argument and invokes that function on each of its elements. Here is how it can be used to print out all of the elements of a list:

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)
scala> someNumbers.foreach((x: Int) => println(x))
-11
-10
-5
0
5
10
```

As another example, collection types also have a `filter` method. This method selects those elements of a collection that pass a test the user supplies. That test is supplied using a function. For example, the function `(x: Int) => x > 0` could be used for filtering. This function maps positive integers to true and all other integers to false. Here is how to use it with `filter`:

```
scala> someNumbers.filter((x: Int) => x > 0)
res6: List[Int] = List(5, 10)
```

²The `foreach` method is defined in trait `Iterable`, which is extended by trait `Collection`, whose subtypes include `List`, `Set`, `Array`, and `Map`.

8.4 Short forms of function literals

Scala provides a number of ways to leave out redundant information and write function literals more briefly. Keep your eyes open for these opportunities, because they allow you to remove clutter from your code.

One way to make a function literal more brief is to leave off the parameter types. For example, the previous example with filter could be written like this:

```
scala> someNumbers.filter((x) => x > 0)
res7: List[Int] = List(5, 10)
```

The Scala compiler knows that `x` must be an integer, because it sees that you are immediately using the function to filter a list of integers (referred to by `someNumbers`). This is called *target typing*. The precise definition is not important, though. In practice, you can try writing a function literal without the argument type, and if the compiler gets confused, you can add the type. Over time you'll get a feel for which situations the compiler can and cannot puzzle out.

Another way to remove useless characters is to leave out parentheses when they are not needed. You can leave out the parentheses around a function argument if the type is inferred:

```
scala> someNumbers.filter(x => x > 0)
res8: List[Int] = List(5, 10)
```

8.5 Placeholder syntax

To make a function literal even more concise, you can use underscores as placeholders for one or more parameters, so long as each parameter appears only one time within the function literal. For example, `_ > 0` is very short notation for a function that checks whether a value is greater than zero. Here's an example:

```
scala> someNumbers.filter(_ > 0)
res10: List[Int] = List(5, 10)
```

You can think of the underscore as a “blank” in the expression that needs to be “filled in.” This blank will be filled in with an argument to the function each time the function is invoked. For example, given that `someNumbers` was initialized on page 174 to the value `List(-11, -10, -5, 0, 5, 10)`, the `filter` method will replace the blank in `_ > 0` first with a `-11`, as in `-11 > 0`, then with a `-10`, as in `-10 > 0`, then with a `-5`, as in `-5 > 0`, and so on to the end of the `List`. The function literal `_ > 0`, therefore, is equivalent to the slightly more verbose `x => x > 0`, as demonstrated here:

```
scala> someNumbers.filter(x => x > 0)
res11: List[Int] = List(5, 10)
```

Sometimes when you use underscores as placeholders for parameters, the compiler might not have enough information to infer missing parameter types. For example:

```
scala> val f = _ + _
<console>:4: error: missing parameter type for expanded
function ((x$1, x$2) => x$1.$plus(x$2))
      val f = _ + _
                  ^
<console>:4: error: missing parameter type for expanded
function ((x$1: <error>, x$2) => x$1.$plus(x$2))
      val f = _ + _
                  ^
```

In such cases, you can specify the types using a colon, like this:

```
scala> val f = (_: Int) + (_: Int)
f: (Int, Int) => Int = <function>
scala> f(5, 10)
res12: Int = 15
```

Note that `_ + _` was expanded into a literal for a function that takes two parameters. This is why you can use this short form only if each parameter appears in the function literal at most once. The first underscore represents the first parameter, the second underscore the second parameter, the third underscore the third parameter, and so on.

8.6 Partially applied functions

Although the previous examples substitute underscores in place of individual parameters, you can also replace an entire parameter list with an underscore. For example, instead of writing `println(_)`, you can write `println _`. Here's an example:

```
someNumbers.foreach(println _)
```

Scala treats this short form exactly as if you had written the following:

```
someNumbers.foreach(x => println(x))
```

Thus, the underscore in this case is not a placeholder for a single parameter. It is a placeholder for an entire parameter list. Remember that you need to leave a space between the function name and the underscore, because otherwise the compiler will think you are referring to a different symbol, such as for example, a method named `println_`, which likely does not exist.

When you use an underscore in this way, you are expressing what's called a *partially applied function*. In Scala, when you invoke a function, passing in any needed arguments, you *apply* that function *to* the arguments. For example, given this function:

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (Int, Int, Int)Int
```

You could apply the function `sum` to the arguments 1, 2, and 3 like this:

```
scala> sum(1, 2, 3)
res13: Int = 6
```

A partially applied function is an expression in which you don't supply all of the arguments needed by the function. Instead, you supply some, or none, of the needed arguments. For example, here's a partially applied function expression involving `sum`, in which you supply none of the three required arguments:

```
scala> val a = sum _
a: (Int, Int, Int) => Int = <function>
```

Given this code, the Scala compiler instantiates a function value that takes the three Int parameters missing from the partially applied function expression, `sum _`, and assigns a reference to that new function value to the variable `a`. When you apply three arguments to this new function value, it will turn around and invoke `sum`, passing in those same three arguments. Here's an example:

```
scala> a(1, 2, 3)
res14: Int = 6
```

This may seem a bit mysterious, so to make sure you understand what's going on, here's what just happened: The variable named `a` refers to an object, which is a function value. That function value is an instance of a class generated automatically by the Scala compiler from the partially applied function expression, `sum _`. The class generated by the compiler has an `apply` method that takes three arguments.³ The reason the generated class's `apply` method takes three arguments is that three is the number of arguments missing in the `sum _` expression (because `sum` takes three arguments). The Scala compiler translates the expression `a(1, 2, 3)` into an invocation of the function value's `apply` method, passing in the three arguments 1, 2, and 3. Thus, `a(1, 2, 3)` is a short form for:

```
scala> a.apply(1, 2, 3)
res15: Int = 6
```

This `apply` method, defined in the class generated automatically by the Scala compiler from the expression `sum _`, simply forwards those three missing parameters to `sum`, and returns the result. In this case `apply` invokes `sum(1, 2, 3)`, and returns what `sum` returns, which is 6.

Another way to think about this kind of expression, in which an underscore used to represent an entire parameter list, is as a way to transform a method or nested function, either of which will have a named defined with `def`, into a first-class function. For example, if you have a nested function, such as `sum(a: Int, b: Int, c: Int): Int`, you can "wrap" it in a function value whose `apply` method has the same parameter list and result types. When you apply this function value to some arguments, it in turn applies `sum`

³The generated class extends trait `Function3`, which declares the three-arg `apply` method.

to those same arguments, and returns the result. Although you can't assign a method or nested function to a variable, or pass it as an argument to another function, you can do these things if you wrap the method or nested function in a function value by placing an underscore after its name.

Now, although `sum_` is indeed a partially applied function, it may not be obvious to you why it is called this. It has this name because you are not applying that function to all of its arguments. In the case of `sum_`, you are applying it to *none* of its arguments. But you can also express a partially applied function by supply *some* but not all of the required arguments. Here's an example:

```
scala> val b = sum(1, _: Int, 3)
b: (Int) => Int = <function>
```

In this case, you've supplied the first and last argument to `sum`, but the middle argument is missing. Since only one argument is missing, the Scala compiler generates a new function class whose `apply` method takes one argument. When invoked with that one argument, this functions `apply` method invokes `sum`, passing in 1, the argument passed to the function, and 3. Here's an example:

```
scala> b(2)
res5: Int = 6
```

In this case, `apply` invoked `sum(1, 2, 3)`.

```
scala> b(5)
res6: Int = 9
```

And in this case, `apply` invoked `sum(1, 5, 3)`.

If you are writing a partially applied function expression in which you leave off all parameters, such as `println_` or `sum_`, you can express it more concisely by leaving off the underscore if a function is required at that point in the code. For example, instead of printing out each of the numbers in `someNumbers`, which was defined on [page 174](#), like this:

```
someNumbers.foreach(println _)
```

You could just write:

```
someNumbers.foreach(println)
```

This last form is allowed only in places where a function is required, such as the invocation of `foreach` in this example. The compiler knows a function is required in this case, because `foreach` requires that a function be passed as an argument. In situations where a function is not required, attempting to use this form will cause a compilation error. Here's an example:

```
scala> val c = sum
<console>:5: error: missing arguments for method sum in
object $iw;
follow this method with '_' if you want to treat it as a
partially applied function
    val c = sum
               ^
               ^
               ^

scala> val d = sum _
d: (Int, Int, Int) => Int = <function>
scala> d(10, 20, 30)
res16: Int = 60
```

This example highlights a difference in the design tradeoffs of Scala and classical functional languages such as Haskell or ML. In these languages, partially applied functions are considered the normal case. Furthermore, these languages have a fairly strict static type system that will usually highlight every error with partial applications that you can make. Scala bears a much closer relation to imperative languages such as Java, where a method that's not applied to all its arguments is considered an error. Furthermore, the object-oriented tradition of subtyping and a universal root type⁴ accepts some programs that would be considered erroneous in classical functional languages. For instance, say you mistook the `drop(n: Int)` method of `List` for `tail()`, and you therefore forgot you need to pass a number to `drop`. You might write:

```
println(drop)
```

⁴Scala's universal root type, `Any`, will be described in Chapter 10.

Had Scala adopted the classical functional tradition that partially applied functions are OK everywhere, this code would typecheck. However, you might be surprised to find out that the output printed by this `println` statement would always be `<function>`! What would have happened is that the expression `drop` would have been treated as a function object. Because `println` takes objects of any type, this would compile OK, but it would have given an unexpected result.

To avoid situations like this, Scala normally requires you to specify function arguments that are left out explicitly, even if the indication is as simple as a ‘`_`’. It allows you to leave off even the `_` only when a function type is expected.

8.7 Closures

So far in this chapter, all the examples of function literals that we’ve given have referred only to passed parameters. For example, in `(x: Int) => x > 0`, the only variable used in the function body, `x > 0`, is `x`, which is defined as a parameter to the function. You can, however, refer to variables defined elsewhere. Here’s an example:

```
(x: Int) => x + more // how much more?
```

This function adds “more” to its argument, but what is `more`? From the point of view of this function, `more` is a *free variable*, because the function literal does not itself give a meaning to it. The `x` variable, by contrast, is a *bound variable*, because it does have a meaning in the context of the function: it is defined as the function’s lone parameter, an `Int`.

If you try using this function literal by itself, without any `more` defined in its scope, the compiler will complain:

```
scala> (x: Int) => x + more
<console>:5: error: not found: value more
          (x: Int) => x + more
                           ^
```

On the other hand, the same function literal will work fine so long as there is something available named `more`:

```
scala> var more = 1
more: Int = 1

scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = <function>

scala> addMore(10)
res18: Int = 11
```

The function value (the object) that's created at runtime from this function literal is called a *closure*. The name arises from the act of “closing” the function literal by “capturing” the bindings of its free variables. A function literal with no free variables, such as $(x: \text{Int}) \Rightarrow x + 1$, is called a *closed term*, where a *term* is a bit of source code. Thus a function value created at runtime from this function literal is technically not a closure, because $(x: \text{Int}) \Rightarrow x + 1$ is already closed as written. But any function literal with free variables, such as $(x: \text{Int}) \Rightarrow x + \text{more}$, is an open term. Therefore, any function value created at runtime from $(x: \text{Int}) \Rightarrow x + \text{more}$ will by definition require that a binding for its free variable, `more`, be captured. The resulting function value, which will contain a reference to the captured `more` variable, is called a closure, because the function value is the end product of the act of closing the open term, $(x: \text{Int}) \Rightarrow x + \text{more}$.

This example brings up a question: what happens if `more` changes after the closure is created? In Scala, the answer is that the closure sees the change. For example:

```
scala> more = 9999
more: Int = 9999

scala> addMore(10)
res20: Int = 10009
```

Intuitively, Scala's closures capture variables themselves, not the value to which variables refer.⁵ As the previous example demonstrates, the closure created for $(x: \text{Int}) \Rightarrow x + \text{more}$ sees the change to `more` made outside the closure. The same is true in the opposite direction. Changes made to a captured variable made by a closure is visible outside the closure. Here's an example.

⁵By contrast, Java's inner classes do not allow you to access modifiable variables in surrounding scopes at all.

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> var sum = 0
sum: Int = 0

scala> someNumbers.foreach(sum += _)

scala> sum
res22: Int = -11
```

This example uses a roundabout way to sum the numbers in a `List` to demonstrate how Scala captures variables in closures. Variable `sum` is in a surrounding scope from the function literal `sum += _`, which adds numbers to `sum`. Even though it is the closure modifying `sum` at runtime, the resulting total, `-11`, is still visible outside the closure.

What if a closure accesses some variable that has several different copies as the program runs? For example, what if a closure uses a local variable of some function, and the function is invoked many times? Which instance of that variable gets used at each access?

There is only one answer that is consistent with the rest of the language: the instance used is the one that was active at the time the closure was created. For example, here is a function that creates and returns “increase” closures:

```
def makeIncreaser(more: Int) = (x: Int) => x + more
```

This function can be called multiple times, to create multiple closures. Each closure will access the `more` variable that was active when the closure was created.

```
scala> val inc1 = makeIncreaser(1)
inc1: (Int) => Int = <function>

scala> val inc9999 = makeIncreaser(9999)
inc9999: (Int) => Int = <function>
```

When you call `makeIncreaser(1)`, a closure is created and returned that captures the value `1` as the binding for `more`. Similarly, when you call `makeIncreaser(9999)`, a closure that captures the value `9999` for `more` is returned. When you apply these closures to arguments (in this case, there’s

just one argument, `x`, which must be passed in), the result that comes back depends on how `more` was defined when the closure was created:

```
scala> inc1(10)
res23: Int = 11
scala> inc9999(10)
res24: Int = 10009
```

8.8 Repeated parameters

Scala allows you to indicate that the last parameter to a function may be repeated. This allows clients to pass variable length argument lists to the function. To denote a repeated parameter, place an asterisk after the type of the parameter. For example:

```
scala> def echo(args: String*) = for (arg <- args) println(arg)
echo: (String*)Unit
```

Defined this way, `echo` can be called with zero to many `String` arguments:

```
scala> echo()
scala> echo("one")
one
scala> echo("hello", "world!")
hello
world!
scala> echo("1", "2", "3", "4")
1
2
3
4
```

Inside the function, the type of the repeated parameter is an `Array` of the declared type of the parameter. Thus, the type of `args` inside the `echo` function, which is declared as type “`String*`” is actually `Array[String]`. Nevertheless, if you have an array of the appropriate type, and attempt to pass it as a repeated parameter, you’ll get a compiler error:

```
scala> val arr = Array("What's", "up", "doc?")
arr: Array[java.lang.String] = [Ljava.lang.String;@f4ec00
scala> echo(arr)
<console>:7: error: type mismatch;
 found   : Array[java.lang.String]
 required: String
        echo(arr)
               ^
```

To accomplish this, you'll need to append the array argument with a colon and an “`_*`” symbol, like this:

```
scala> echo(arr: _*)
What's
up
doc?
```

8.9 Tail recursion

In [Section 7.2](#) on page 154, we mentioned that to transform a while loop that updates vars into a more functional style that uses only vals, you may sometimes need to use recursion.⁶ Here's an example of a recursive function that approximates a value by repeatedly improving a guess until it is good enough:

```
def approximate(guess: Domain) : Domain =
  if (isGoodEnough(guess)) guess
  else approximate(improve(guess))
```

A function like this is often used in search problems, with the appropriate implementations for the type `Domain` and the `isGoodEnough` and `improve` functions. If you want the `approximate` function to run faster, you might be tempted to write it with a while loop to try and speed it up, like this:

⁶A recursive function is one that calls itself.

```
def approximateLoop(initialGuess: Domain): Domain = {
    var guess = initialGuess
    while (!isGoodEnough(guess))
        guess = improve(guess)
    guess
}
```

Which of the two versions of `approximate` is preferable? In terms of brevity and `var` avoidance, the first, functional one wins.⁷ But is the imperative approach perhaps more efficient? In fact, if we measure execution times it turns out that they are almost exactly the same! This might seem surprising, because a recursive call looks much more expensive than a simple jump from the end of a loop to its beginning.

However, in the case of `approximate` above, the Scala compiler is able to apply an important optimization. Note that the recursive call to `approximate` is the last thing that happens in the evaluation of the function's body. Functions like `approximate`, which call themselves as their last action, are called *tail recursive*. The Scala compiler detects tail recursion and replaces it with a jump back to the beginning of the function, after updating the function parameters with the new values. So the compiled code for `approximate` is essentially the same as the compiled code for `approximateLoop`. Both functions compile down to the same thirteen instructions of Java bytecodes. If you look through the bytecodes generated by the Scala compiler for the tail recursive method, `approximate`, you'll see that although both `isGoodEnough` and `improve` are invoked in the body of the method, `approximate` is not. The Scala compiler optimized away the recursive call:

```
public Domain approximate(Domain);
Code:
0:  aload_0
1:  astore_2
2:  aload_0
3:  aload_1
4:  invokevirtual #24; //Method isGoodEnough:(LDomain;)Z
7:  ifeq 12
```

⁷The benefit of avoiding `vars` was described in Step 3 on page 63.

```
10:  aload_1
11:  areturn
12:  aload_0
13:  aload_1
14:  invokevirtual #27; //Method improve:(LDomain;)LDomain;
17:  astore_1
18:  goto  2
```

The moral is that you should not shy away from using recursive algorithms to solve your problem. Often, a recursive solution is more elegant and concise than a loop-based one. If the solution is tail recursive, there won't be any runtime overhead to be paid.

Tracing tail-recursive functions

One consequence to watch out for is that a tail-recursive function will not build a new stack-frame for each call; all calls will execute in a single stack-frame. This may surprise a programmer inspecting a stack-trace of a program that failed. For instance, consider the function `boom`, shown next, which calls itself recursively a number of times and then crashes:

```
scala> def boom(x: Int): Int =
|     if (x == 0) throw new Exception("boom!")
|     else boom(x - 1) + 1
boom: (Int)Int
```

This function is *not* tail-recursive because it still performs a `(+1)` operation after the recursive call. Here's what you'll get when you run it:

```
scala> boom(5)
java.lang.Exception: boom!
    at .boom(<console>:5)
    at .boom(<console>:6)
    at .boom(<console>:6)
    at .boom(<console>:6)
    at .boom(<console>:6)
    at .boom(<console>:6)
    at .<init>(<console>:6)
```

...

You see one entry per recursive call in the stack-trace, as expected. If you now modify `boom` so that it does become tail-recursive:

```
scala> def bang(x: Int): Int =  
|   if (x == 0) throw new Exception("bang!")  
|   else bang(x - 1)  
bang: (Int)Int
```

You'll get:

```
scala> bang(5)  
java.lang.Exception: bang!  
at .bang(<console>:5)  
at .<init>(<console>:6)  
...
```

This time, you see only a single stack-frame for `bang`. You might think that `bang` crashed before it called itself, but this is not the case. If you think you might be confused by tail-call optimizations when looking at a stack-trace, you can turn them off by giving a

`-g:notc`

argument to the `scala` shell or to the `scalac` compiler. With that option specified, you will get a longer stack trace:

```
scala> bang(5)  
java.lang.Exception: bang!  
at .bang(<console>:5)  
at .bang(<console>:5)  
at .bang(<console>:5)  
at .bang(<console>:5)  
at .bang(<console>:5)  
at .bang(<console>:5)  
at .<init>(<console>:6)  
...
```

Limits of tail recursion

The use of tail recursion in Scala is fairly limited, because the JVM instruction set makes implementing more advanced forms of tail recursion very difficult. In fact, only directly recursive calls to a function are optimized. If the recursion is indirect, as in the following example of two mutually recursive functions, no optimization is possible:

```
def isEven(x: Int): Boolean =  
  if (x == 0) true else isOdd(x - 1)  
def isOdd(x: Int): Boolean =  
  if (x == 0) false else isEven(x - 1)
```

You also won't get a tail-call optimization if the final call goes to a function value. Consider for instance the following recursive code:

```
val funValue = nestedFun _  
def nestedFun(x: Int) {  
  if (x != 0) { println(x); funValue(x - 1) }  
}
```

The `funValue` variable refers to a function value that essentially wraps `nestedFun`. When you apply the function value to an argument, it turns around and applies `nestedFun` to that same argument, and returns the result. You might hope, therefore, the Scala compiler would perform a tail-call optimization, but in this case it would not. Thus, tail-call optimization is limited to situations in which a method or nested function calls itself directly as its last operation, without going through a function value or some other intermediary. (If you don't fully understand tail recursion yet, see [Section 8.9](#)).

8.10 Conclusion

This chapter has shown you several new ways functions can be defined in Scala. You can nest them inside each other and you can use them as first-class values. You have also seen several lightweight methods to define a function value without giving it a name. Such function values are also called closures. They are very flexible building blocks for creating your own control structures.

Chapter 9

Control Abstraction

All functions are separated into common parts, which are the same in every invocation of the function, and non-common parts, which may vary from one function invocation to the next. The common parts are in the body of the function, while the non-common parts must be supplied via arguments. When you use a function value as an argument, the non-common part of the algorithm is itself some other algorithm! At each invocation of such a function, you can pass in a different function value as an argument, and the invoked function will, at times of its choosing, invoke the passed function value. These *higher-order functions*—functions that take functions as parameters—give you extra opportunities to condense and simplify code.

9.1 Reducing code duplication

One benefit of higher-order functions is they enable you to create control abstractions that allow you to reduce code duplication. For example, suppose you are writing a file browser, and you want to provide an API that allows users to search for files matching some criterion. First, you add a facility to search for files whose names end in a particular string. This would enable your users to find, for example, all files with a “.scala” extension. You could provide such an API by defining a public `filesEnding` method inside a singleton object like this:

```
object FileMatcher {  
    private def filesHere = (new java.io.File(".")).listFiles
```

```
def filesEnding(query: String) =  
    for (file <- filesHere; if file.getName.endsWith(query))  
        yield file  
}
```

The `filesEnding` method obtains the list of all files in the current directory using the private helper method `filesHere`, then filters them based on whether each file name ends with the user-specified query. Given `filesHere` is private, the `filesEnding` method is the only accessible method defined in `FilesMatcher`, the API you provide to your users.

So far so good, and there is no repeated code yet. Later on, though, you decide to let people search based on any part of the file name. This is good for when your users cannot remember if they named a file `phb-important.doc`, `stupid-phb-report.doc`, `may2003salesdoc.phb`, or something entirely different, but they think that “phb” appears in the name somewhere. You go back to work and add this function to your `FileMatcher` API:

```
def filesContaining(query: String) =  
    for (file <- filesHere; if file.getName.contains(query))  
        yield file
```

This function works just like `filesEnding`. It searches `filesHere`, checks the name, and returns the file if the name matches. The only difference is that this function uses `contains` instead of `endsWith`.

The months go by, and the program becomes more successful. Eventually, you give in to the requests of a few power users who want to search based on regular expressions. These sloppy guys have immense directories with thousands of files, and they would like to do things like find all “pdf” files that have “oopsla” in the title somewhere. To support them, you write this function:

```
def filesRegex(query: String) =  
    for (file <- filesHere; if file.getName.matches(query))  
        yield file
```

Experienced programmers will notice all of this repetition and wonder if it can be factored into a common helper function. Doing it the obvious way does not work, however. You would like to be able to do the following:

```
def filesMatching(query: String, method) =  
    for (file <- filesHere; if file.getName.method(query))  
        yield file
```

This approach would work in a dynamically-typed language, but Scala is statically typed and therefore does not allow code construction like this. So what do you do?

Function values provide an answer. While you cannot pass around a method name as a value, you can get the same effect by passing around a function value that calls the method for you. In this case, you could add a matcher parameter to the method whose sole purpose is to check a file name against a query.

```
def filesMatching(  
    query: String,  
    matcher: (String, String) => Boolean  
) =  
    for (file <- filesHere; if matcher(file.getName, query))  
        yield file
```

In this version of the method, the `if` clause now uses `matcher` to check the file name against the query. Precisely what this check does depends on what is specified as the matcher. Second, look at the type of `matcher` itself. It is a function, and thus has a `=>` in the type. This function takes two string arguments—the file name and the query—and returns a boolean, so its full type signature is `(String, String) => Boolean`.

Given this new `filesMatching` helper method, the specific cases can now be simplified to call the helper method for most of the work:

```
def filesEnding(query: String) =  
    filesMatching(query, _.endsWith(_))  
def filesContaining(query: String) =  
    filesMatching(query, _.contains(_))  
def filesRegex(query: String) =  
    filesMatching(query, _.matches(_))
```

The function literals shown in this example use the placeholder syntax, introduced in the previous chapter, which may not as yet feel very natural to

you. Thus, here's a clarification of how placeholders are used in this example. The function literal `_.endsWith(_)`, used in the `filesEnding` method, means the same thing as:

```
(fileName: String, query: String) => fileName.endsWith(query)
```

Because `filesMatching` takes a function that takes two `String` arguments, however, you need not specify the types of the arguments. Thus you could also write `(fileName, query) => fileName.endsWith(query)`. Since the parameters are each used only once in the body of the function, and since the first parameter, `fileName`, is used first in the body, and the second parameter, `query`, is used second, you can use the placeholder syntax: `_.endsWith(_)`. The first underscore is a placeholder for the first parameter, the file name, and the second underscore a placeholder for the second parameter, the query string.

This code is already simplified, but it can actually be even shorter. Notice that the query gets passed to `filesMatching`, but `filesMatching` does nothing with the query except to pass it back to the passed `matcher` function. This passing back and forth is unnecessary, because the caller already knew the query to begin with! You might as well simply remove the `query` parameter from `filesMatching` and `matcher`, thus simplifying the code to the following:

```
object FileMatcher {  
    private def filesHere = (new java.io.File(".")).listFiles  
    private def filesMatching(matcher: String => Boolean) =  
        for (file <- filesHere; if matcher(file.getName))  
            yield file  
    def filesEnding(query: String) =  
        filesMatching(_.endsWith(query))  
    def filesContaining(query: String) =  
        filesMatching(_.contains(query))  
    def filesRegex(query: String) =  
        filesMatching(_.matches(query))  
}
```

This example demonstrates the way in which first-class functions can help you eliminate code duplication where it would be very difficult to do so without them. In Java, for example, you could create an interface containing a method that takes one `String` and returns a `Boolean`, then create and pass anonymous inner class instances that implement this interface to `filesMatching`. Although this approach would remove the code duplication you are trying to eliminate, it would at the same time add as much or more new code. Thus the benefit is not worth the cost, and often, many programmers would simply opt to live with the duplication.

Moreover, this example demonstrates how closures can help you reduce code duplication. The function literals used in the previous example, such as `_.endsWith(_)` and `_.contains(_)`, are instantiated at runtime into function values that are *not* closures, because they don't capture any free variables. Both variables used in the expression, `_.endsWith(_)`, for example, are represented by underscores, which means they are taken from arguments to the function. Thus, `_.endsWith(_)` uses two bound variables, and no free variables. By contrast, the function literal `_.endsWith(query)`, used in the most recent example, contains one bound variable, the argument represented by the underscore, and one free variable named `query`. It is only because Scala supports closures that you were able to remove the `query` parameter from `filesMatching` in the most recent example, thereby simplifying the code even further.

9.2 Simplifying client code

In addition to helping you reduce code duplication as you implement an API, as demonstrated in the previous example, you can provide higher-order functions in an API itself to make client code more concise. A good example of this is the special-purpose looping methods provided by Scala's collection types,¹ many of which are listed in [Table 3.1](#) in [Chapter 3](#). Scala provides while loops and for expressions as built-in control structures that can help you with all of your looping needs. The while loop supports an imperative style, and the for expression a functional style, of looping code.

¹All of these special-purpose looping methods are defined in trait `Iterable`, which is extended by most collection types, including `List`, `Set`, `Array`, and `Map`.

In addition to the built-in looping constructs, however, Scala’s collections API provides many higher-order functions—functions that take functions as arguments—for common looping needs. More often than not, any non-trivial loop can be written as a call to one of these methods, thus shortening your code.

One example is `exists`, a method that determines whether a passed value is contained in a collection. You could of course search for an element by having a `var` initialized to `false`, looping through the collection checking each item, and setting the `var` to `true` if you find what you are looking for. Here’s a method that uses this approach to determine whether a passed `List` contains a `0`:

```
def containsZero(nums: List[Int]): Boolean = {  
    var exists = false  
    for (num <- nums)  
        if (num == 0)  
            exists = true  
    exists  
}
```

If you define this method in the interpreter, you can call it like this:

```
scala> containsZero(List(1, 2, 3, 4))  
res37: Boolean = false  
  
scala> containsZero(List(1, 2, 0, 4))  
res38: Boolean = true
```

A more concise way to define the method, though, is by calling the higher-order function `exists` on the passed `List`, like this:

```
def containsZero(nums: List[Int]) = nums.exists(_ == 0)
```

This version of `containsZero` yields the same results as the previous:

```
scala> containsZero(Nil)  
res43: Boolean = false  
  
scala> containsZero(List(0, 0, 0))  
res44: Boolean = true
```

The `exists` method represents a control abstraction—a special-purpose looping construct—provided by the Scala library (as opposed to being built into the Scala language like `while` or `for`). In the previous section, the higher-order function, `filesMatching`, reduces code duplication in the implementation of the object `FileMatcher`. The `exists` method provides a similar benefit, but because `exists` is public in Scala’s collections API, the code duplication it reduces is client code of that API. If `exists` didn’t exist, and you wanted to write a `containsOne` method, you might write it like this:

```
def containsOne(nums: List[Int]): Boolean = {
    var exists = false
    for (num <- nums)
        if (num == 1)
            exists = true
    exists
}
```

If you compare the body of `containsZero` with that of `containsOne`, you’ll find that everything is repeated except that a 0 is changed to a 1. Using `exists`, you could write this instead:

```
def containsOne(nums: List[Int]) = nums.exists(_ == 1)
```

The body of the code in this version is again identical to the body of the corresponding `containsZero` method (the version that uses `exists`), except the 0 is changed to a 1. Yet the amount of code duplication is much smaller because all of the looping infrastructure is factored out into the `exists` method itself.

9.3 Currying

In Chapter 1, we said that Scala allows you to create new control abstractions that “feel like native language support.” Although the examples you’ve seen so far are indeed control abstractions, it is unlikely anyone would mistake them for native language support. To understand how to make control abstractions that feel more like language extensions, you first need to understand the functional programming technique called *currying*.

A curried function is applied to multiple argument lists, instead of just one. Here's a regular, not curried function, which adds two Int parameters, x and y:

```
scala> def plainOldSum(x: Int, y: Int) = x + y
plainOldSum: (Int,Int)Int
```

You invoke it in the usual way:

```
scala> plainOldSum(1, 2)
res46: Int = 3
```

By contrast, here's a similar function that's curried. Instead of one list of two Int parameters, you apply this function to two lists of one Int parameter each:

```
scala> def curriedSum(x: Int)(y: Int) = x + y
curriedSum: (Int)(Int)Int
```

Here's how you invoke it:

```
scala> curriedSum(1)(2)
res47: Int = 3
```

What's happening here is that when you invoke `curriedSum`, you actually get two traditional function invocations back to back. The first function invocation takes a single Int parameter named x, and returns a function value for the second function. This second function takes the Int parameter y. Here's a function named `first` that does in spirit what the first traditional function invocation of `curriedSum` would do:

```
scala> def first(x: Int) = (y: Int) => x + y
first: (Int)(Int) => Int
```

Applying 1 to the first function—in other words, invoking the first function and passing in 1—yields the second function:

```
scala> val second = first(1)
second: (Int) => Int = <function>
```

Applying 2 to the second function yields the result:

```
scala> second(2)
res4: Int = 3
```

Although these `first` and `second` functions are just an illustration of the currying process having nothing to do with the `curriedSum` function, there is a way to get an actual reference to `curriedSum`'s “second” function. You can use the placeholder notation to use `curriedSum` in a partially applied function expression, like this:

```
scala> val onePlus = curriedSum(1)_
onePlus: (Int) => Int = <function>
```

The underscore in `curriedSum(1)_` is a placeholder for the second parameter list.² The result is a reference to a function that, when invoked, adds one to its sole `Int` argument and returns the result:

```
scala> onePlus(2)
res5: Int = 3
```

And here's how you'd get a function that adds two to its sole `Int` argument:

```
scala> val twoPlus = curriedSum(2)_
twoPlus: (Int) => Int = <function>
scala> twoPlus(2)
res6: Int = 4
```

Now you may think this is a lot of trouble just to figure out that $2 + 2$ yields 4, and you may be right, but currying does have a few uses. One of the uses, in fact, is in helping you make control abstractions that feel like native language support. This use case is described in the next section.

9.4 Writing new control structures

In languages with first-class functions, you can effectively make new control structures even though the syntax of the language is fixed. All you need to do is create methods that take functions as arguments.

²In the previous chapter, when the placeholder notation was used on traditional methods, like `println_`, you had to leave a space between the name and the underscore. In this case you don't, because whereas `println_` is a legal identifier in Scala, `curriedSum(1)_` is not.

For example, here is the “twice” control structure, which repeats an operation two times and returns the result:

```
scala> def twice(op: Double => Double, x: Double) = op(op(x))
twice: ((Double) => Double,Double)Double
scala> twice(_ + 1, 5)
res24: Double = 7.0
```

The type of `op` in this example is `Double => Double`, which means it is a function that takes one `Double` as an argument and returns another `Double`.

Any time you find a control pattern repeated in multiple parts of your code, you should think about implementing it as a new control structure. For example, one common pattern is “open a resource, operate on it, and then close the resource.” This is sometimes called the *loan pattern*. You can capture this pattern using a method like the following:

```
def withPrintWriter(file: File, op: PrintWriter => Unit) {
    val writer = new java.io.PrintWriter(file)
    try {
        op(writer)
    } finally {
        writer.close()
    }
}
```

Given such a method, you can use it as follows:

```
withPrintWriter(
    new File("date.txt"),
    writer => writer.println(new java.util.Date)
)
```

The advantage of using this method is that it’s the `withPrintWriter` method instead of the user code that assures the file is closed at the end. So it’s impossible to forget to close the file.

One way in which you can make the client code look a bit more like a built-in control structure is to use curly braces instead of parentheses to surround the argument list. In any method invocation in Scala, you can opt

to use curly braces to surround the argument list instead of parentheses. For example, instead of:

```
scala> println("Hello, world!")
Hello, world!
```

You could write:

```
scala> println { "Hello, world!" }
Hello, world!
```

In the second example, you used curly braces instead of parentheses to surround the arguments to `println`. Using this same technique to call function `withPrintWriter` would give you code that looks like:

```
withPrintWriter {
    new File("date.txt"),
    writer => writer.println(new java.util.Date)
}
```

Although in the previous example, `withPrintWriter` is starting to look more like a built-in control construct, you can do better. Because the function passed to `withPrintWriter` is the last argument in the list, you can use currying to pull the first argument, the `File`, outside the curly braces. Here's how you'd need to define `withPrintWriter`:

```
def withPrintWriter(file: File)(op: PrintWriter => Unit) {
    val writer = new java.util.PrintWriter(file)
    try {
        op(writer)
    } finally {
        writer.close()
    }
}
```

Unless you look carefully, you may think this version of `withPrintWriter` is identical to the one shown previously, but it's not. The new version differs from the old one only in that there are now two parameter lists with one parameter each instead of one parameter list with two parameters. Look between the two parameters. In the previous version of `withPrintWriter`,

shown on page 199, you'll see ...`File`, `operation`... But in this version, you'll see ...`File`)(`operation`... Given the above definition, you can call the method with a more pleasing syntax:

```
val file = new File("date.txt")
withPrintWriter(file) {
    writer => writer.println(new java.util.Date)
}
```

In this example, the first argument list, which contains one `File` argument, is written surrounded by parentheses. The second argument list, which contains one function argument, is surrounded by curly braces.

9.5 By-name parameters

The `withPrintWriter` method shown in the previous section differs from built-in control structures of the language, such as `if` and `while`, in that the code between the curly braces takes an argument. The `withPrintWriter` method requires one argument of type `PrintWriter`. This argument shows up as the “`writer =>`” in:

```
withPrintWriter(file) {
    writer => writer.println(new java.util.Date)
}
```

What if you want to implement something more like `if` or `while`, however, where there is no value to pass into the code between the curly braces of the control structure? To help with such situations, Scala provides [by-name parameters](#).

As a concrete example, suppose you want to implement an assertion construct called `myAssert`.³ The `myAssert` function will take a function value as input and consult a flag to decide what to do. If the flag is set, `myAssert` will invoke the passed function and verify that it returns `true`. If the flag is turned off, `myAssert` will quietly do nothing at all.

Without using by-name parameters, you could write `myAssert` like this:

³You'll call this `myAssert`, not `assert`, because Scala provides an `assert` of its own, which will be described in [Section 10.4](#) on page 209.

```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
    if (assertionsEnabled && !predicate())
        throw new AssertionException
```

The definition is fine, but using it is a little bit awkward:

```
myAssert(() => 5 > 3)
```

You would really prefer to leave out the empty parameter list and `=>` symbol in the function literal and write the code like this:

```
myAssert(5 > 3) // Won't work, because missing () =>
```

By-name parameters exist precisely so that you can do this. To make a by-name parameter, you give the parameter a type starting with `=>` instead of `() =>`. For example, you could change `myAssert`'s predicate parameter into a by-name parameter by changing its type, “`() => Boolean`” into “`=> Boolean`.” Here’s how that would look:

```
def byNameAssert(predicate: => Boolean) =
    if (assertionsEnabled && !predicate)
        throw new AssertionException
```

Now you can leave out the empty parameter in the property you want to assert. The result is that using `byNameAssert` looks exactly like using a built-in control structure:

```
byNameAssert(5 > 3)
```

A by-name type, in which the empty parameter list, `()`, is left out, is only allowed for parameters. There is no such thing as a by-name variable or a by-name field.

Now, you may be wondering why you couldn’t simply write `myAssert` using a plain old `Boolean` for the type of its parameter, like this:

```
def boolAssert(predicate: Boolean) =
    if (assertionsEnabled && !predicate)
        throw new AssertionException
```

It turns out you can, and when you use it, the code also looks quite natural:

```
boolAssert3(5 > 3)
```

Nevertheless, one difference exists between these two approaches that is useful to note. Because the type of `boolAssert`'s parameter is `Boolean`, the expression inside the parentheses in `boolAssert(5 > 3)` is evaluated *before* the call to `boolAssert`. The expression `5 > 3` yields `true`, which is passed to `boolAssert`. By contrast, because the type of `byNameAssert`'s parameter is `=> Boolean`, the expression inside the parentheses in `byNameAssert2(5 > 3)` is *not* evaluated before the call to `byNameAssert`. Instead a function value will be created whose `apply` method will evaluate `5 > 3`, and this function value will be passed to `byNameAssert`.

The difference between the two approaches, therefore, is that if assertions are disabled, you'll see any side effects that the expression inside the parentheses may have in `boolAssert`, but not in `byNameAssert`. For example, if assertions are disabled, attempting to assert on “`throw new Exception`” will yield an exception in `boolAssert`'s case:

```
scala> var assertionsEnabled = false
assertionsEnabled: Boolean = false

scala> boolAssert(throw new Exception)
java.lang.Exception
at .<init>(<console>:6)
at .<clinit>(<console>)
at RequestResult$.<init>(<console>:3)
at RequestResult$.<clinit>(<console>)
at RequestResult$result(<console>)...
```

But attempting to assert on same code in `byNameAssert`'s case will *not* yield an exception:

```
scala> byNameAssert(throw new Exception)
```

In this case, the code that throws the exception is wrapped in a function value that's passed to `byNameAssert`. Because assertions are disabled, `byNameAssert` never invokes the function, so the exception is never thrown.

9.6 Conclusion

This chapter has shown you several new ways functions can be defined in Scala. You can nest them inside each other and you can use them as first-class values. You have also seen several lightweight methods to define a function value without giving it a name. Such function values are also called closures. They are very flexible building blocks for creating your own control structures. You have seen two syntactic tweaks that make operating on closures more pleasant: currying and call-by-name parameters. With the help of these, you can write control structures that look as if they were built-in language constructs.

Chapter 10

Composition and Inheritance

This chapter discusses more of Scala’s support for object-oriented programming. The topics are not as fundamental as those in [Chapter 4](#), but they will frequently arise as you program in Scala.

10.1 Introduction

As a running example, this chapter presents a simple library for building and rendering two-dimensional layout elements. Each element represents a rectangle filled with text. Elements can be composed above or beside each other. For instance, assume you are given a method with the following signature which creates a layout element containing a string:

```
elem(s: String): Element
```

Then the expression below would construct a larger element consisting of two columns, each with a height of two.

```
val column1 = elem("hello") above elem("***")
val column2 = elem("***") above elem("world")
column1 beside column2
```

Printing the result of this expression would give:

```
hello ***
*** world
```

10.2 Abstract classes

The first task covered in this chapter is how to define the type `Element` of layout elements. What members should layout elements have? Since elements are two dimensional rectangles of characters, it makes sense to include a member `contents`, which refers to the contents of a layout element. The contents can be represented as an array of strings, where each string represents a line. Hence, the type of the result returned by `contents` should be `Array[String]`.

Then there should be methods `width` and `height` that provide the dimensions of the layout element.

Finally, there should be methods `above` and `beside` for forming new elements by placing an element above and beside another, respectively.

All five methods together are bundled in a class `Element`. The outline of this class is as follows:

```
abstract class Element {  
    def contents: Array[String]  
    def width: Int = ...  
    def height: Int = ...  
    def above(that: Element): Element = ...  
    def beside(that: Element): Element = ...  
}
```

The `Element` class declares five methods: `contents`, `width`, `height`, `above`, and `beside`. The implementations of the last four of these methods are left out here; they will be given below. The first method, `contents`, does not have an implementation. In other words, the method is an *abstract* member of class `Element`. A class with abstract members must itself be declared abstract; this is done by writing an *abstract* modifier in front of the `class` keyword:

```
abstract class Element ...
```

The *abstract* modifier in front of a class signifies that the class may have abstract members which do not have an implementation. Therefore, it is not permitted to create an object of an abstract class. If you try to write

```
scala> new Element
```

You should get something like this:

```
<console>:5: error: class Element is abstract; cannot be instantiated
  val res6 = new Element
                           ^
```

You will see below how to create subclasses of class `Element` which fill in the missing definition and which can be instantiated.

Note that the methods in class `Elements` do not carry an `abstract` modifier. A method is abstract if it does not have an implementation (*i.e.*, no equals sign or body). Unlike Java, no abstract modifier is necessary (nor allowed).

Methods which do have an implementation are called *concrete*. Another bit of terminology distinguishes between *declarations* and *definitions*. We say that class `Element` *defines* the concrete method `height`, whereas it *declares* the abstract method `contents`.

10.3 The Uniform Access Principle

The next thing to turn to is the implementation of the concrete methods in class `Element`. Here are the first two:

```
def width: Int =
  if (height == 0) 0 else contents(0).length
def height: Int = contents.length
```

The `height` method returns the number of lines in `contents`. The `width` method returns the length of the first line, or, if there are no lines in the element, zero. (This means you cannot define an element with a height of zero and a non-zero width.)

Note that neither method has a parameter list, not even an empty one. Parameterless methods such as `width` or `height` are quite common in Scala. The recommendation is to use a parameterless method whenever there are no arguments and the method neither changes nor depends on mutable state.

This convention supports the *uniform access principle*, which says that client code should not be affected by a decision to implement some attribute as a field or as a method. For instance, you could have chosen to implement

width and height as fields instead of methods, simply by changing the def in their definition to a val:

```
val width =  
    if (contents.length == 0) 0 else contents(0).length  
val height = contents.length
```

The two pairs of definitions are completely equivalent from a client's point of view. The only difference is that field accesses might be slightly faster than method invocations, because the field values are pre-computed when the class is initialized, instead of being computed on each method call. On the other hand, the fields require extra memory space in every Element object. So it depends on the usage profile of a class whether an attribute is better represented as a field or as an access function, and that usage profile might change over time. The point is that clients of the Element class should not be affected when its internal implementation changes.

In particular, a client of class Element should not need to be rewritten if a field of that class gets changed into an access function as long as the access function is *pure*, i.e., it does not have any side effects and it does not depend on mutable state. It should not need to care either way.

So far so good. But there's still a slight complication that has to do with the way Java handles things. The problem is that Java does not implement the uniform access principle. So it's `string.length()` in Java, not `string.length` (even though it's `array.length`, not `array.length()`). Needless to say, this is very confusing.

To bridge that gap, Scala is very liberal when it comes to mixing parameterless methods and methods with '()' parameters. In particular, you can override a '()' -method with a parameterless method and *vice versa*. You can also leave out an empty argument list '()' from a method call. For instance, the following two lines are both legal in Scala:

```
Array(1, 2, 3).toString  
"abc".length
```

In principle it's possible to leave out all empty parameter lists in Scala code. However, it is recommended to still write an empty parameter list when the invoked method reads or writes reassignable variables (vars), either directly or indirectly by using mutable objects. That way, the parameter list acts as

a visual clue that some interesting computation is triggered by the call. For instance:

```
myString.length // no () because no side-effect  
out.println() // better not to drop the ()
```

To summarize, it is encouraged style in Scala to write parameterless pure member functions without ‘()’. On the other hand, you should never write a function that has side-effects without a ‘()’, because then it would look like nothing is evaluated. So you would be surprised to see the side effect. This applies to both function definitions and applications. As described in [Step 2](#), one way to think about whether a function has side effects is if the function performs a conceptual “operation,” use the parentheses, but if it merely provides access to a conceptual ‘property,’ leave the parentheses off.

10.4 Assertions and assumptions

Note that the `width` method gives a correct result only if all lines in the array have the same length. You can state this property as a set of assertions in class `Element`:

```
for (line <- contents)  
    assert(line.length == width, "element is not rectangular")
```

Assertions in Scala are written as calls of a predefined method `assert`.¹ The expression `assert(condition)` throws an `AssertionError` if condition does not hold. There’s also a two argument version: `assert(condition, explanation)` tests condition, and, if it does not hold, throws an `AssertionError` which contains the given `String` explanation.

A failing assertion always indicates that some code is incorrect. The incorrect code is not always the code that contains the assertion, however. For instance, consider the following function which returns the inverse of its floating point argument.

```
def inverse(x: Double) = 1 / x
```

¹The `assert` method is defined in the `Predef` singleton object, whose members are automatically imported into every Scala source file.

This function will throw an `ArithmeticeException` if `x` is zero. You could specify the legal argument restriction directly, using an assertion:

```
def inverse(x: Double) = { assert(x != 0); 1 / x }
```

In that case, `inverse(0)` would throw an `AssertionError` instead of an `ArithmeticeException`. But in a sense, this is still not right, because it is not the `inverse` method that should be blamed for this fault but its caller, which passed the zero argument. In situations like this it is better to `assume` that the argument is non-zero, instead of asserting it. Scala defines analogs of both variants of `assert` which are called `assume`. Instead of throwing an `AssertionError` like `assert` does, an `assume` throws an `IllegalArgumentException` when it fails. So `assume` clearly blames the caller of the method for a failure, not the method itself. Here's `inverse`, again, using an `assume`:

```
def inverse(x: Double) = { assume(x != 0, "must be non-zero"); 1/x }
```

Now if you try to execute `inverse(0)`, you should get:

```
java.lang.IllegalArgumentException: assumption failed: must be non-zero
```

Just like Java assertions, assertions and assumptions can be enabled and disabled using the JVM's `-ea` and `-da` command-line flags.

10.5 Subclasses

For the implementation of the remaining methods above and beside you need a way to create new element objects. You have already seen that “new Element” cannot be used for this because class `Element` is abstract. You need to create a subclass which extends `Element` and which implements the abstract `contents` method. Here's a possible way to do this:

```
class ArrayElement(contents: Array[String]) extends Element {
    def contents: Array[String] = contents
}
```

Class `ArrayElement` is defined to *extend* class `Element`. Just like in Java, you use an `extends` clause

```
extends Element
```

after the class name to express this. Such an extends clause has two effects: It makes class `ArrayElement` *inherit* all non-private members from class `Element`. And it makes the type `ArrayElement` a *subtype* of the type `Element`. If you leave out an extends clause, the Scala compiler implicitly assumes your class extends from `scala.AnyRef`, which on the Java platform is the same as class `java.lang.Object`.

Given the extends clause above, class `ArrayElement` is called a *subclass* of class `Element`. Conversely, `Element` is a *superclass* of `ArrayElement`.

Inheritance means that all members of the superclass are also members of the subclass, with two exceptions. First, private members of the superclass are not inherited in a subclass. Second, a member of a superclass is not inherited if a member with the same name and parameters is already implemented in the subclass. In that case we say the member of the subclass *overrides* the member of the superclass. If the member in the subclass is concrete and the member of the superclass is abstract, we also say that the concrete member *implements* the abstract one.

For instance, the `contents` method in `ArrayElement` overrides (or, alternatively: implements) the abstract method `contents` in class `Element`. On the other hand, class `ArrayElement` inherits the `width`, `height`, `above`, and `beside` methods from class `Element`. For instance, given an `ArrayElement ae`, you can query its `width` using `ae.width`, just as if `width` has been defined in class `ArrayElement`.

Subtyping means that a value of the subclass can be used wherever a value of the superclass is required. For instance, in the value definition

```
val e: Element = new ArrayElement(...)
```

The variable `e` is defined to be of type `Element`, so its initializing value should also be an `Element`. In fact the type of the initializing value is `ArrayElement`. This is OK, because class `ArrayElement` extends class `Element` so the type `ArrayElement` is compatible with the type `Element`.

10.6 Two name spaces, not four

The uniform access principle is just one aspect where Scala treats fields and methods more uniformly than Java. Another difference is that, in Scala,

fields and methods belong to the same name space. This makes it possible that a field may override a parameterless method. For instance, you could change the implementation of `contents` in class `ArrayElement` from a method to a field without having to modify the abstract method definition of `contents` in class `Element`:

```
class ArrayElement(contents: Array[String]) extends Element {  
    val contents: Array[String] = contents  
}
```

The field `contents` (defined with a `val`) in `ArrayElement` is a perfectly good implementation of the parameterless method `contents` (declared with a `def`) in class `Element`.

On the other hand, in Scala it is forbidden to define a field and method with the same name in the same class. This contrasts with Java, which allows you to declare like-named fields and methods in the same class. For example, this Java class would compile just fine:

```
// This is Java  
class CompilesFine {  
    private int f = 0;  
    public int f() {  
        return 1;  
    }  
}
```

But the corresponding Scala class would not compile:

```
class WontCompile {  
    private var f = 0 // Won't compile, because a field  
    def f = 1         // and method have the same name  
}
```

Generally, Scala has just two name spaces for definitions in place of Java's four. Java's four namespaces are:

- fields
- methods

- types
- packages

By contrast, Scala's two namespaces are:

- values (fields, methods, and packages)
- types (class and trait names)

The reason Scala places fields and methods into the same namespace is precisely so you can override a parameterless method with a `val`, something you can't do with Java.²

10.7 Class parameter fields

Consider again the definition of class `ArrayElement` above. It has a parameter `conts` whose sole purpose is to be copied into the `contents` field. The name `conts` of the parameter was chosen just so that it would look similar to the field name `contents` without actually clashing with it. This is a “code smell”, a sure sign that there is some unnecessary redundancy and repetition in your code.

You can avoid the code smell by combining the parameter and the field in a single class parameter field definition, like this:

```
class ArrayElement(val contents: Array[String]) extends Element {}
```

Note that now the `contents` parameter is prefixed by `val`. This is a shorthand which defines at the same time a parameter and a field with the same name. Concretely, class `ArrayElement` now has an (unre assignable) field `contents` which can be accessed from outside the class. The field is initialized with the value of the parameter. It's as if the class had been written as follows, where `x123` is an arbitrary fresh name for the parameter:

²The reason that packages share the same name space as fields and methods in Scala is to enable you to import packages in addition to just importing the names of types, and the fields and methods of singleton objects. This is also something you can't do in Java. It will be described in [Section 13.2 on page 283](#).

```
class ArrayElement(x123: Array[String]) extends Element {  
    val contents: Array[String] = x123  
}
```

One can also prefix a class parameter with var, then the corresponding field would be reassignable. Finally, it is possible to add modifiers such as private, protected, or override to these *parametric fields*, just as one can do for any other class member. Consider for instance the following class definition

```
class C	override val x: Int, private var y: String, z: Boolean) {}
```

This is a shorthand for the following definition of a class C with an overriding member x and a private member y.

```
class C(xParam: Int, yParam: String, z: Boolean) {  
    override val x = xParam  
    private var y = yParam  
}
```

Both members are initialized from the corresponding parameters.³

10.8 More method implementations

Now that ArrayElement is defined, the next step is to implement method above in class Element. Putting one element above another means concatenating the two contents values of the elements. So a first draft of method above could look like this:

```
def above(that: Element): Element =  
    new ArrayElement(this.contents ++ that.contents)
```

The ‘++’ operation concatenates two arrays. Arrays in Scala are represented as Java arrays, but support many more methods. Specifically, arrays in Scala inherit from a class `scala.Seq`, which represents sequence-like structures and contains a number of methods for accessing and transforming sequences.

³The parameter names xParam and yParam were chosen arbitrarily. The important thing was that they not clash with any other name in scope.

Some other array methods will be explained in this chapter, and a comprehensive list of all methods will be given in [Chapter 15](#).

In fact, the code shown previously is not quite sufficient, because it does not permit you to put elements of different widths on top of each other. For example, evaluating the expression

```
new ArrayElement(Array("hello")) above  
new ArrayElement(Array("world!"))
```

should give an assertion error because the second line in the combined element is longer than the first. To correct this problem, you need to apply a method that adapts the width of the two elements so that they are equal. Adaptation means placing an element at the center of a wider box, padding it with spaces left and right. We'll design a method `widen` for performing this task.

10.9 Private helper methods

Here's a first implementation of `widen` (you'll see a more elegant one below).

```
private def widen(w: Int): Element =  
  if (w <= width) this  
  else {  
    val lpad = (w - width) / 2  
    val rpad = w - (lpad + width)  
    new ArrayElement(  
      for (line <- contents)  
        yield spaces(lpad) + line + spaces(rpad)  
    )  
  }
```

The `widen` method takes a target width `w` as parameter and returns an `Element`. If the current element width is already greater or equal to the target width, the element itself is returned. Otherwise, the method returns a new element where each line in `contents` is prefixed by `lpad` spaces and followed by `rpad` spaces.

Note that the `for` expression that performs this computation is used directly as an argument to the object creation `new ArrayElement`. This is

possible because every statement which is not a definition is an expression in Scala. So even for expressions count as expressions.

The widen method makes use of another method, spaces, which creates a string consisting of a given number of spaces. This one is defined as follows.

```
private def spaces(n: Int) = new String(Array.make(n, ' '))
```

This makes use of the `Array.make` method, defined in Scala's standard `Array` object. `Array.make` takes two arguments: a length and an array element value. It constructs an array of the given length with all elements initialized to the given array element value. So

```
Array.make(5, 'Z')
```

makes the array `Array('Z', 'Z', 'Z', 'Z', 'Z')`.

Using `widen`, the above method can now be implemented correctly as follows.

```
def above(that: Element): Element = {
    val this1 = this widen that.width
    val that1 = that widen this.width
    new ArrayElement(this1.contents ++ that1.contents)
}
```

10.10 Imperative or functional?

The next method to implement is `beside`. To put two elements beside each other, you create a new element in which every line results from concatenating corresponding lines of the two elements. As a first step, the elements must be adjusted so that they have the same height. This leads to the following design of method `beside`:

```
def beside(that: Element): Element = {
    val this1 = this heighten that.height
    val that1 = that heighten this.height
    val contents = new Array[String](this1.contents.length)
    for (i <- 0 until this1.contents.length)
```

```
contents(i) = this1.contents(i) + that1.contents(i)
new ArrayElement(contents)
}
```

The `beside` method first adjusts the two arguments to have the same height. It uses the `heighten` method for this, which is like the `widen`, except that it adjusts the height of a layout element instead of its width. The height-adjusted arguments are named `this1` and `that1`. After that, the method allocates a new array `contents` and fills it with the concatenation of the corresponding array elements in `this1.contents` and `that1.contents`. It finally produces a new `ArrayElement` containing `contents`.

The last four lines can alternatively be abbreviated to one expression:

```
new ArrayElement(
    for ((line1, line2) <- this1.contents zip that1.contents)
        yield line1 + line2
)
```

Here, the two arrays `this1.contents` and `that1.contents` are transformed into an array of pairs using the `zip` operator. The `zip` method picks corresponding elements in its two arguments and forms an array of pairs. For instance, this expression

```
Array(1, 2, 3) zip Array("a", "b", "c")
```

will evaluate to:

```
Array((1, "a"), (2, "b"))
```

If one of the two operand arrays is longer than the other, any remaining elements are dropped. In the expression above, the third element of the left operand does not form part of the result, because it does not have a corresponding element in the right operand.

The zipped array is then iterated over by a `for` expression. Here, the syntax `for ((line1, line2) <- ...)` allows you to name both elements of a pair in one *pattern*, *i.e.*, `line1` stands now for the first element of the pair, and `line2` stands of the second. Pattern matching is one of the major language innovations in Scala; there will be much more on this topic in Chapter 12.

The `for` expression has a `yield` part and therefore returns a result. The result is of the same kind as the expression iterated over, *i.e.*, it is an array. Each element of the array is the result of concatenating the corresponding lines `line1` and `line2`. So the end result is the same as in the first version of `beside`, but it is obtained in a purely functional way.

You still need a way to display elements. As usual, this is done by defining a `toString` method that returns an element formatted as a string. Here is its definition:

```
override def toString = contents mkString "\n"
```

The implementation of `toString` makes use of a `mkString` method which is defined for all sequences, including arrays. An expression like `arr mkString sep` returns a string consisting of all elements of the array `arr`. Each element is mapped to a string by calling its `toString` method. A separator string `sep` is inserted between consecutive element strings. So the expression `contents mkString "\n"` formats the `contents` array as a string, where every array element appears on a line by itself.

Note that `toString` does not carry an empty '()' parameter list. This follows the recommendations for the uniform access principle, because `toString` is a pure method that does not take any parameters.

10.11 Adding other subclasses

You now have a complete system consisting of two classes: An abstract class `Element` which is inherited by a concrete class `ArrayElement`. One might also envisage other ways to express an element. Think for instance of a layout element consisting of a single line which is given by a string. Another possibility would be a layout element of given width and height that is filled everywhere by some given character. One important aspect of object-oriented programming is that it makes it easy to extend a system with new data-variants. You can simply add further subclasses that extend a common parent class.

For instance, here are two classes for single-line elements and uniform rectangles:

```
class LineElement(s: String) extends Element {
```

```
def contents = Array(s)
override def width = s.length
override def height = 1
}

class UniformElement(
    ch: Char,
    override val width: Int,
    override val height: Int
) extends Element {
    private val line = new String(Array.make(width, ch))
    def contents = Array.make(height, line)
}
```

With the new subclasses, the inheritance hierarchy for layout elements now looks as in the left part of [Figure 10.1](#). Other hierarchies are also possible. For instance, class `LineElement` could inherit from `ArrayElement` instead of being a direct subclass of `Element`, as is shown in the right part of [Figure 10.1](#). Here's a modified version `LineElement2` which implements this hierarchy:

```
class LineElement2(s: String) extends ArrayElement(Array(s)) {
    override def width = s.length
    override def height = 1
}
```

Since class `LineElement2` inherits from the parameterized class `ArrayElement`, it needs to pass an argument to the primary constructor of its superclass. The argument simply follows the name of the superclass in parentheses, as in

```
extends ArrayElement(Array(s))
```

10.12 Override modifiers and the fragile base class problem

Note that the definitions of `width` and `height` in these classes carry an `override` modifier. Previously, you have already seen this modifier in the

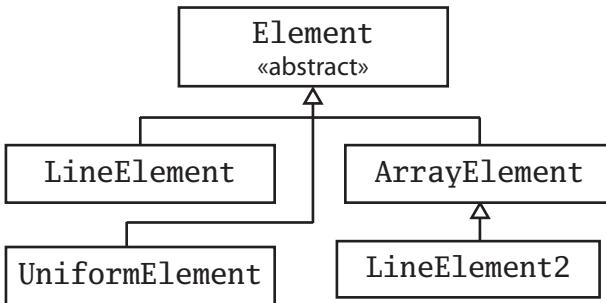


Figure 10.1: Class hierarchy of layout elements

definition of the `toString` method. Scala requires such a modifier for all members that override a concrete member in a parent class. The modifier is optional if a member implements some abstract member with the same name. The modifier is forbidden if a member does not override or implement some other member in a base class. Since `height` and `width` in class `LineElement` and `UniformElement` override concrete definitions in class `Element`, the new definitions need to be flagged with `override`.

This provides useful information for the compiler which helps avoid some hard-to-catch errors and makes system evolution safer. For instance, if you happen to misspell the method or give it a different parameter list, the compiler will respond with an error message:

```

$ scalac LayoutElement.scala
.../LayoutElement.scala:50:
error: method hight overrides nothing
      override def hight = 1
                           ^
one error found
  
```

The `override` convention is even more important when it comes to system evolution. Say you have defined a library of 2D drawing methods. You have made it publicly available and it is widely used. In the next version of the library you want to add to your base class `Shape` a new method

```
def hidden(): Boolean
```

The method is used by various drawing methods to determine whether a shape needs to be drawn. This could lead to a significant speedup, but you cannot do this without the risk of breaking client code. After all, a client could have defined a subclass of Shape with a different implementation of hidden. Say, the client's method actually makes the receiver object disappear instead of testing whether the object is hidden. Because the two versions of hidden override each other, your drawing methods would end up making objects disappear, which is certainly not what you want! These “accidental overrides” are at the root of what is called the “fragile base class” problem. The problem is that if you add new members to base classes in a class hierarchy, you risk breaking client code.

Scala cannot completely solve the fragile base class problem, but it improves on the situation by comparison to Java. If the drawing library and its clients are written in Scala, then the client's original implementation of hidden could not have an override modifier, because at the time there was no other method with that name. Once you add the hidden method to the second version of your shape class, a recompile of the client would give you an error like the following:

```
/src/examples/Shapes.scala:6: error: error overriding method hidden  
in class Shape of type ()Boolean;  
method hidden needs 'override' modifier  
def hidden(): boolean =  
^
```

That is, instead of wrong behavior you get a compile-time error, which is usually much preferable.

10.13 Factories

You now have a hierarchy of classes for layout elements. This hierarchy could be presented to a user “as is”. But you might also choose to hide the hierarchy in a factory object. A factory object contains methods that construct other objects. Clients would then use these factory methods for object construction rather than constructing the objects directly with new. An advantage of this approach is that object creation can be centralized and the details of how objects are represented with classes can be hidden.

The first task in constructing a factory for layout elements is to choose where the factory methods should be located. Should they be members of a singleton object or of a class? What should the containing object or class be called? There are many possibilities. A straightforward solution is to create a companion object of class `Element` and make this be the factory object for layout elements. That way, you need to expose only the class/object combo of `Element` to your clients, and you can hide the three implementation classes `ArrayElement`, `LineElement`, and `UniformElement`.

Here is a design of the `Element` object that follows this scheme:

```
object Element {  
    def elem(contents: Array[String]): Element =  
        new ArrayElement(contents)  
    def elem(chr: Char, width: Int, height: Int): Element =  
        new UniformElement(chr, width, height)  
    def elem(line: String): Element =  
        new LineElement(line)  
}
```

The `Element` contains three overloaded variants of an `elem` method. Each variant constructs a different kind of layout object.

10.14 Putting it all together

With the factory methods in `Element`, the subclasses `ArrayElement`, `LineElement` and `UniformElement` can now be private because they need no longer be accessed directly by clients. Some simplifications of class `Element` are also possible. First, direct object construction can now be replaced by a call to a factory method. Another simplification concerns the implementations of the adjustment methods `widen` and `heighten` (which was not yet shown). Instead of directly manipulating content arrays, they can also be implemented by composing the elements with blank rectangles of the right size, using recursive invocations of the above and `beside` methods. A complete implementation of class `ArrayElement` with these simplifications is shown below.

```
import Element.elem
```

```
abstract class Element {  
    def contents: Array[String]  
  
    def width: Int = contents(0).length  
    def height: Int = contents.length  
  
    def above(that: Element): Element = {  
        val this1 = this widen that.width  
        val that1 = that widen this.width  
        elem(this1.contents ++ that1.contents)  
    }  
  
    def beside(that: Element): Element = {  
        val this1 = this heighten that.height  
        val that1 = that heighten this.height  
        elem(  
            for ((line1, line2) <- this1.contents zip that1.contents)  
            yield line1 + line2)  
    }  
  
    def widen(w: Int): Element =  
        if (w <= width) this  
        else {  
            val left = elem(' ', (w - width) / 2, height)  
            var right = elem(' ', w - width - left.width, height)  
            left beside this beside right  
        }  
  
    def heighten(h: Int): Element =  
        if (h <= height) this  
        else {  
            val top = elem(' ', width, (h - height) / 2)  
            var bot = elem(' ', width, h - height - top.height)  
            top above this above bot  
        }  
  
    override def toString = contents mkString "\n"  
}
```

A fun way to test almost all elements of the API for layout elements is writing a test program that draws a spiral with a given number of edges. The test

program is called `layout.Spiral`. You invoke it like this:

```
$ scala layout.Spiral 6
```

This should draw a spiral with 6 edges as shown below:

```
+----+
| 
| +-+
| + |
| | 
+---+
```

Here's a larger example:

```
$ scala layout.Spiral 17
```

```
+-----+
| 
| +-----+
| | 
| | +-----+ | | | | |
| | | 
| | | +----+ | | 
| | | | 
| | | | ++ | | | 
| | | | | 
| | | | +--+ | | | 
| | | | | 
| | | +-----+ | | 
| | | | 
| | +-----+ | | 
| | | 
| +-----+ | | 
| | 
+-----+
```

The spiral-drawing program is shown below.

```
import Element._
object Spiral {
    val space = elem(" ")
```

```

val corner = elem("+")

def spiral(nedges: Int, direction: Int): Element = {
    if (nedges == 1) elem("+")
    else {
        val sp = spiral(nedges - 1, (direction + 3) % 4)
        def verticalBar = elem('|', 1, sp.height)
        def horizontalBar = elem('-', sp.width, 1)
        if (direction == 0)
            (corner beside horizontalBar) above (sp beside space)
        else if (direction == 1)
            (sp above space) beside (corner above verticalBar)
        else if (direction == 2)
            (space beside sp) above (horizontalBar beside corner)
        else
            (verticalBar above corner) beside (space above sp)
    }
}

def main(args: Array[String]) = println(spiral(args(0).toInt, 0))
}

```

The program defines a `spiral` method which takes two parameters, the number of edges to be drawn and a direction in which the first edge should be drawn. Directions are integers from 0 to 3, where 0 means north, 1 west, 2 south, and 3 means east. A single edge spiral is of the form ‘+’. For spirals of more than one edge, the method first draws recursively a spiral with one fewer edges, and then adds a new edge consisting of a horizontalbar or verticalbar and a corner ‘+’.

10.15 Scala’s class hierarchy

Figure 10.2 shows an outline of Scala’s class hierarchy. At the top of the hierarchy there is class `Any`. Every Scala class inherits from this class. Class `Any` defines some methods which are inherited by all other classes. These include the following:

```
final def ==(that: Any): Boolean
```

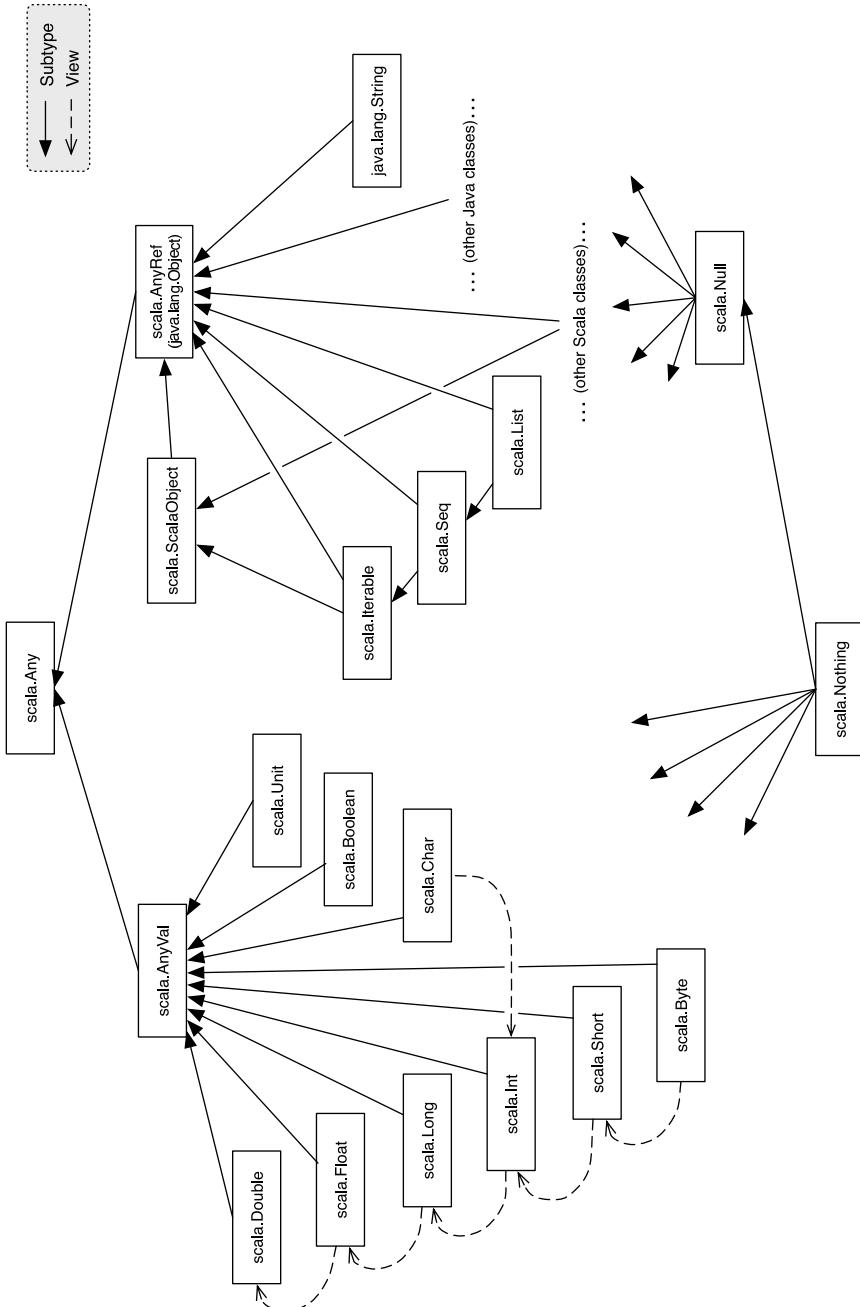


Figure 10.2: Class hierarchy of Scala.

```
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def hashCode: Int
def toString: String
```

This means that every Scala value can be compared using ‘==’, ‘!=’ or `equals`, hashed using `hashCode`, and formatted using `toString`. The equality and inequality methods ‘==’ and ‘!=’ are declared `final` in class `Any`, which means that they cannot be overridden in subclasses. In fact, ‘==’ is always the same as `equals` and ‘!=’ is always the negation of `equals`. So individual classes can tailor what ‘==’ or ‘!=’ means by overriding the `equals` method. [Chapter 21](#) will have more to say how this should be done.

The root class `Any` has two subclasses: `AnyVal` and `AnyRef`. `AnyVal` is the parent class of every built-in *value class* in Scala. There are nine such value classes: `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Boolean`, and `Unit`. The first eight of these correspond to Java’s primitive types, and their values are represented as Java’s primitive values. The instance objects of these classes are all written as literals in Scala. For instance, `42` is an object which is an instance of `Int`, `'x'` is an instance of `Char` and `false` is an instance of `Boolean`. You cannot create objects of these classes using `new`. This is enforced by the “trick” that value classes are all defined to be abstract. So if you write:

```
scala> new Int
```

you would get:

```
<console>:4: error: class Int is abstract; cannot be instantiated
val res0 = new Int
^
```

The last value class `Unit`, corresponds roughly to Java’s `void` type; it is used as the return type of a method which does not otherwise return an interesting result. `Unit` has a single instance value, which is written `()`.

As explained in [Chapter 5](#), the value classes support the usual arithmetic and boolean operators as methods. For instance, `Int` has methods named `+` and `*`, or `Boolean` has methods named `||` and `&&`. Value classes also inherit all methods from class `Any`. You can test this in the interpreter:

```
scala> 42.toString
res0: java.lang.String = 42

scala> 42.hashCode
res1: Int = 42

scala> 42.equals(42)
res2: Boolean = true
```

Note that the value class space is flat; all value classes are subtypes of `scala.AnyVal`, but they do not subclass each other. Instead there are implicit conversions (or: views) between elements of different value classes. For instance a number of class `scala.Int` is automatically widened to an element of class `scala.Long` when required.

Implicit conversions are also used to add more functionality to value types. For instance the type `Int` supports all of the operations below.

```
scala> 42 max 43
res3: Int = 43

scala> 42 min 43
res4: Int = 42

scala> 1 until 10
res5: Range = Range(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> 1 to 10
res6: Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> 3.abs
res7: Int = 3

scala> (-3).abs
res8: Int = 3
```

Here's how this works: The methods `min`, `max`, `until`, `to`, and `abs` are all defined in a class `scala.runtime.RichInt` and there is an implicit conversion from class `Int` to `RichInt`. The conversion is applied whenever a method is invoked on an `Int`, which is undefined in `Int` but defined in `RichInt`. Similar "booster classes" and implicit conversions exist for the other value classes.

The other subclass of the root class Any is class AnyRef. This is the base class of all *reference classes* in Scala. On the Java Platform, AnyRef is in fact just an alias for class `java.lang.Object`. So classes written in Java as well as classes written in Scala all inherit from AnyRef. Scala classes are different from Java classes in that they also inherit from a special marker interface called `ScalaObject`.

10.16 Implementing primitives

How is all this implemented? In fact, Scala stores integers in the same way as Java: as 32-bit words. This is important for efficiency on the JVM and also for interoperability with Java libraries. Standard operations like addition or multiplication are implemented as primitive operations. However, Scala uses the “backup” class `java.lang.Integer` whenever an integer needs to be seen as a (Java) object. This happens for instance when invoking the `toString` method on an integer number or when assigning an integer to a variable of type Any. Integers of type Int are converted transparently to “boxed integers” of type `java.lang.Integer` whenever necessary.

All this sounds a lot like auto-boxing in Java 5 and it is indeed quite similar. There’s one crucial difference, though, in that boxing in Scala is much less visible than boxing in Java. Try the following in Java:

```
boolean isEqual(int x, int y) {  
    return x == y;  
}  
System.out.println(isEqual(42, 42));
```

You will surely get true. Now, change the argument types of `isEqual` to `java.lang.Integer` (or `Object`, the result will be the same):

```
boolean isEqual(Integer x, Integer y) {  
    return x == y;  
}  
System.out.println(isEqual(42, 42));
```

You will find that you get false! What happens is that the number 42 gets boxed twice, so that the arguments for x and y are two different objects. Because `==` means reference equality on reference types, and `Integer` is a

reference type, the result is `false`. This is one aspect where it shows that Java is not a pure object-oriented language. There is a difference between primitive types and reference types which can be clearly observed.

Let's repeat the same experiment in Scala:

```
scala> def isEqual(x: Int, y: Int) = x == y
isEqual: (Int,Int)Boolean
scala> isEqual(42, 42)
res9: Boolean = true
scala> def isEqual(x: Any, y: Any) = x == y
isEqual: (Any,Any)Boolean
scala> isEqual(42, 42)
res10: Boolean = true
```

In fact, the equality operation `==` in Scala is designed to be transparent with respect to the type's representation. For value types, it is the natural (numeric or boolean) equality. For reference types, `==` is treated as an alias of the `equals` method inherited from `Object`. That method is originally defined as reference equality, but is overridden by many subclasses to implement their natural notion of equality. This also means that in Scala you never fall into Java's well-known trap concerning string comparisons. In Scala, string comparison works as it should:

```
scala> val x = "abc"
x: java.lang.String = abc
scala> val y = "abc"
y: java.lang.String = abc
scala> x == y
res11: Boolean = true
```

In Java, the result of comparing `x` with `y` would depend on where `x` and `y` are declared. If they are declared in the same class, the comparison would yield `true`, but if they are declared in different classes, it would yield `false`.

However, there are situations where you need reference equality instead of user-defined equality. An example is hash-consing, where efficiency is paramount. For these cases, class `AnyRef` defines an additional `eq` method, which cannot be overridden, and which is implemented as reference equality

(*i.e.*, it behaves like `==` in Java for reference types). There's also the negation of `eq`, which is called `ne`. Example:

```
scala> val x = new String("abc")
x: java.lang.String = abc
scala> val y = new String("abc")
y: java.lang.String = abc
scala> x == y
res12: Boolean = true
scala> x eq y
res13: Boolean = false
scala> x ne y
res14: Boolean = true
```

10.17 Bottom types

At the bottom of the type hierarchy in [Figure 10.2](#) you see the two classes `scala.Null` and `scala.Nothing`. These are special types that handle some “corner cases” of Scala's object-oriented type system in a uniform way.

Class `Null` is the type of the `null` reference; it is a subclass of every reference class (*i.e.*, every class which itself inherits from `AnyRef`). `Null` is not compatible with value types. Therefore, you cannot assign a `null` value to an integer variable, say.

The type `Nothing` is at the very bottom of Scala's class hierarchy. It is a subtype of every other type. However, there exist no values of this type whatsoever. Why does it make sense to have a type without values? One use of `Nothing` is that it signals abnormal termination. For instance there's the `error` method in the `Predef` object of Scala's standard library, which is defined as follows.

```
def error(message: String): Nothing = throw new Error(message)
```

The return type of `error` is `Nothing`, which tells users that the method will not return normally (it throws an exception instead). Because `Nothing` is a subtype of every other type, you can use methods like `error` in very flexible ways. For instance:

```
def divide(x: Int, y: Int): Int =  
  if (y != 0) x / y  
  else error("can't divide by zero")
```

The then-branch of the conditional above has type `Int`, whereas the `else` branch has type `Nothing`. Because `Nothing` is a subtype of `Int`, the type of the whole conditional is `Int`, as required.

10.18 Conclusion

In this section, you have seen more concepts related to object-oriented programming in Scala. Among others, you have encountered abstract classes, inheritance and subtyping, class hierarchies, class parameter fields, and method overriding. You should have developed a feel for constructing a non-trivial class hierarchy in Scala.

Another important aspect that was treated mostly “between the lines” in this chapter was composition. Layout elements are a good example of a system where objects can be constructed from simple parts (arrays, lines, and rectangles) with the aid of composing operators (`above` and `beside`). Such composing operators are also often called *combinators* because they combine elements of some domain into new elements.

Thinking in terms of combinators is generally a good way to approach library design: It pays to think about the fundamental ways to construct objects in an application domain. What are the simple objects? In what ways can more interesting objects be constructed out of simpler ones? How do combinators hang together? What are the most general combinations? Do they satisfy any interesting laws? If you have good answers to these questions, your library design is on track.

So far, the whole treatment of classes and objects was based on single inheritance, where every class inherits from just one superclass. In the next chapter, you will find out about *traits*, which let you construct even more interesting class hierarchies.

Chapter 11

Traits and Mixins

Traits offer a more fine-grained way to reuse code than normal inheritance. Like inheritance, traits let you add code to a class that is written elsewhere in the program. Unlike inheritance, however, a class can “mix in” any number of traits. With inheritance, only one superclass is allowed per class.

This chapter introduces traits and shows you two of the most common ways they are useful: widening thin interfaces to thick ones, and defining stackable modifications. [Chapter 25](#) will discuss the role of traits in defining modules.

11.1 Syntax

A trait definition looks just like a class definition except that it uses the keyword `trait`:

```
trait Printable {  
    def print() {  
        println(this)  
    }  
}
```

Once a trait is defined, it can be mixed into a class using the `with` keyword.¹ For example, here are a couple of classes that use the above trait:

```
class Frog extends Object with Printable {  
    override def toString = "a frog"  
}
```

Methods inherited via a trait can then be used just like methods inherited otherwise:

```
scala> val frog = new Frog  
frog: Frog = a frog  
  
scala> frog.toString  
res0: java.lang.String = a frog  
  
scala> frog.print()  
a frog
```

A trait is also usable as a type. Here is an example using `Printable` as a type:

```
scala> val pr: Printable = frog  
pr: Printable = a frog  
  
scala> pr.print()  
a frog
```

As a syntactic shorthand, you can extend a trait directly if you do not care to extend a more specific superclass. You could have just as well written the `Frog` class like this:

```
class Frog extends Printable {  
    override def toString = "a frog"  
}
```

A trait can define the same kinds of members that you would otherwise find in a class. However, a trait cannot have any “class” parameters, parameters

¹The phrase “mix in” essentially means that you get the benefit of both inheritance and composition. Your class inherits the abstract interface of the trait it mixes in through Java’s interface inheritance mechanism, and “inherits” any concrete method implementations in the trait via composition.

passed to the primary constructor of a class.² In other words, although you could define a class like this:

```
class Point(x: Int, y: Int) {  
}
```

The following attempt to define a trait would not compile:

```
trait NoPoint(x: Int, y: Int) { // Does not compile  
}
```

11.2 Thin versus thick interfaces

One major use of traits is to automatically add methods to a class in terms of methods that the class already has. That is, traits can expand a *thin* interface into a *thick* interface.

Thin versus thick interfaces are a commonly faced trade-off in object-oriented design. The trade-off is between the implementors and clients of an interface. A thick interface has many methods, which make it convenient for the caller. Clients can pick a method that corresponds exactly to the functionality they want, instead of having to use a more primitive method and write extra code to adapt the method to their needs. A thin interface, on the other hand, has fewer methods, and thus is easier to implement. Java's interfaces are most often of the thin kind.

Scala traits make thick interfaces more convenient. Unlike Java interfaces, traits can define methods that include code, *i.e.*, they can not only declare abstract methods, but also define concrete ones. You just saw one example of this: the `print` method of the `Printable` trait is concrete.

Adding a concrete method to a trait tilts the thin-thick trade-off heavily towards thick interfaces. Unlike with Java interfaces, adding a concrete method to a Scala trait is a one-time effort. You only need to implement the method once, in the trait itself, instead of needing to reimplement it for every class that mixes in the trait. Thus, thick interfaces are less work to provide in Scala than in a language without traits.

²They are called *class parameters* because you can only define them for classes, not for traits or singleton objects.

To use traits as interface thickeners, simply define a trait with a small number of abstract methods—the thin part of the trait’s interface—and a potentially large number of concrete methods, all implemented in terms of the abstract methods. Then you can take any class implementing the thin version of the interface, mix in the thickening trait, and end up with a class that has all of the thick interface available.

11.3 The standard Ordered trait

Comparison is one place where a thick interface is convenient. Whenever you have objects that are ordered, it is convenient if you can use the precise ordering operation for each situation. Sometimes you want ‘<’ (less than), and sometimes you want ‘ \leq ’ (less than or equal). A thin interface would provide just one of these methods, forcing you to write things like $((x < y) \mid\mid (x == y))$. A thick interface would provide you with all of the usual comparison operators, so that you can directly write things like $(x \leq y)$. The standard Ordered trait allows you to implement one method and gain access to four different variations.

Here is what you might do without the Ordered trait. If you are implementing a Book class that has an ordering, you might write the following code:

```
class Book(val author: String, val title: String) {  
    def <(that: Book) =  
        (author < that.author) ||  
        ((author == that.author) && title < that.title)  
    def >(that: Book) = that < this  
    def <=(that: Book) = (this < that) || (this == that)  
    def >=(that: Book) = (this > that) || (this == that)  
    override def equals(that: Any) =  
        that match {  
            case that: Book =>  
                (author == that.author) && (title == that.title)  
            case _ => false  
        }  
}
```

```
}
```

This class defines four comparison operators (`<`, `>`, `<=`, and `>=`) plus the standard `equals` method. As an aside, you should always override `equals` instead of `==`, so as to maintain Java compatibility. Don't worry; the compiler will fuss at you if you forget and try to override `==`. You'll find more explanations about equality in chapter 21.

This `Book` class is a classic demonstration of the costs of defining a thick interface. First, notice that three of the comparison operators are defined in terms of the first one. For example, `>` is defined as the opposite of `<`, and `<=` is defined as literally "less than or equal." Additionally, notice that all three of these methods would be the same for any other class that is comparable. There is nothing special about books regarding `<=`. In a comparison context, `<=` is *always* used to mean "less than or equals." There is a lot of boilerplate code in this class which would probably show up in similar form everytime you create a class that implements ordering comparison operations.

The `Ordered` trait can reduce this waste. To use the `Ordered` trait, your class must define a single `compare` method which does the real work of comparison. Then you can mix in `Ordered`, and clients of your class can use four different comparison operations even though you did not define them. Here is how it looks for the `Book` class:

```
class Book(val author: String, val title: String)
extends Ordered[Book]
{
    def compare(that: Book): Int =
        if (author < that.author) -1
        else if (author > that.author) 1
        else if (title < that.title) -1
        else if (title > that.title) 1
        else 0

    override def equals(that: Any) =
        that match {
            case that: Book => compare(that) == 0
            case _ => false
        }
}
```

Warning: if you use `Ordered`, you must still define your own `equals()` method. For technical reasons involving runtime types, this method cannot be correctly defined in the `Ordered` trait itself.

The complete `Ordered` trait, minus comments and compatibility cruft, is as follows:

```
trait Ordered[T] {  
    def compare(that: T): Int  
  
    def <(that: T): Boolean = (this compare that) < 0  
    def >(that: T): Boolean = (this compare that) > 0  
    def <=(that: T): Boolean = (this compare that) <= 0  
    def >=(that: T): Boolean = (this compare that) >= 0  
}
```

11.4 Traits for modifying interfaces

You have now seen one major use of traits: turning a thin interface into a thick one. Now let us turn to a second major use: provide stackable modifications to behavior. This section focuses on modifications, while the next one is about making them stackable.

As an example modification, consider the caching of hash codes. Hash codes need to be computed quickly, because hash-based collections make more hash-code comparisons than full `==` comparisons. If a class's hash-code computation is slow, then collections holding that class can waste a lot of time computing hashes.

One way to speed up hashing is to cache the computed values. That way, even if a hash routine is not fast already, the cost is only paid one time per object instead of one time per call to `hashCode`. Here is a simple trait that caches the hash code of the class it is mixed into:

```
trait HashCaching {  
    /** A cache holding the computed hash. */  
    private var cachedHash: Int = 0  
    /** A boolean indicating whether the cache is defined */  
    private var hashComputed: Boolean = false  
    /** The hash code computation is abstract */
```

```
def computeHash: Int
/** Override the default Java hash computation */
override def hashCode = {
    if (!hashComputed) {
        cachedHash = computeHash
        hashComputed = true
    }
    cachedHash
}
```

Given this trait, the Book class can now be modified to include cached hash codes.

```
class Book(val author: String, val title: String)
extends Ordered[Book]
with HashCaching
{
    // compare and equals() as before...
    def computeHash = author.hashCode + title.hashCode
}
```

Stylistically, this version of HashCaching does not strictly *modify* the hashCode method, but instead refactors hashing so that the computation is actually done in a separate method. It is also possible to have HashCaching modify a hashCode in place, but there is a catch.

The new HashCaching can be written without problems:

```
trait HashCaching {
    private var cachedHash: Int = 0
    private var hashComputed: Boolean = false
    /** Override the default Java hash computation */
    override def hashCode = {
        if (!hashComputed) {
            cachedHash = super.hashCode
            hashComputed = true
        }
    }
}
```

```
    cachedHash  
}  
}
```

However, attempting to use it runs into a problem:

```
class Book(val author: String, val title: String)  
extends Ordered[Book]  
with HashCaching  
{  
    // compare and equals() as before...  
  
    override def hashCode = {  
        Thread.sleep(3000) // simulate a VERY slow hash  
        author.hashCode + title.hashCode  
    }  
}
```

This version does not get its hash code cached! The problem is that Book’s hashCode method overrides HashCaching’s. In this way, a class can always override anything it inherits, including things it inherits via traits. This is important so that you stay in control of the classes you are writing. However, sometimes you really do want a trait to modify one of your methods.

In this case, you can divide your class into two parts, putting methods you wish to be overridable into a “base” class. For the book example it looks like this:

```
abstract class BaseBook(val author: String, val title: String)  
{  
    override def hashCode = {  
        Thread.sleep(3000)  
        author.hashCode + title.hashCode  
    }  
}  
  
class Book(author: String, title: String)  
extends BaseBook(author, title)  
with Ordered[Book]  
with HashCaching  
{
```

```
// compare and equals() as before...
}
```

Now the hash value gets cached, just as desired.

Arranging HashCaching like this is not just a matter of stylistic taste. If you arrange a trait like this, using `super` calls instead of refactoring a method, then the trait can be stacked with other traits that modify the same method.

11.5 Stacking modifications

Stackable modifications can be combined with each other in any way you desire. Given a set of stackable traits that each modify a class in some way, you can pick and choose any of those traits you would like to use when defining new classes.

Let us continue the hashing example, and consider another common hashing challenge: many times, the most useful bits of a hash are concentrated somewhere in the middle. For some hashing collections, it is better to either move the useful bits to low-order bits, or to spread the useful bits throughout the integer.

Thus, you can often improve on a hash function by scrambling the bits around. Here is a trait that does just that:

```
trait HashScrambling
{
    override def hashCode = {
        val original = super.hashCode
        def rl(i: Int) = Integer.rotateLeft(original, i)
        original ^ rl(8) ^ rl(16) ^ rl(24)
    }
}
```

To use this in the `Book` class, change its definition to:

```
class Book(author: String, title: String)
extends BaseBook(author, title)
with Ordered[Book]
with HashScrambling
```

```
with HashCaching
{
    // compare and equals() as before...
}
```

Be aware that the order of the `with` clauses is significant. With the order specified above, the scrambled hash is cached. If you mix in the traits in the opposite order, then the unscrambled hash will be cached, and the hash will be scrambled again each time `hashCode` is called.

11.6 Locking and logging queues

Locking and logging queues are a well-known example of stackable modifications. It is well known within the Scala community because it was considered during the design of traits, and it is known in language-design circles because of early Scala materials that use it as an example. This section overviews the problem, and gives the solution in Scala using stackable traits.

The locking logging example involves the following three concepts:

- *Queue*: an abstract class that can get and put elements.
- *Locking*: a modification of a queue to use locking.
- *Logging*: a modification of a queue to log gets and puts.

Using stackable traits, the solution is just as easy as the hash code examples seen so far. To simplify the example, this code assumes that the queues are over integers, as opposed to arbitrary objects.

```
trait Queue {
    def get(): Int
    def put(x: Int)
}

trait LockingQueue extends Queue {
    abstract override def get(): Int =
        synchronized { super.get() }
    abstract override def put(x: Int) =
        synchronized { super.put(x) }
```

```
}

trait LoggingQueue extends Queue {
    def log(message: String) = println(message)
    abstract override def get(): Int = {
        val x = super.get()
        log("got: " + x)
        x
    }
    abstract override def put(x: Int) {
        super.put(x)
        log("put: " + x)
    }
}
```

There are two new issues introduced by this example that did not appear in the hashing examples.

First, the locking and logging traits specify `Queue` as a superclass. This means that any class they are mixed into, must be a subtype of `Queue`. The `hashCode` examples only overrode methods from class `Any`, so it was not necessary to specify a superclass. In this example, the traits override methods that are specific to `Queues`.

Second, these traits make `super` calls on methods that are declared abstract! Such calls are illegal for normal classes, because at runtime they will certainly fail. For a trait, however, such a call can succeed, so long as the trait is mixed in *after* another trait which gives a concrete definition to the method. This construct is frequently needed with traits that implement stackable modifications. To tell the compiler you are doing this on purpose, you must mark such methods as `abstract override`. This combination of modifiers is only allowed for members of traits; it makes no sense outside of traits.

By the way, this second issue shows an important difference between `super` calls in traits and `super` calls in classes. In a class, `super` calls invoke a known method, while in a trait, the method is different for each place in the code the trait is mixed in.

Stylistically, notice that there is as yet no commitment about which traits are used by which class. Clients remain free to select which traits to use each time they create a new queue using `new`. Here is a concrete queue class,

along with two queue objects using different trait combinations:³

```
import scala.collection.mutable.ArrayBuffer
class StandardQueue extends Queue {
    private val buf = new ArrayBuffer[Int]
    def get = buf.remove(0)
    def put(x: Int) = buf += x
}
val q1 = new StandardQueue with LockingQueue with LoggingQueue
val q2 = new StandardQueue with LoggingQueue
```

Given these two modification traits, you can use all four combinations of mixins: with or without logging, and with or without locking. Additionally, if you choose both locking and logging, you can select which order you want them to happen in, i.e. whether to log before or after the lock is acquired. Thus by defining two traits, you get five different possible modifications. With three traits, the number would rise to sixteen combinations, and with four traits, an enormous sixty-five possible combinations. Thus, whenever you see a way to divide behavior into two or more stackable traits, you should strongly consider doing so. Stackable traits provide a large number of useful combinations out of a small amount of code.

Aside: The previous example illustrates that Scala has synchronized blocks to make a region of code atomic. Staying in line with the expression-oriented nature of Scala, a synchronized block is an expression that computes a result. For instance, the first synchronized block in method get computes an integer as result. Unlike in Java, there is no synchronized method modifier in Scala. So you can't write

```
synchronized def method() = body // ERROR
```

But the following can always be used as a replacement:

```
def method() = synchronized { body }
```

You'll find out more about concurrency in [Chapter 23](#).

³As mentioned in [Chapter 3](#), an import statement allows you to use a class's or a trait's simple name. Since you import `scala.collection.mutable.ArrayBuffer` here, you can just say `ArrayBuffer` in the rest of the source file.

11.7 Traits versus multiple inheritance

Traits are not the same as the multiple inheritance used in other languages. The difference is in the meaning of super calls. In multiple inheritance, a super call invokes a method in one of the superclasses of the calling class. With traits, the invoked method is found according to a *linearization* of the classes and traits that are mixed into a class. Because of this difference, Scala's traits support stacking of modifications as described above.

Before looking at linearization, take a moment to consider how to stack modifications in a language with multiple inheritance. Perhaps your first try is to implement locking and logging queues as described previously. You would then instantiate a queue and call a method on it, like this:

```
val q1 = new StandardQueue with LockingQueue with LoggingQueue
q1.put(42) // which put is called?
```

The first question is, which put method gets invoked by this call? Perhaps the rule is that the last superclass wins, in which case LoggingQueue will get called. LoggingQueue calls super and then logs the call, and that is it. No locking happened! Likewise, if the rule is that the first superclass wins, the resulting queue locks accesses but does not log them. Thus neither ordering works.

Your next try might be to make an explicit subclass for locking, logging queues. Now you have new problems. for example, suppose you try the following:

```
trait LockingLoggingQueue extends LockingQueue with LoggingQueue {
    def put(x: Int) = {
        LockingQueue.super.put(x)
        LoggingQueue.super.put(x)
    }
}
```

Now what happens is that the base class's put method gets called *twice*—once with locking, and once with logging, but in no case with both.

There is simply no good solution to this problem using multiple inheritance. You have to factor the code differently. By contrast, the traits solution in Scala is straightforward. You simply mix in LockingQueue or

LoggingQueue, or both, and the Scala treatment of super makes it all work out. Something must be different here from multiple inheritance.

The answer is in linearization. When you instantiate a class with new, Scala takes the class and all of its inherited classes and traits and puts them in a single, long, *linear* order. Then, whenever you call super inside one of those classes, the invoked method is the next one up the chain. If all of the methods but the last calls super, then the net result is the stackable behavior described earlier.

The precise order of the linearization is described in the language specification. It is a little bit complicated, but the main thing you need to know is that, in any linearization, a class is always linearized before *all* of its superclasses and mixed in traits. Thus, when you write a method that calls super, that method is definitely modifying the behavior of the superclasses and mixed in traits, not the other way around. This rule means that you can use super the same way for writing both subclasses that modify a superclass's methods, and stackable traits that modify the class they are mixed into.

The main properties of Scala's linearization are illustrated by the following example. You can safely skip the rest of this section if you are not interested in the details right now.

Say you have a class C which inherits from a superclass Sup and two traits Trait1 and Trait2. Trait2 extends in turn another trait Trait3.

```
class Sup
trait Trait1 extends Sup
trait Trait2 extends Sup with Trait3
class C extends Sup with Trait1 with Trait2.
```

Then the linearization of C is computed from back to front as follows. The *last* part of the linearization of C is the linearization of class Sup. This linearization is copied over without any changes. The second to last part is the linearization of the first mixin, trait Trait1, but all classes that are already in the linearization of Sup are left out now, so that each class appears only once C's linearization. This is preceded by the linearization of Trait2, where again any classes that have already been copied in the linearizations of the superclass or the first mixin are left out. Finally, the first class in the linearization of C is C itself.

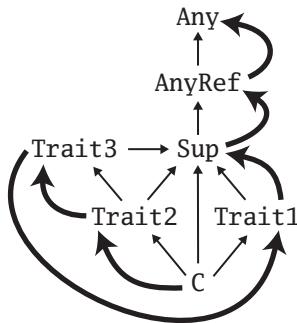


Figure 11.1: Linearization of class C

If you apply this schema to the previous classes and traits, you should get

Class	Linearization
Sup	Sup, AnyRef, Any
Trait1	Trait1, Sup, AnyRef, Any
Trait2	Trait2, Trait3, Sup, AnyRef, Any
C	C, Trait2, Trait3, Trait1, Sup, AnyRef, Any

Figure 11.1 presents

a diagram that depicts the linearization of class C graphically.

11.8 To trait, or not to trait?

Whenever you implement a reusable collection of behavior, you will have to decide whether you want to use a trait or an abstract class. There is no firm rule, but here are a few guidelines to consider.

If the behavior will not be reused, then make it into a concrete class. It is not reusable behavior after all.

If it might be reused in multiple, unrelated classes, then make it a trait. Only traits can be mixed into different parts of the class hierarchy.

If you want to inherit it in Java code, then use an abstract class. Since traits with code do not have a close Java analog, it is awkward to inherit from a trait in Java. Inheriting from a Scala class, meanwhile, is exactly like inheriting from a Java class.

If you plan to distribute it in compiled form, then lean towards using an abstract class, unless the behavior is only used within the code your group

distributes. When a trait changes, classes that inherit from it must be recompiled more frequently than if the trait had been made a class.

If efficiency is very important, lean towards using class. Most Java runtimes make a virtual method invocation of a class member a faster operation than an interface method invocation. Traits get compiled to interfaces and therefore pay a slight performance overhead. However, you should make this choice only if you know that the class or trait in question will constitute a performance bottleneck.

If you still do not know, after considering the above, then start by making it as a trait. You can always change it later, and in general using a trait keeps more options open.

Chapter 12

Case Classes and Pattern Matching

This chapter introduces *case classes* and *pattern matching*, twin constructs that support you when writing regular, non-encapsulated data structures. The two constructs are particularly helpful for tree-like recursive data.

If you have programmed in a functional language before, then you will probably recognize pattern matching. Case classes will still be new, however. Case classes are Scala's secret for allowing pattern matching without requiring a large amount of boilerplate to set everything up. In the common case, you add a single `case` keyword to each class that you want to be pattern matchable.

This chapter starts with a simple example of case classes and pattern matching. It then goes through all of the kinds of patterns that are supported, talks about the role of *sealed* classes, discusses the `Option` type, and shows some unobvious places in the language that pattern matching is used. Finally, a larger, more realistic example of pattern matching is shown.

12.1 A simple example

Before delving into all the rules and nuances of pattern matching, it is worth looking at a simple example to get the general idea. Let us say you want to write a library that manipulates arithmetic expressions.

A first step to tackle this problem is the definition of the input data. To keep things simple, let's concentrate on arithmetic expressions consisting of variables, numbers, and unary and binary operations. This is expressed by the following hierarchy of Scala classes:

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
                  left: Expr, right: Expr) extends Expr
```

There is an abstract base class `Expr` with four subclasses, one for each kind of expressions that's considered.

Instead of an abstract class, we could have equally well chosen to model the root of that class hierarchy as a trait (modeling it as an abstract class is slightly more efficient). The bodies of all five classes are empty. In Scala you can leave out the braces around an empty class body if you wish, so `class C {}` is the same as `class C {}`.

Case classes

The only other noteworthy thing about these declarations is that each subclass has a `case` modifier. Classes with such a modifier are called *case classes*. Using the modifier makes the Scala compiler add some syntactic conveniences to your class.

First, it adds a factory method with the name of the class. This means you can write directly, say, `Var("x")` to construct a `Var` object instead of the slightly longer `new Var("x")`:

```
scala> val v = Var("x")
v: Var = Var(x)
scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))
```

Second, all arguments in the parameter list of a case class implicitly get a `val` prefix, so they are maintained as fields.

```
scala> v.name
res0: String = x
scala> op.left
res1: Expr = Number(1.0)
```

Third, the compiler adds the “natural” implementations of `toString`, `hashCode`, and `equals` to your class. They will print, hash, and compare a whole tree consisting of the class and (recursively) all its arguments. Since `==` in Scala always forwards to `equals`, this means in particular that elements of case classes are always compared structurally.

```
scala> println(op)
BinOp(+, Number(1.0), Var(x))
scala> op.right == Var("x")
res3: Boolean = true
```

All these conventions add a lot of convenience, at a small price. The price is that you have to write the case modifier and that your classes and objects become a bit larger because additional methods are generated and an implicit field is added for each constructor parameter.

However, the biggest advantage of case classes is that they support pattern matching.

Pattern Matching

Let’s say you want to simplify arithmetic expressions of the kinds presented above. There is a multitude of possible simplification rules. The following three rules just serve as an illustration:

```
UnOp("-", UnOp("-", e)) => e    // Double negation
BinOp("+", e, Number(0)) => e    // Adding zero
BinOp("*", e, Number(1)) => e    // Multiplying by one
```

Using pattern matching, these rules can be taken almost as they are to form the core of a simplification method in Scala:

```
def simplifyTop(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) => e    // Double negation
  case BinOp("+", e, Number(0)) => e    // Adding zero
  case BinOp("*", e, Number(1)) => e    // Multiplying by one
  case _ => expr
}
```

Here’s a test application of `simplifyTop`:

```
scala> simplifyTop(UnOp("-", UnOp("-", Var("x"))))
res4: Expr = Var(x)
```

The right hand side of `simplifyTop` consists of a `match` expression. `match` corresponds to `switch` in Java, but is written after the selector expression. *I.e.* it's

```
selector match { alternatives }
```

instead of

```
switch (selector) { alternatives }
```

A pattern match includes a sequence of *alternatives*, each starting with the keyword `case`. Each alternative includes a *pattern* and one or more expressions to evaluate if the pattern matches. An arrow symbol `=>` separates the pattern from the expressions.

A match expression is evaluated by trying each of the patterns in the order they are written. The first pattern that matches is selected, and the part following the arrow is selected and executed.

A constant pattern like `"-"` or `1` matches values that are equal to the constant with respect to `==`. A variable pattern like `e` matches every value. The variable then refers to that value in the right hand side of the case clause. In this example, note that the first three examples evaluate to `e`, a variable that is bound within the associated pattern. The wildcard pattern `"_"` also matches every value, but it does not introduce a variable name to refer to that value. In this example, notice how the match ends with a default case that does nothing.

A constructor pattern looks like `UnOp("-", e)`. This pattern matches all values of type `UnOp` whose first argument matches `"-"` and whose second argument matches `e`. Note that the arguments to the constructor are themselves patterns. This allows you to write deep patterns like

```
UnOp("-", UnOp("-", e))
```

using a concise notation. Imagine trying to implement this same functionality using the visitor design pattern [Gam94]! Almost as awkward, imagine implementing it as a long sequence of `if` statements, type tests, and type casts.

match compared to switch

Match expressions can be seen as a generalization of Java-style switches. A Java-style switch can be naturally expressed as a match expression where each pattern is a constant and the last pattern may be a wildcard (which represents the default case of the switch). There are three differences to keep in mind, however. First, match is an *expression* in Scala, *i.e.* it always returns a result. Second, Scala's alternative expressions never “fall through” into the next case. Third, if none of the patterns matches, an exception named `MatchError` is thrown. This means you always have to make sure that all cases are covered, even if it means adding a default case where there's nothing to do.

Here's an example:

```
expr match {  
    case BinOp(op, left, right) => println(expr+"is a binary operation")  
    case _ =>  
}
```

The second case is necessary because otherwise the expression above would throw a `MatchError` for every `expr` argument which is not a `BinOp`.

12.2 Kinds of patterns

The previous example showed several kinds of patterns in quick succession. Now take a minute to look at each.

The syntax of patterns is easy, so do not worry about that too much. All patterns look exactly like the corresponding expression. For instance, the pattern `Number(x)` matches any number object, binding `x` to the number. Used as an expression, `Number(x)` recreates an equivalent object, assuming `x` is already bound to the number. The main thing to pay attention to is just what kinds of patterns are possible.

The wildcard

The wildcard pattern `_` matches any object whatsoever. You have already seen it used as a default, catch-all alternative, like this:

```
expr match {
    case BinOp(op, left, right) => println(expr+"is a binary operation")
    case _ =>
}
```

It can also be used to ignore parts of an object that you do not care about. For example, the above example does not actually care what the elements of a binary operation are. It just checks whether it is a binary operation at all. Thus the code can just as well use the wildcard pattern for the elements of the `BinOp`:

```
expr match {
    case BinOp(_, _, _) => println(expr+"is a binary operation")
    case _ => println("It's something else")
}
```

Constants

A constant pattern matches only itself. Any literal may be used as a constant. For example, `5`, `true`, and `"hello"` are all constant patterns. Also, any named value can also be used as a constant. For example, `Nil` is a pattern that matches only the empty list. Here are some examples of constant patterns:

```
def describe(x: Any) = x match {
    case 5 => "five"
    case true => "truth"
    case "hello" => "hi!"
    case Nil => "the empty list"
    case _ => "something else"
}
```

Here is how the above pattern match looks in action:

```
scala> describe(5)
res5: java.lang.String = five
scala> describe(true)
res6: java.lang.String = truth
scala> describe("hello")
```

```
res7: java.lang.String = hi!
scala> describe(Nil)
res8: java.lang.String = the empty list
scala> describe(List(1,2,3))
res9: java.lang.String = something else
```

Variables

A variable pattern matches any object, just like a wildcard. Unlike a wildcard, Scala binds the variable to whatever the object is. You can then use this variable to act on the object further. For example, here is a pattern match that has a special case for zero, and a default case for all other values. The default cases uses a variable pattern so that it has a name for the value, whatever it is.

```
10 match {
    case 0 => "zero"
    case somethingElse => "not zero: " + somethingElse
}
```

Variable or constant?

Constant patterns can have symbolic names. For instance, the following works as in Java:

```
scala> import Math._
import Math._
scala> E match {
    | case Pi => "strange math? Pi = "+Pi
    | case _ => "OK"
    |
}
res10: java.lang.String = OK
```

This poses the question how the Scala compiler knows that `Pi` is a constant imported from the `java.lang.Math` object, and not a variable that stands for the selector value itself. Scala uses a simple lexical rule for disambiguation: A simple name starting with a lower-case letter is taken to be a pattern

variable; all other references are taken to be constants. To see the difference, create a lower-case alias for `pi` and try with that:

```
scala> val pi = Math.Pi
pi: Double = 3.141592653589793

scala> E match {
|   case pi => "strange math? Pi = "+pi
| }
res11: java.lang.String = strange math? Pi =
2.718281828459045
```

If you need to, you can still use a lower-case name for a pattern constant, using one of two tricks. First, if the constant is a field of some object, you can prefix it with a qualifier. For instance, `pi` is a variable pattern, but `this.pi` or `obj.pi` are constants. If that does not work (because `pi` is a locally defined variable, say), you can alternatively enclose the variable name in back ticks. For instance, ‘`pi`’ would again be interpreted as a constant, not as a variable.

Aside: The back tick syntax for identifiers is also useful outside of patterns. The idea is that you can put any string that’s accepted by the host VM as an identifier between back ticks. The result is always a Scala identifier. This holds even if the name contained in the back ticks would be a Scala reserved word. A typical use case is accessing the (static) `yield` method in Java’s `Thread` class. You cannot write `Thread.yield` because `yield` is a reserved word in Scala. However, you can still refer to the method using back ticks, e.g. `Thread.‘yield’()`.

Constructors

Constructors are where pattern matching becomes really powerful. A constructor pattern looks like `BinOp("+", e, Number(0))`. It consists of a name (`BinOp`) and then a number of patterns within parentheses ("`+`", `e`, and `Number(0)`). Assuming the name designates a case class, such a pattern means to first check that the object is a member of the named case class, and then to check that the constructor parameters of the object match the extra patterns supplied.

These extra patterns mean that Scala patterns support *deep matches*. Such patterns not only check the top-level object supplied, but also check

the contents of the object against further patterns. Since the extra patterns can themselves be constructor patterns, you can use them to check arbitrarily deep into an object. For example, the pattern `BinOp("+", e, Number(0))` checks that the top-level object is a `BinOp`, that its third constructor parameter is a `Number`, and that the value field of that number is 0. This pattern is one line long yet checks three levels deep.

Sequences

You can match against sequence types like `List` or `Array` just like you match against case classes. Use the same syntax, but now you can specify any number of elements within the pattern. For example, here is a pattern that checks for a three-element list starting with zero:

```
case List(0, _, _) => println("found it")
```

If you want to match against a sequence without specifying how long it can be, you can specify `_*` as the last element of the pattern. This funny-looking pattern matches any number of elements within a sequence, including zero elements. Here is an example that matches any list that starts with zero, regardless of how long the list is:

```
case List(0, _*) => println("found it")
```

Tuples

You can match against tuples, too. A pattern like `(_, _, _)` matches an arbitrary 3-tuple.

```
("hello", 10, true) match {  
    case (word, idx, bool) => // use word, idx and bool here...  
}
```

Typed patterns

Patterns have other uses as well. A form of patterns not seen so far is a convenient replacement for Java's type tests and type casts. Here's an example:

```
scala> def generalSize(x: Any) = x match {  
|   case s: String => s.length  
|   case m: Map[_,_] => m.size  
|   case _ => -1  
| }  
generalSize: (Any)Int  
  
scala> generalSize("abc")  
res12: Int = 3  
  
scala> generalSize(Map(1 -> 'a', 2 -> 'b'))  
res13: Int = 2  
  
scala> generalSize(Math.Pi)  
res14: Int = -1
```

The `generalSize` method returns the size or length of objects of various types. Its argument is of type `Any`, so it could be any value. If the argument is a `String`, the method returns the string's length. The pattern `s: String` is a typed pattern; it matches every (non-null) instance of `String`. The pattern variable `s` then refers to that string.

Note that, even though `s` and `x` refer to the same value, the type of `x` is `Any`, but the type of `s` is `String`. So you can write `s.length` in the alternative expression that corresponds to the pattern, but you could not write `x.length`, because the type `Any` does not have a `length` member.

An equivalent but more long-winded way that achieves the effect of a match against a typed pattern employs a type-test followed by a type-cast. Scala uses a different syntax than Java for these. To test whether an expression `expr` has type `String`, say, you write

```
expr.isInstanceOf[String]
```

To cast the same expression to type `String`, you would use

```
expr.asInstanceOf[String]
```

Using a type test and cast, you could rewrite the first case of the previous match expression as follows:

```
if (x.isInstanceOf[String]) {  
  val s = x.asInstanceOf[String]
```

```
s.length  
} else ...
```

The operators `isInstanceOf` and `asInstanceOf` are treated as predefined methods of class `Any` which take a type parameter in square brackets. In fact, `x.asInstanceOf[String]` is a special case of a method invocation with an explicit type parameter `String`.

As you will have noted by now, writing type tests and casts is rather long-winded in Scala. That's intentional, because it is not encouraged practice. You are usually better off using a pattern match with a typed pattern. That's particularly true if you need to do both a type test and a type cast, because both operations are then rolled into a single pattern match.

The second case of the previous match expression contains the type pattern `m: Map[_, _]`. This pattern matches any value that is a `Map` of some arbitrary key and value types and lets `m` refer to that value. Therefore, `m.size` is well-typed and returns the size of the map. The underscores in the type pattern are like wildcards in other patterns. You could have also used (lowercase) type variables instead.

Type erasure

Can you also test for a map with specific element types? This would be handy, say for testing whether a given value is a map from type `Int` to type `Int`. Let's try:

```
scala> def isIntIntMap(x: Any) = x match {  
|   case m: Map[Int, Int] => true  
|   case _ => false  
| }  
warning: there were unchecked warnings; re-run with  
-unchecked for details  
isIntIntMap: (Any)Boolean
```

The interpreter emitted an “unchecked warning”. You can find out details by starting the interpreter again with the `-unchecked` command-line option:

```
scala> :quit  
$ scala -unchecked  
Welcome to Scala version 2.7.0 (Java HotSpot(TM) Client VM,
```

Java 1.5.0_13).

Type in expressions to have them evaluated.

Type :help for more information.

```
scala> def isIntIntMap(x: Any) = x match {  
|   case m: Map[Int, Int] => true  
|   case _ => false  
| }  
<console>:5: warning: non variable type-argument Int in  
type pattern is unchecked since it is eliminated by erasure  
           case m: Map[Int, Int] => true  
                           ^
```

In fact, Scala uses the *erasure* model of generics, just like Java does. This means that no information about type arguments is maintained at runtime. Consequently, there is no way to determine at runtime whether a given Map object has been created with two Int arguments, rather than with arguments of different types. All the system can do is determine that a value is a Map of some arbitrary type parameters. You can verify this behavior by applying isIntIntMap to arguments of different instances of class Map:

```
scala> isIntIntMap(Map(1 -> 1))  
res15: Boolean = true  
  
scala> isIntIntMap(Map("abc" -> "abc"))  
res16: Boolean = true
```

The first application returns true, which looks correct, but the second application also returns true, which might be a surprise. To alert you to the possibly non-intuitive runtime behavior, the compiler emits unchecked warnings like the one shown above.

The only exception to the erasure rule concerns arrays, because these are handled specially in Java as well as in Scala. The element type of an array is stored with the array value, so you can pattern match on it. Here's an example:

```
scala> def isStringArray(x: Any) = x match {  
|   case a: Array[String] => "yes"  
|   case x: AnyRef => "no"  
| }
```

```
isStringArray: (Any)java.lang.String
scala> val as = Array("abc")
as: Array[java.lang.String] = Array(abc)
scala> isStringArray(as)
res17: java.lang.String = yes
scala> val ai = Array(1, 2, 3)
ai: Array[Int] = Array(1, 2, 3)
scala> isStringArray(ai)
res18: java.lang.String = no
```

Variable binding

In addition to the standalone variable patterns, you can also add a variable to any other pattern. You simply write the variable name, an at sign @, and then the pattern. This gives you a variable-binding pattern. The meaning of such a pattern is to perform the pattern match as normal, and if the pattern succeeds, set the variable to the matched object just as with a simple variable pattern.

As an example, here is a pattern match that looks for the absolute value operation being applied twice in a row. Such an expression can be simplified to only take the absolute value one time.

```
case UnOp("abs", e@UnOp("abs", _)) => e
```

In this example, there is a variable-binding pattern with `e` as the variable and `UnOp("abs", _)` as the pattern. If the entire pattern match succeeds, then the part that matched the `UnOp("abs", _)` part is made available as variable `e`. As the code is written, `e` then gets returned as is.

12.3 Pattern guards

Sometimes, syntactic pattern matching is not precise enough. For instance, say you are given the task of formulating a simplification rule that replaces sum expressions with two identical operands such as $e + e$ by multiplications of two, e.g. $e * 2$. In the language of Expr trees, an expression like

```
BinOp("+", Var("x"), Var("x"))
```

would be transformed by this rule to

```
BinOp("*", Var("x"), Number(2))
```

You might try to define this rule as follows:

```
scala> def simplifyAdd(e: Expr) = e match {
|   case BinOp("+", x, x) => BinOp("*", x, Number(2))
|   case _ => e
| }
<console>:13: error: x is already defined as value x
           case BinOp("+", x, x) => BinOp("*", x, Number(2))
                           ^
```

This fails, because Scala restricts patterns to be *linear*: a pattern variable may only appear once in a pattern. However, you can re-formulate the match with a *pattern guard*:

```
scala> def simplifyAdd(e: Expr) = e match {
|   case BinOp("+", x, y) if x == y =>
|     BinOp("*", x, Number(2))
|   case _ =>
|     e
| }
simplifyAdd: (Expr)Expr
```

A pattern guard comes after a pattern and starts with an `if`. The guard can be an arbitrary boolean expression, which typically refers to variables in the pattern. If a pattern guard is present, the match succeeds only if the guard evaluates to `true`. Hence, the first case above would only match binary operations with two equal operands.

Other examples of guarded patterns are

```
case n: Int if 0 < n => ...
// match only positive integers

case s: String if s.charAt(0) == 'a' => ...
// match only strings starting with the letter 'a'
```

12.4 Pattern overlaps

Patterns are tried in the order in which they are written. The following version of `simplify` presents an example where this order matters.

```
def simplifyAny(expr: Expr) = expr match {
    case UnOp("-", UnOp("-", e)) => e    // '-' is its own inverse
    case BinOp("+", e, Number(0)) => e    // '0' is a neutral element for '+'
    case BinOp("*", e, Number(1)) => e    // '1' is a neutral element for '*'
    case UnOp(op, e) => UnOp(op, simplifyAny(e))
    case BinOp(op, l, r) => BinOp(op, simplifyAny(l), simplifyAny(r))
    case _ => expr
}
```

This version of `simplify` will apply simplification rules anywhere in an expression, not just at the top, as `simplifyTop` did. It can be derived from `simplifyTop` by adding two more cases for general unary and binary expressions (cases four and five in the example above).

The fourth case has the pattern `UnOp(op, e)`; *i.e.* it matches every unary operation. The operator and operand of the unary operation can be arbitrary. They are bound to the pattern variables `op` and `e`, respectively. The alternative in this case applies `simplifyAny` recursively to the operand `e` and then re-builds the same unary operation with the (possibly) simplified operand. The fifth case for `BinOp` is analogous; it is a “catch-all” case for arbitrary binary operations, which recursively applies the simplification method to its operands.

In this example, it is important that the “catch-all” cases come *after* the more specific simplification rules. If you wrote them in the other order, then the catch-all case would be run in favor of the more specific rules. In many cases, the compiler will even complain if you try. For example:

```
scala> def simplifyBad(expr: Expr) = expr match {
    | case UnOp(op, e) => UnOp(op, simplifyBad(e))
    | case UnOp("-", UnOp("-", e)) => e
    | }
<console>:20: error: unreachable code
           case UnOp("-", UnOp("-", e)) => e
                                         ^
```

12.5 Sealed classes

Whenever you write a pattern match, you need to make sure you have covered all of the possible cases. Sometimes you can do this by adding a default case at the end of the match, but that only applies if there is a sensible default behavior. What do you do if there is no default? How can you ever feel safe that you covered all the cases?

In fact, you can enlist the help of the Scala compiler in detecting missing combinations of patterns in a match expression. To be able to do this, the compiler needs to be able to tell which are the possible cases. In general, this is impossible in Scala, because new case classes can be defined at any time and in arbitrary compilation units. For instance, nobody would prevent you from adding a fifth case class to the `Expr` class hierarchy in a different compilation unit from the one where the other four cases are defined.

The alternative is to make the superclass of your case classes *sealed*. A sealed class cannot have any new subclasses added except the ones in the same file. This is very useful for pattern matching, because it means you only need to worry about the subclasses you already know about. What's more, you get better compiler support as well. If you match against case classes that inherit from a sealed class, the compiler will flag missing combinations of patterns with a warning message.

Therefore, if you write a hierarchy of classes intended to be pattern matched, you should consider sealing them. Simply put the `sealed` keyword in front of the class at the top of the hierarchy. Programmers using your class hierarchy will then feel confident in pattern matching against it. The `sealed` keyword, therefore, is often a license to pattern match.

To experiment with sealed classes, you could turn the root `Expr` of the arithmetic expression example defined previously into a sealed class:

```
sealed abstract class Expr {}
```

The four case classes `Var`, `Number`, `UnOp`, and `BinOp` can stay as they are. Now define a pattern match where some of the possible cases are left out:

```
def describe(e: Expr): String = e match {
    case Number(x) => "a number"
    case Var(_)      => "a variable"
}
```

You will get a compiler warning like the following:

```
warning: match is not exhaustive!
missing combination      UnOp
missing combination      BinOp
```

The warning tells you that there's a risk your code might produce a `MatchError` exception because some possible patterns (`UnOp`, `BinOp`) are not handled. The warning points to a potential source of runtime faults, so it is usually a welcome help in getting your program right.

However, at times you might encounter a situation where the compiler is too picky in emitting the warning. For instance, you might know from the context that you will only ever apply the `describe` method above to expressions that are either `Numbers` or `Vars`. So you know that in fact no `MatchError` will be produced. To make the warning go away, you could add a third catch-all case to the method, like this:

```
def describe(e: Expr): String = (e: @unchecked) match {
    case Number(x) => "a number"
    case Var(_)      => "a variable"
    case _            => throw new RuntimeException // Should not happen
}
```

That works, but it is not ideal. You will probably not be very happy that you were forced to add code that will never be executed (or so you think), just to make the compiler shut up.

A more lightweight alternative is to add an `@unchecked` annotation to the selector expression of the match. This is done as follows.

```
def describe(e: Expr): String = (e: @unchecked) match {
    case Number(x) => "a number"
    case Var(_)     => "a variable"
}
```

Annotations are described in [Chapter 26](#). In general, you can add an annotation to an expression in the same way you add a type: Follow the expression with a colon and the name of the annotation. The `@unchecked` annotation has a special meaning for pattern matching. If a match selector expression carries this annotation, exhaustivity checking for the patterns that follow will be suppressed.

12.6 The Option type

Scala has a standard type named `Option` for optional values. Such a value can be of two forms: It can be of the form `Some(x)` where `x` is the actual value. Or it can be the `None` object, which represents a missing value.

Optional values are produced by some of the standard operations on Scala's collections. For instance, the `get` method of a `Map` produces `Some(value)` if a value corresponding to a given key has been found, or `None` if the given key is not defined in the `Map`. Here's an example:

```
scala> val capitals =
|   Map("France" -> "Paris", "Japan" -> "Tokyo")
capitals:
scala.collection.immutable.Map[java.lang.String,java.lang.String]
= Map(France -> Paris, Japan -> Tokyo)

scala> capitals get "France"
res19: Option[java.lang.String] = Some(Paris)

scala> capitals get "North Pole"
res20: Option[java.lang.String] = None
```

The most common way to take optional values apart is through a pattern match. For instance:

```
scala> def show(x: Option[String]) = x match {
|   case Some(s) => s
|   case None => "?"
| }
show: (Option[String])String

scala> show(capitals get "Japan")
res21: String = Tokyo

scala> show(capitals get "North Pole")
res22: String = ?
```

The `Option` type is used frequently in Scala programs. Compare this to the dominant idiom in Java of using `null` to indicate no value. For example, the `get` method of `java.util.HashMap` returns either a value stored in the `HashMap`, or `null` if no value was found. This approach works, but is error

prone, because it is difficult in practice to keep track of which variables in a program are allowed to be null. If a variable is allowed to be null, then you must remember to check it for null every time you use it. When you forget to check, you open the possibility that a `NullPointerException` may result at runtime. Because such exceptions may not happen very often, it can be difficult to discover the bug during testing.

By contrast, Scala encourages the use of `Option` to indicate an optional value. This approach to optional values has several advantages over Java's. First, it is far more obvious to readers of code that a variable whose type is `Option[String]` is an optional `String` than a variable of type `String`, which may sometimes be null. But most importantly, that programming error described earlier of using a variable that may be null without first checking it for null becomes in Scala a type error. If a variable is of type `Option[String]` and you try to use it as a `String`, your Scala program will not compile.

12.7 Patterns everywhere

Patterns are allowed in many parts of Scala, not just in standalone match expressions. Take a look at some other places you can use patterns.

Variable definitions

Any time you define a `val` or a `var`, you can use a pattern instead of a simple identifier. For example, you can use this to take apart a tuple and assign each of its parts to its own variable:

```
scala> val mytuple = (123, "abc")
mytuple: (Int, java.lang.String) = (123,abc)
scala> val (number, string) = mytuple
number: Int = 123
string: java.lang.String = abc
```

This construct is quite useful when working with case classes. If you know the precise case class you are working with, then you can deconstruct it with a pattern.

```
scala> val exp = new BinOp("*", Number(5), Number(1))
```

```
exp: BinOp = BinOp(*,Number(5.0),Number(1.0))
scala> val BinOp(op, left, right) = exp
op: String = *
left: Expr = Number(5.0)
right: Expr = Number(1.0)
```

Functions

A match expression can be used anywhere a function literal can be used. Essentially, a match expression *is* a function literal, only more general. Instead of having a single entry point and list of parameters, a match expression has multiple entry points, each with their own list of parameters. Each case clause is an entry point to the function, and the parameters are specified with the pattern. The body of each entry point is the right-hand side of the case clause.

Here is a very simple example:

```
scala> val increase: Int=>Int = { case x: Int => x + 5 }
increase: (Int) => Int = <function>
scala> increase(12)
res23: Int = 17
```

This match expression has only one case clause. That case clause binds variable x on its left-hand side. Its body, x+5, is allowed to reference x, just like the body of $y \Rightarrow y+5$ can access y.

This facility is quite useful for the actors library, described in [Chapter 23](#). Here is a typical piece of code, where a pattern match is passed directly to the react method:

```
react {
  case (name: String, actor: Actor) => {
    actor ! getip(name)
    act()
  }
  case msg => {
    println("Unhandled message: " + msg)
    act()
  }
}
```

```
    }  
}
```

One other generalization is worth noting: a match expression gives you a *partial* function. If you apply such a function on a value it does not support, it will generate a run-time exception. For example, here is a partial function that returns the second element of a list of integers.

```
val second: List[Int]=>Int = {  
    case x::y::_ => y  
}
```

When you run this, the compiler will correctly complain that the match is not exhaustive:

```
<console>:17: warning: match is not exhaustive!  
missing combination           Nil
```

This function will succeed if you pass it a three-element list, but not if you pass it an empty list:

```
scala> second(List(5,6,7))  
res24: Int = 6  
  
scala> second(List())  
scala.MatchError: List()  
        at $anonfun$1.apply(<console>:17)  
        at $anonfun$1.apply(<console>:17)
```

If you want to check whether a partial function is defined, you must first tell the compiler that you know you are working with partial functions. The type `List[Int]=>Int` includes all functions from lists of integers to integers, whether or not they are partial. The type that only includes *partial* functions from lists of integers to integers is written `PartialFunction[List[Int],Int]`. Here is the second function again, this time written with a partial function type.

```
val second: PartialFunction[List[Int],Int] = {  
    case x::y::_ => y  
}
```

Partial functions have a method `isDefinedAt` that can be used to test whether the function is defined at a particular element. In this case, the function is defined for any list that has at least two elements.

```
scala> second.isDefinedAt(List(5,6,7))
res24: Boolean = true

scala> second.isDefinedAt(List())
res25: Boolean = false
```

In general, you should try to work with complete functions whenever possible, because using partial functions allows for run-time errors that the compiler cannot help you with. Sometimes partial functions are really helpful, though. You might be sure that an unhandled value will never be supplied. Alternatively, you might be using a framework that expects partial functions and so will always check `isDefinedAt` before calling the function. An example of the latter is the `react` example given above, where the argument is a partially defined function, defined precisely for those messages that the caller wants to handle.

For expressions

You can also use a pattern in the generator of a for expression. For instance:

```
scala> for ((country, city) <- capitals)
    |   println("the capital of "+country+" is "+city)
the capital of France is Paris
the capital of Japan is Tokyo
```

The for expression above retrieves all key/value pairs from the `capitals` map. Each pair is matched against the pattern `(country, city)`, which defines the two variables `country` and `city`.

The pair pattern above was special because the match against it can never fail. Indeed, `capitals` yields a sequence of pairs, so you can be sure that every generated pair can be matched against a pair pattern. But it is equally possible that a pattern might not match a generated value. Here's an example where that is the case:

```
scala> val results = List(Some("apple"), None, Some("orange"))
results: List[Option[java.lang.String]] = List(Some(apple),
```

```
None, Some(orange))  
scala> for (Some(fruit) <- results) println(fruit)  
apple  
orange
```

As you can see from this example, generated values that do not match the pattern are discarded. For instance, the second element `None` in the `results` list does not match the pattern `Some(fruit)`; therefore it does not show up in the output.

12.8 A larger example

After having learned the different forms of patterns, you might be interested in seeing them applied in a larger example. The proposed task is to write an expression formatter class that displays an arithmetic expression in a two-dimensional layout. Divisions such as $x / x + 1$ should be printed vertically, by placing the numerator on top of the denominator, like this:

$$\begin{array}{c} x \\ \hline x + 1 \end{array}$$

As another example, here's the expression $((a / (b * c) + 1 / n) / 3)$ in two dimensional layout:

$$\begin{array}{r} a \quad 1 \\ \hline b * c \quad n \\ \hline 3 \end{array}$$

From these examples it looks like the class (let's call it `ExprFormatter`) will have to do a fair bit of layout juggling, so it makes sense to use the layout library developed in [Chapter 10](#):

```
class ExprFormatter {
```

A useful first step is to concentrate on horizontal layout. A structured expression like

```
BinOp("+",
  BinOp("*",
    BinOp("+", Var("x"), Var("y")),
    Var("z")),
  Number(1))
```

should print $(x + y) * z + 1$. Note that parentheses are mandatory around “ $x + y$,” but would be optional around “ $(x + y) * z$.” To keep the layout as legible as possible, your goal should be to omit parentheses wherever they are redundant, while ensuring that all necessary parentheses are present.

To know about where to put parentheses, one needs to know about the relative precedence of each operator, so it’s a good idea to tackle this first. You could express the relative precedence directly as a map literal of the form

```
Map(
  "|" -> 0, "||" -> 0,
  "&" -> 1, "&&" -> 1, ...
)
```

However, this involves some bit of pre-computation of precedences from the programmer’s part. A more convenient approach is to just define groups of operators of increasing precedence and then calculate the precedence of each operator from that. Here’s the code for that:

```
/** Contains all operators in groups of increasing precedence */
protected val opGroups =
  Array(
    Set("|", "||"),
    Set("&", "&&"),
    Set("^"),
    Set("==", "!="),
    Set("<", "<=", ">", ">="),
    Set("+", "-"),
    Set("*", "%")
  )

/** A mapping from operators to their precedence */
private val precedence = {
```

```
val assocs =  
  for {  
    i <- 0 until opGroups.length  
    op <- opGroups(i)  
  } yield op -> i  
Map() ++ assocs  
}
```

The precedence value is a Map from operators to their precedences (which are small integers starting with 0). It is calculated using a for expression with two generators. The first generator produces every index i of the `opGroups` array. The second generator produces every operator `op` in `opGroups(i)`. For each such operator the for expression yields an association from the operator `op` to its index i . Hence, the relative position of an operator in the array is taken to be its precedence. Associations are written with an infix arrow, e.g. `op -> i`. So far you have seen associations only as part of map constructions, but they are also values in their own right. In fact, the association `op -> i` is nothing else but the pair `(op, i)`.

Now that you have fixed the precedence of all binary operators except “`/`” it makes sense to generalize this concept to also cover unary operators. The precedence of a unary operator is higher than the precedence of every binary operator:

```
protected val unaryPrecedence = opGroups.length
```

The precedence of a fraction is treated differently from the other operators because fractions use vertical layout. However, it will prove convenient to assign to the division operator the special precedence value `-1`:

```
protected val fractionPrecedence = -1
```

After these preparations, you are ready to write the main format method. This method takes two arguments: an expression `e` of type `Expr`, together with the precedence `enclPrec` of the operator directly enclosing the expression `e` (if there’s no enclosing operator, `enclPrec` is supposed to be zero). The method yields a layout element which represents a two-dimensional array of characters. See [Chapter 10](#) for a description how layout elements are formed and what operations they provide.

Here's the `format` method in its entirety. Each of its cases will be discussed individually below.

```
private def format(e: Expr, enclPrec: Int): Element = e match {
    case Var(name) =>
        elem(name)
    case Number(num) =>
        def stripDot(s: String) =
            if (s endsWith ".") s.substring(0, s.length - 2) else s
        elem(stripDot(num.toString))
    case UnOp(op, arg) =>
        elem(op) beside format(arg, unaryPrecedence)
    case BinOp("/", left, right) =>
        val top = format(left, fractionPrecedence)
        val bot = format(right, fractionPrecedence)
        val line = elem('-', top.width max bot.width, 1)
        val frac = top above line above bot
        if (enclPrec != fractionPrecedence) frac
        else elem(" ") beside frac beside elem(" ")
    case BinOp(op, left, right) =>
        val opPrec = precedence(op)
        val l = format(left, opPrec)
        val r = format(right, opPrec + 1)
        val oper = l beside elem(" "+op+" ") beside r
        if (enclPrec <= opPrec) oper
        else elem("(") beside oper beside elem(")")
}
```

As expected, `format` proceeds by a pattern match on the kind of expression. There are five cases.

The first case is:

```
case Var(name) =>
    elem(name)
```

If the expression is a variable, the result is an element formed from the variable's name.

The second case is:

```

case Number(num) =>
def stripDot(s: String) =
  if (s endsWith ".0") s.substring(0, s.length - 2) else s
elem(stripDot(num.toString))

```

If the expression is a number, the result is an element formed from the number's value. The `stripDot` function cleans up the display of a floating-point number by stripping any `.0` suffix from a string.

The third case is:

```

case UnOp(op, arg) =>
  elem(op) beside format(arg, unaryPrecedence)

```

If the expression is a unary operation `UnOp(op, arg)` the result is formed from the operation `op` and the result of formatting the argument `arg` with the highest-possible environment precedence. This means that if `arg` is a binary operation (but not a fraction) it will always be displayed in parentheses.

The fourth case is:

```

case BinOp("/", left, right) =>
  val top = format(left, fractionPrecedence)
  val bot = format(right, fractionPrecedence)
  val line = elem('-', top.width max bot.width, 1)
  val frac = top above line above bot
  if (enclPrec != fractionPrecedence) frac
  else elem(" ") beside frac beside elem(" ")

```

If the expression is a fraction, an intermediate result `frac` is formed by placing the formatted operands `left` and `right` on top of each other, separated by an horizontal line element. The width of the horizontal line is the maximum of the widths of the formatted operands. This intermediate result is also the final result unless the fraction appears itself as an argument of another fraction. In the latter case, a space is added on each side of `frac`. To see the reason why, consider the expression `(a / b) / c`. Without the widening correction, formatting this expression would give

a
-

b
—
c

The problem with this layout is evident—it's not clear where the top-level fractional bar is. The expression above could mean either $(a / b) / c$ or else $a / (b / c)$. To disambiguate, a space should be added on each side to the layout of the nested fraction a / b . Then the layout becomes unambiguous:

a
—
b

c

The fifth and last case is:

```
case BinOp(op, left, right) =>
  val opPrec = precedence(op)
  val l = format(left, opPrec)
  val r = format(right, opPrec + 1)
  val oper = l beside elem(" "+op+" ") beside r
  if (enclPrec <= opPrec) oper
  else elem("(") beside oper beside elem(")")
```

This case applies for all other binary operations. Since it comes after the case starting with

```
case BinOp("/", left, right) => ...
```

you know that the operator op in the pattern `BinOp(op, left, right)` cannot be a division. To format such a binary operation, one needs to format first its operands `left` and `right`. The precedence parameter for formatting the left operand is the precedence `opPrec` of the operator `op`, while for the right operand it is one more than that. This scheme ensures that parentheses also reflect the correct associativity. For instance, the operation

```
BinOp("-", Var("a"), BinOp("-", Var("b"), Var("c"))))
```

would be correctly parenthesized as $a - (b - c)$. The intermediate result `oper` is then formed by placing the formatted left and right operands side-by-side, separated by the operator. If the precedence of the current operator is smaller than the precedence of the enclosing operator, `r` is placed between parentheses, otherwise it is returned as-is.

This finishes the design of the `format` function. For convenience, you can also add an overloaded variant which formats a top-level expression:

```
def format(e: Expr): Element = format(e, 0)
} // end ExprFormatter
```

Here's a test program which exercises the `ExprFormatter` class:

```
object Test extends Application {
    val f = new ExprFormatter
    val e1 = BinOp("*", BinOp("/", Number(1), Number(2)),
                  BinOp("+", Var("x"), Number(1)))
    val e2 = BinOp("+", BinOp("/", Var("x"), Number(2)),
                  BinOp("/", Number(1.5), Var("x")))
    val e3 = BinOp("/", e1, e2)
    def show(e: Expr) = println(e+":\n"+f.format(e)+"\n")
    for (val e <- Array(e1, e2, e3)) show(e)
}
```

Note that, even though this program does not define a `main` method, it is still a runnable application because it inherits from the `Application` trait. That trait simply defines an empty `main` method which gets inherited by the `Test` object. The actual work in the `Test` object gets done as part of the object's initialization, before the `main` method is run. That's why you can apply this trick only if your program does not take any command-line arguments. Once there are arguments, you need to write the `main` method explicitly. You can then run the `Test` program with the command:

```
scala Test
```

This should give the following output.

```
BinOp(*,BinOp(/,Number(1.0),Number(2.0)),BinOp(+,Var(x),  
Number(1.0))):  
1  
- * (x + 1)  
2  
  
BinOp(+,BinOp(/,Var(x),Number(2.0)),BinOp(/,Number(1.5),  
Var(x))):  
x 1.5  
- + ---  
2   x  
  
BinOp(/,BinOp(*,BinOp(/,Number(1.0),Number(2.0)),BinOp(+  
,Var(x),Number(1.0))),BinOp(+,BinOp(/,Var(x),Number(2.0)  
) ,BinOp(/,Number(1.5),Var(x))):  
1  
- * (x + 1)  
2  
-----  
x 1.5  
- + ---  
2   x
```

12.9 Conclusion

This chapter has described Scala’s case classes and pattern matching in detail. Using them, you can take advantage of several concise idioms not normally available in object-oriented languages.

Scala’s pattern matching goes further than this chapter describes, however. If you want to use pattern matching on one of your classes, but you do not want to open access to your classes the way case classes do, then you can use the *extractors* described in [Chapter 24](#).

Chapter 13

Packages and Imports

When working with large programs, there is a risk that programmers will either step on each other's toes with conflicting code changes, or be so afraid of that risk that they become mired in communication that attempts to prevent such conflicts. One way to reduce this problem is to write in a modular style. The program is divided into a number of smaller modules, each of which has an inside and an outside. Programmers working on the inside of a module—its *implementation*—then only need to coordinate with other programmers working on that very same module. Only when there are changes to the outside of a module—its *interface*—is it necessary to coordinate with developers working on other modules. Interface and implementation were discussed in [Chapter 4](#) for classes, but the concept applies just as well to packages.

This chapter shows several constructs that help you program in a modular style. It shows how to place things in packages, how to make names visible through imports, and how to control the visibility of definitions through access modifiers. The constructs are similar in spirit with constructs in Java, but there are some differences—usually ways that are more consistent—so it is worth reading this chapter even if you already know Java.

Looking ahead, [Chapter 25](#) shows some additional techniques to make your code modular that are distinctly Scala. Before getting to that, though, take a look now at the Java-like techniques that are available.

13.1 Packages

Packages in Scala are similar to packages in Java. There is a global hierarchy of packages. You can place the contents of an entire file into one of these packages by putting a package clause at the top of the file.

```
package bobsrockets.navigation
class Navigator { ... }
```

In the above example, class `Navigator` goes into the package `bobsrockets.navigation`. Presumably, this is the navigation software developed by Bob's Rockets, Inc.

Scala also supports a syntax more like C# namespaces where a package clause is followed by a section in curly braces which contains the definitions that go into the package. Among other things, this syntax lets you put different parts of a file into different packages. For example, you might include a class's tests in the same file as the original code, but put the tests in a different package, as shown in [Figure 13.1](#).

In [Figure 13.1](#), object `NavigatorTest` goes into package `bobsrockets.tests`, and class `Navigator` goes into `bobsrockets.navigation`. In fact, the first Java-like syntax is just syntactic sugar for the more general nested syntax. So the following three versions of `bobsrockets.navigation.Navigator` are all equivalent:

```
// Java-like package clause
package bobsrockets.navigation
class Navigator { ... }

// Namespace-like package
package bobsrockets.navigation {
    class Navigator { ... }
}

// Nested namespace-like package
package bobsrockets {
    package navigation {
        class Navigator { ... }
    }
}
```

```
package bobsrockets {  
    package navigation {  
        class Navigator { ... }  
    }  
    package tests {  
        object NavigatorTest {  
            val x = new navigation.Navigator  
            ...  
        }  
    }  
}
```

Figure 13.1: Scala packages nest. Code inside `NavigatorTest` can access package `navigation` directly, instead of needing to write `bobsrockets.navigation`.

As this notation hints, Scala’s packages truly nest. That is, package `navigation` is semantically *inside* of package `bobsrockets`. Java packages, despite being hierarchical, do not nest. In Java, whenever you name a package, you have to start at the root of the package hierarchy. Scala uses a more regular rule in order to simplify the language.

Take a closer look at Figure 13.1. Inside the `NavigatorTest` object, it is not necessary to reference `Navigator` as `bobsrockets.navigation.Navigator`, its fully qualified name. Since packages nest, it can be referred to as simply as `navigation.Navigator`. This shorter name is possible because class `NavigatorTest` is contained in package `bobsrockets`, which has `navigation` as a member. Therefore, `navigation` can be referred to without prefix, just like the methods of a class can refer to other methods of a class without a prefix.

Another consequence of Scala’s scoping rules is that packages in some inner scope hide packages of the same name that are defined in an outer scope. For instance, consider the following code:

```
package bobsrockets {  
    package navigation {
```

```
package tests {  
    object Test1 { ... }  
}  
////// how to access Test1, Test2, Test3 here?  
}  
package tests {  
    object Test2 { ... }  
}  
}  
package tests {  
    object Test3 { ... }  
}
```

There are three packages here named `tests`. One is in package `bobsrockets.navigation`, one is in `bobsrockets`, and one is at the top level. Such repeated names work fine—after all they are a major reason to use packages!—but they do mean you must use some care to access precisely the one you mean.

To see how to choose the one you mean, take a look at the line marked `//////` above. How would you reference each of `Test1`, `Test2`, and `Test3`? Accessing the first one is easiest. A reference to `tests` by itself will get you to package `bobsrockets.navigation.tests`, because that is the `tests` package that is defined in the closest enclosing scope. Thus, you can refer to the first test class as simply `tests.Test1`. Referring to the second one also is not tricky. You can write `bobrockets.tests.Test2` and be clear about which one you are referencing. That leaves the question of the third test class, however. How can you access `Test3`, considering that there is a nested `tests` package shadowing the top-level one?

To help in this situation, Scala provides a package named `_root_` that is outside any package a user can write. Put in other words, every top-level package you can write is treated as a member of package `_root_`. Thus, `_root_.tests` gives you the top-level `tests` package, and `_root_.tests.Test3` designates the outermost test class.

13.2 Imports

As in Java, packages and their members can be imported using `import` clauses. Imported items can then be accessed by a single identifier like `File`, as opposed to requiring a qualified name like `java.io.File`.

Scala's `import` clauses are quite a bit more flexible than Java's. There are three principal differences. In Scala, imports may appear anywhere, they may refer to singleton objects in addition to packages, and they let you rename and hide some of the imported members. The rest of this section explains the details. Assume for the discussion the following code which defines some kinds of fruit:

```
package bobsdelights
trait Fruit {
    val name: String
    val color: Color
}
object Fruits {
    object Apple extends Fruit { ... }
    object Orange extends Fruit { ... }
    object Pear extends Fruit { ... }
    val menu = List(Apple, Orange, Pear)
}
```

An `import` clause makes members of a package or object available by their names alone without needing to prefix them by the package or object. Here are some simple examples:

```
import bobsdelights.Fruit      // easy access to Fruit
import bobsdelights._          // easy access to all members of bobsdelights
import bobsdelights.Fruits._   // easy access to all members of Fruits
```

The first of these corresponds to Java's single type import, the second to Java's "on demand" import. The only difference is that Scala's on demand imports are written with a trailing under-bar '`_`' instead of an asterisk '`*`' (after all, `*` is a valid identifier in Scala!). The third import clause above corresponds roughly to Java's import of static class fields, but it is more general.

Imports in Scala can appear anywhere, not just at the beginning of a compilation unit. They can refer to arbitrary values. For instance, the following is possible:

```
def showFruit(f: Fruit) {  
    import f._  
    println(name+s are "+color)  
}
```

Here, method `showFruit` imports all members of its parameter `f`, which is of type `Fruit`. The subsequent `println` statement can refer to `name` and `color` directly. These two references are equivalent to `f.name` and `f.color`. This syntax is particularly useful when you use objects as modules, as described in [Chapter 25](#).

Imports can import packages themselves, not just their non-package members. This is only natural if you think of nested packages being contained in their surrounding package.

```
import java.util.regex  
regex.Pattern.compile("a*b") // accesses java.util.regex.Pattern
```

Imports in Scala can also rename or hide members. This is done with an *import selector clause* enclosed in braces which follows the object from which members are imported. Here are some examples:

```
import Fruits.{Apple, Orange}
```

This imports just the two members `Apple` and `Orange` from object `Fruits`.

```
import Fruits.{Apple => McIntosh, Orange}
```

This imports the two members `Apple` and `Orange` from object `Fruits`. However, the `Apple` object is renamed to `McIntosh`. So this object can be accessed with either `Fruits.Apple` or `McIntosh`. A renaming clause is always of the form `<original-name> => <new-name>`.

```
import java.sql.{Date=>SDate}
```

This imports the SQL date class as `SDate`, so that you can simultaneously import the normal Java date class as simply `Date`.

```
import java.{sql=>S}
```

This imports the SQL package as S, so that you can write things like S.Date.

```
import Fruits.{_}
```

This imports all members from object Fruits, just as import Fruits._ does.

```
import Fruits.{Apple => McIntosh, _}
```

This imports all members from object Fruits but renames Apple to McIntosh.

```
import Fruits.{Pear => _, _}
```

This imports all members *except* Pear. A clause of the form <original-name> => _ excludes <original-name> from the names that are imported. In a sense, renaming something to _ means hiding it altogether.

These examples demonstrate the great flexibility Scala offers when it comes to importing members selectively and possibly under different names. In summary, an import selector can consist of the following:

- A simple name x. This includes x in the set of imported names.
- A renaming clause x => y. This makes the member named x visible under the name y.
- A hiding clause x => _. This excludes x from the set of imported names.
- A “catch-all” _. This imports all members except those members mentioned in a preceding clause. If a catch-all is given, it must come last in the list of import selectors.

The simpler import clauses shown at the beginning of this section can be seen as special abbreviations of import clauses with a selector clause: `import p._` is equivalent to `import p.{_}` and `import p.n` is equivalent to `import p.{n}`.

13.3 Access modifiers

Members of packages, classes or objects can be labeled with the access modifiers `private` and `protected`. These modifiers restrict accesses to the members to certain regions of code. Scala's treatment of access modifiers roughly follows Java's but there are some important differences which are explained in the following.

Private members

Private members are treated similarly to Java. A member labeled `private` is visible only inside the class or object that contains the member definition. In Scala, this rule applies also for inner classes. This treatment is more consistent, but differs from Java. Consider this example:

```
class Outer {  
    class Inner {  
        private def f() { println("f") }  
        class InnerMost {  
            f() // OK  
        }  
    }  
    (new Inner).f() // error: 'f' is not accessible  
}
```

In Scala, the access `(new Inner).f()` is illegal because `f` is declared `private` in `Inner` and the access is not from within class `Inner`. By contrast, the first access to `f` in class `InnerMost` is OK, because that access is contained in the body of class `Inner`. Java would permit both accesses because it lets an outer class access private members of its inner classes.

Protected members

Access to `protected` members is a bit more restrictive than in Java. In Scala, a `protected` member is only accessible from subclasses of the class in which the member is defined. In Java such accesses are also possible from other classes in the same package. In Scala, there is another way to achieve this effect, as described below, so `protected` is free to be left as is. The following example illustrates protected accesses.

```
package p {  
    class Super {  
        protected def f() { println("f") }  
    }  
    class Sub extends Super {  
        f()  
    }  
    class Other {  
        (new Super).f() // error: 'f' is not accessible  
    }  
}
```

Here, the access to `f` in class `Sub` is OK because `f` is declared `protected` in `Super` and `Sub` is a subclass of `Super`. By contrast the access to `f` in `Other` is not permitted, because `Other` does not inherit from `Super`. In Java, the latter access would be still permitted because `Other` is in the same package as `Sub`.

Public members

Every member not labeled `private` or `protected` is assumed to be `public`. There is no explicit modifier for public members. Such members can be accessed everywhere.

Scope of protection

Access modifiers in Scala can be augmented with qualifiers. A modifier of the form `private[X]` or `protected[X]` means that access is private or protected “up to” `X`, where `X` designates some enclosing package, class or object.

Qualified access modifiers give you very fine-grained control over visibility. In particular they enable you to express Java’s accessibility notions such as package private, package protected, or private up to outermost class, which are not directly expressible with simple modifiers in Scala. But they also let you express accessibility rules which cannot be expressed in Java. [Figure 13.2](#) presents an example with many access qualifiers being used.

In this figure, class `Navigator` is labeled `private[bobsrockets]`. This means that this class is visible in all classes and objects that are contained

```
package bobsrockets {  
    package navigation {  
        private[bobsrockets] class Navigator {  
            protected[navigation] def useChart(sc: StarChart) ...  
            class LegOfJourney {  
                private[Navigator] length = ...  
            }  
            private[this] current: xml.Node = ...  
        }  
    }  
    package tests {  
        import navigation._  
        object Test {  
            private[tests] val navigator = new Navigator  
            ...  
        }  
    }  
}
```

Figure 13.2: Access qualifiers

in package `bobsrockets`. In particular, the access to `Navigator` in object `Test` is permitted, because `Test` is contained in `bobsrockets`. On the other hand, code outside the package `bobsrockets` cannot access class `Navigator`.

This technique is quite useful in large projects which span several packages. It allows you to define things that are visible in several sub-packages of your project but that remain hidden from clients external to your project. The same technique is not possible in Java. There, once a definition escapes its immediate package boundary, it is visible to the world at large.

Of course, the qualifier of a `private` may also be the directly enclosing package. An example is the access modifier of `navigator` in object `Test` in Figure 13.2. Such an access modifier is equivalent to Java's package-private access.

All qualifiers can also be applied to `protected`, with the same meaning as `private`. That is, a modifier `protected[X]` in a class `C` allows access to the labeled definition in all subclasses of `C` and also within the enclosing package, class, or object `X`. For instance, the `useChart` method in [Figure 13.2](#) is accessible in all subclasses of `Navigator` and also in all code contained in the enclosing package `navigation`. It thus corresponds exactly to the meaning of `protected` in Java.

The qualifiers of `private` can also refer to an enclosing class or object. For instance the `length` method in class `LegOfJourney` in [Figure 13.2](#) is labeled `private[Navigator]`, so it is visible from everywhere in class `Navigator`. This gives the same access capabilities as for private members of inner classes in Java. A `private[C]` where `C` is the directly enclosing class is the same as just `private` in Scala.

Finally, Scala also has an access modifier which is even more restrictive than `private`. A definition labeled `private[this]` is accessible only from within the same object that contains the definition. Such a definition is called *object-private*. For instance, the definition

```
private[this] current: xml.Node
```

in class `Navigator` in [Figure 13.2](#) is object-private. This means that any access must not only be within class `Navigator`, but it must also be made from the very same instance of `Navigator`. Thus the following two accesses are legal:

```
current  
this.current
```

The following access, though, is not allowed:

```
val other = new Navigator  
other.current
```

Marking a member `private[this]` is a guarantee that it will not be seen from other objects of the same class. This can be useful for documentation. It also sometimes lets you write more general variance annotations (see [Section 17.6](#) for details).

To summarize, the following table lists the effects of `private` qualifiers. Each line shows a qualified `private` modifier and what it would mean if such a modifier were added to a member of class `LegOfJourney` in [Figure 13.2](#).

private[_root_]	same as public access
private[bobsrockets]	access within outer package
private[navigation]	same as package visibility in Java
private[Navigator]	same as private in Java
private[LegOfJourney]	same as private in Scala
private[this]	access only from same object

Visibility and companion objects

In Java, static members and instance members belong to the same class, so access modifiers apply uniformly to them. You have already seen that in Scala there are no static members; instead one can have a companion object which contains members that exist only once. For instance, in the code below object Rocket is a companion of class Rocket.

```
class Rocket {  
    private def canGetHome = deltaV(fuel) < needed  
}  
object Rocket {  
    private def deltaV(fuel: Double) = ...  
  
    def chooseStrategy(rocket: Rocket) {  
        if (rocket.canGetHome)  
            goHome()  
        else  
            pickAStar()  
    }  
}
```

Scala's access rules privilege companion objects and classes when it comes to private or protected accesses. A class shares all its access rights with its companion object and *vice versa*. In particular, an object can access all private members of its companion class, just as a class can access all private members of its companion object.

For instance, the Rocket class above can access method deltaV, which is declared private in object Rocket. Analogously, the Rocket object can access the private method canGetHome in class Rocket.

One exception where this analogy between Scala and Java breaks down concerns protected static members. A protected static member of a Java

class C can be accessed in all subclasses of C. By contrast, a protected member in a companion object makes no sense, as objects are not inherited, so there can be no subclasses.

Conclusion

Now you have seen the basic constructs for dividing a program into packages. This gives you a simple and useful kind of modularity, so that you can work with very large bodies of code without different parts of the code trampling on each other. This system is the same in spirit as Java's packages, but as you have seen there are some differences where Scala chooses to be more consistent or more general.

Looking ahead, [Chapter 25](#) describes a more flexible module system than division into packages. In addition to letting you separate code into several namespaces, that approach allows modules to be parameterized and to inherit from each other.

Chapter 14

Working with Lists

Lists are probably the most commonly used data structure in Scala programs. This chapter explains lists in detail. It presents many common operations that can be performed on lists. It also teaches some important design principles for programs working on lists.

14.1 List literals

You have seen lists already in the preceding chapters, so you know that a list containing the elements 'a', 'b', 'c' is written `List('a', 'b', 'c')`. Here are some other examples:

```
val fruit = List("apples", "oranges", "pears")
val nums  = List(1, 2, 3, 4)
val diag3 = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty = List()
```

Lists are quite similar to arrays, but there are two important differences. First, lists are immutable. That is, elements of a list cannot be changed by assignment. Second, lists have a recursive structure, whereas arrays are flat.

14.2 The List type

Like arrays, lists are *homogeneous*. That is, the elements of a list all have the same type. The type of a list which has elements of type T is written

`List[T]`. For instance, here are the same four lists defined above with explicit types added:

```
val fruit: List[String]    = List("apples", "oranges", "pears")
val nums : List[Int]        = List(1, 2, 3, 4)
val diag3: List[List[Int]] = List(List(1, 0, 0),
                                    List(0, 1, 0),
                                    List(0, 0, 1))
val empty: List[Nothing]   = List()
```

The list type in Scala is *covariant*. This means that for each pair of types S and T, if S is a subtype of T then also `List[S]` is a subtype of `List[T]`. For instance, `List[String]` is a subtype of `List[Object]`. This is natural because every list of strings can also be seen as a list of objects.¹

Note that the empty list has type `List[Nothing]`. You have seen in [Section 10.17](#) that `Nothing` is the bottom type in Scala's class hierarchy. That is, it is a subtype of every other Scala type. Because lists are covariant, it follows that `List[Nothing]` is a subtype of `List[T]`, for any type T. So the empty list object, which has type `List[Nothing]`, can also be seen as an object of every other list type of the form `List[T]`. That's why it is permissible to write code like

```
// List() is also of type List[String]!
val xs: List[String] = List()
```

14.3 Constructing lists

All lists are built from two fundamental building blocks, `Nil` and ‘`::`’ (pronounced “cons”). `Nil` represents an empty list. The infix operator ‘`::`’ expresses list extension at the front. That is, `x :: xs` represents a list whose first element is `x`, which is followed by (the elements of) list `xs`. Hence, the previous list values above could also have been defined as follows:

```
val fruit  = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums   = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3 = (1 :: (0 :: (0 :: Nil))) ::
```

¹[Chapter 17](#) gives more details on variance, and how to specify it.

Table 14.1: Basic list operations.

What it is	What it does
<code>empty.isEmpty</code>	returns true
<code>fruit.isEmpty</code>	returns false
<code>fruit.head</code>	returns "apples"
<code>hline fruit.tail.head</code>	returns "oranges"
<code>diag3.head</code>	returns <code>List(1, 0, 0)</code>

```
(0 :: (1 :: (0 :: Nil))) ::  
(0 :: (0 :: (1 :: Nil))) :: Nil  
val empty = Nil
```

In fact the previous definitions of `fruit`, `nums`, `diag3`, and `empty` in terms of `List(...)` are just wrappers that expand to the definitions above. For instance, the invocation `List(1, 2, 3)` creates the list `1 :: (2 :: (3 :: Nil))`.

Because it ends in a colon, the ‘`::`’ operation associates to the right: `A :: B :: C` is interpreted as `A :: (B :: C)`. Therefore, you can drop the parentheses in the definitions above. For instance

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

is equivalent to the previous definition of `nums`.

14.4 Basic operations on lists

All operations on lists can be expressed in terms of the following three:

- `head` returns the first element of a list,
- `tail` returns the list consisting of all elements except the first element,
- `isEmpty` returns `true` if the list is empty

These operations are defined as methods of class `List`. You invoke them by selecting from the list that’s operated on. Some examples are shown in Table 14.1.

The `head` and `tail` methods are defined only for non-empty lists. When selected from an empty list, they throw an exception. For instance:

```
scala> Nil.head  
java.util.NoSuchElementException: head of empty list
```

As an example of how lists can be processed, consider sorting the elements of a list of numbers into ascending order. One simple way to do so is *insertion sort*, which works as follows: To sort a non-empty list `x :: xs`, sort the remainder `xs` and insert the first element `x` at the right position in the result. Sorting an empty list yields the empty list. Expressed as Scala code:

```
def isort(xs: List[Int]): List[Int] =  
  if (xs.isEmpty) Nil  
  else insert(xs.head, isort(xs.tail))  
  
def insert(x: Int, xs: List[Int]): List[Int] =  
  if (xs.isEmpty || x <= xs.head) x :: xs  
  else xs.head :: insert(x, xs.tail)
```

14.5 List patterns

Lists can also be taken apart using pattern matching. List patterns correspond one-by-one to list expressions. You can either match on all elements of a list using a pattern of the form `List(...)`. Or you take lists apart bit by bit using patterns composed from the ‘`::`’ operator and the `Nil` constant.

Here’s an example of the first kind of pattern:

```
scala> val List(a, b, c) = fruit  
a: java.lang.String = apples  
b: java.lang.String = oranges  
c: java.lang.String = pears
```

The pattern `List(a, b, c)` matches lists of length 3, and binds the three elements to the pattern variables `a`, `b`, `c`. If you don’t know the number of list elements beforehand, it’s better to match with `::` instead. For instance, the pattern `a :: b :: rest` matches lists of length 2 or greater:

```
scala> val a :: b :: rest = fruit
a: java.lang.String = apples
b: java.lang.String = oranges
rest: List[java.lang.String] = List(pears)
```

Taking lists apart with patterns is an alternative to taking them apart with the basic methods `head`, `tail`, and `isEmpty`. For instance, here's insertion sort again, this time written with pattern matching:

```
def isort(xs: List[Int]): List[Int] = xs match {
    case List()    => List()
    case x :: xs1 => insert(x, isort(xs1))
}

def insert(x: Int, xs: List[Int]): List[Int] = xs match {
    case List()  => List(x)
    case y :: ys => if (x <= y) x :: xs
                     else y :: insert(x, ys)
}
```

Often, pattern matching over lists is clearer than decomposing them with methods, so pattern matching should be a useful part of your list processing toolbox.

Aside: If you review the possible forms of patterns explained in [Chapter 12](#), you might find that neither `List(...)` nor `::` looks like it fits one of the cases of patterns defined there. In fact, `List(...)` is an instance of a library-defined extractor pattern. Such patterns will be treated in [Chapter 24](#). The “cons” pattern `x :: xs` is a special case of an infix operation pattern. You know already that, when seen as an expression, an infix operation is equivalent to a method call. For patterns, the rules are different: When seen as a pattern, an infix operation such as `p op q` is equivalent to `op(p, q)`. That is, the infix operator `op` is treated as a constructor pattern. In particular, a cons pattern such as `x :: xs` is treated as `::(x, xs)`. This hints that there should be a class named `::` that corresponds to the pattern constructor. Indeed there is such a class. It is named `scala.::` and is exactly the class that builds non-empty lists. So `::` exists twice in Scala, once as a name of a class in package `scala`, and another time as a method in class `List`. The effect of the method `::` is to produce an instance of the class `scala.::`. You'll find out more details about how the `List` class is implemented in [Chapter 20](#).

This is all you need to know about lists in Scala to be able to use them correctly. However, there is also a large number of methods that capture common patterns of operations over lists. These methods make list processing programs more concise and often also clearer. The next two sections present the most important methods defined in the `List` class. The presentation is split into two parts. First order methods are explained in the next section, higher-order methods in the one after that.

14.6 Operations on lists Part I: First-order methods

This section explains most first-order methods defined in the `List` class. First-order methods are methods that do not take functions as arguments. The section also introduces by means of two examples some recommended techniques to structure programs that operate on lists.

Concatenating lists

An operation similar to ‘`::`’ is list concatenation, written ‘`::::`’. Unlike ‘`::`’, ‘`::::`’ takes two lists as arguments. The result of `xs :::: ys` is a new list which contains all the elements of `xs`, followed by all the elements of `ys`. Here are some examples:

```
scala> List(1, 2) :::: List(3, 4, 5)
res0: List[Int] = List(1, 2, 3, 4, 5)

scala> List() :::: List(1, 2, 3)
res1: List[Int] = List(1, 2, 3)

scala> List(1, 2, 3) :::: List(4)
res2: List[Int] = List(1, 2, 3, 4)
```

Like `cons`, list concatenation associates to the right. An expression like this:

`xs :::: ys :::: zs`

is interpreted like this:

`xs :::: (ys :::: zs)`

The Divide and Conquer principle

Concatenation ‘`:::`’ is implemented as a method in class `List`. It would also be possible to implement concatenation “by hand”, using pattern matching on lists. It’s instructive to try to do that yourself. First, let’s settle on a signature for the concatenation method (let’s call it `append`). In order not to mixup things too much, assume that `append` is defined outside the `List` class. So it will take the two lists to be concatenated as parameters. These two lists must agree on their element type, but that element type can be arbitrary. This can be expressed by giving `append` a type parameter which represents the element type of the two input lists.

```
def append[T](xs: List[T], ys: List[T]): List[T] = ...
```

To design the implementation of `append`, it pays to remember the “divide and conquer” design principle for programs over recursive data structures such as lists. Many algorithms over lists first split an input list into simpler cases using a pattern match. That’s the *divide* part of the principle. They then construct a result for each case. If the result is a non-empty list, some of its parts may be constructed by recursive invocations of the same algorithm. That’s the *conquer* part of the principle.

Let’s apply this principle to the implementation of the `append` method. The first question to ask is on which list to match. This is less trivial in the case of `append` than for many other methods because there are two choices. However, the subsequent “conquer” phase tells you that you need to construct a list consisting of all elements of both input lists. Since lists are constructed from the start towards the end, and the first elements of the output list coincides with the elements of the first input `xs`, it makes sense to concentrate on this input as a source for a pattern match. The most common pattern match over lists simply distinguishes an empty from a non-empty list. So this gives the following outline of an `append` method:

```
def append[T](xs: List[T], ys: List[T]): List[T] = xs match {  
    case List() => ??  
    case x :: xs1 => ??  
}
```

All that remains is to fill in the two places marked with “`??`”. The first such place is the alternative where the input list `xs` is empty. In this case

concatenation yields the second list:

```
case List() => ys
```

The second place left open is the alternative where the input list `xs` consists of some head `x` followed by a tail `xs1`. In this case the result is also a non-empty list. To construct a non-empty list you need to know what the head and the tail of that list should be. You know that the first element of the result list is `x`. As for the remaining elements, these can be computed by appending the rest `xs1` of the first list to the second list `ys`. This completes the design and gives:

```
def append[T](xs: List[T], ys: List[T]): List[T] = xs match {
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
}
```

The computation of the second alternative illustrated the “conquer” part of the divide and conquer principle: Think first what the shape of the desired output should be, then compute the individual parts of that shape, using recursive invocations of the algorithm where appropriate. Finally, construct the output from these parts.

Taking the length of a list: `length`

The `length` method computes the length of a list.

```
scala> List(1, 2, 3).length
res3: Int = 3
```

In contrast to the situation with arrays, `length` is a relatively expensive operation on lists. It takes time proportional to the number of elements of the list. That’s why it’s not a good idea to replace a test such as `xs.isEmpty` by `xs.length == 0`. The two tests are equivalent, but the second one is slower, in particular if the list `xs` is long.

Accessing the end of a list: `init` and `last`

You know already the basic operations `head` and `tail`, which respectively take the first element of a list, and the rest of the list except the first element. They each have a dual operation:

`last` returns the last element of a (non-empty) list, whereas `init` returns a list consisting of all elements except the last one.

```
scala> val abcde = List('a', 'b', 'c', 'd', 'e')
abcde: List[Char] = List(a, b, c, d, e)
scala> abcde.last
res4: Char = e
scala> abcde.init
res5: List[Char] = List(a, b, c, d)
```

Like `head` and `tail`, these methods throw an exception when applied on an empty list:

```
scala> List().init
java.lang.UnsupportedOperationException: Nil.init
      at scala.List.init(List.scala:544)
      at ...
scala> List().last
java.util.NoSuchElementException: Nil.last
      at scala.List.last(List.scala:563)
      at ...
```

Unlike `head` and `tail`, which are both constant time, `init` and `last` need to traverse the spine of a list to compute their result. They therefore take time proportional to the length of the list. Consequently, it's a good idea to organize your data so that most accesses are to the head of a list, rather than the last element.

Reversing lists: `reverse`

If at some point in the computation an algorithm demands frequent accesses to the end of a list, it's sometimes better to reverse the list first and work with the result instead. Here's how to do the reversal:

```
scala> abcde.reverse
res6: List[Char] = List(e, d, c, b, a)
```

Note that, like all other list operations, `reverse` creates a new list rather than changing the one it operates on. Since lists are immutable, such a change

would not be possible, anyway. To verify this, check that the original value of `abcde` is unchanged after the reverse operation:

```
scala> abcde
res7: List[Char] = List(a, b, c, d, e)
```

The `reverse`, `init`, and `last` operations satisfy some laws which can be used for reasoning about computations and for simplifying programs.

1. `reverse` is its own inverse:

```
xs.reverse.reverse = xs
```

2. `reverse` turns `init` to `tail` and `last` to `head`:

```
xs.reverse.init == xs.tail
xs.reverse.tail == xs.init
xs.reverse.head == xs.last
xs.reverse.last == xs.head
```

`Reverse` could be implemented using concatenation ‘`:::`’, like in the following method `rev`:

```
def rev[T](xs: List[T]): List[T] = xs match {
    case List() => xs
    case x :: xs1 => rev(xs1) :: List(x)
}
```

However, this method is less efficient than one would hope for. To study the complexity of `rev`, assume that the list `xs` has length `n`. Notice that there are `n` recursive calls to `rev`. Each call except the last involves a list concatenation `xs :: ys` takes time proportional to the length of its first argument `xs`. Hence, the total complexity of `xs` is

$$n + (n - 1) + \dots + 1 = (1 + n) * n / 2$$

In other words, `rev`'s complexity is quadratic in the length of its input argument. This is disappointing when comparing to the standard reversal of a mutable, linked list, which has linear complexity. However, the current implementation of `rev` is not the best one can do. You will see in [Section 14.7](#) how to speed it up.

Prefixes and suffixes: drop, take and splitAt

The drop and take operations generalize tail and init in that they take arbitrary prefixes or suffixes of a list. The expression `xs take n` returns the first n elements of the list `xs`. If $n > xs.length$, the whole list `xs` is returned. The operation `xs drop n` returns all elements of list `xs` except the first n ones. If $n > xs.length$, the empty list is returned.

The splitAt operation splits the list at a given index, returning a pair of two lists. It is defined by the equality

```
xs splitAt n = (xs take n, xs drop n)
```

However, `splitAt` avoids traversing the list `xs` twice. Here are some examples:

```
scala> abcde take 2
res8: List[Char] = List(a, b)
scala> abcde drop 2
res9: List[Char] = List(c, d, e)
scala> abcde splitAt 2
res10: (List[Char], List[Char]) = (List(a, b),List(c, d, e))
```

Element selection: apply and indices

Random element selection is supported through the `apply` method; however it is a less common operation for lists than it is for arrays.

```
scala> abcde apply 2
res11: Char = c
```

As for all other types, `apply` is implicitly inserted when an object appears in the function position in a method call, so the line above can be shortened to:

```
scala> abcde(2)
res12: Char = c
```

One reason why random element selection is less popular for lists than for arrays is that `xs(n)` takes time proportional to the index n . In fact, `apply` is simply defined by a combination of `drop` and `head`:

```
xs apply n  =  (xs drop n).head
```

This definition also makes clear that list indices range from 0 up to the length of the list minus one. The `indices` method returns a list consisting of all valid indices of a given list:

```
scala> abcde.indices
res13: List[Int] = List(0, 1, 2, 3, 4)
res50: List[Int] = List(0, 1, 2, 3, 4)
```

Zipping lists: `zip`

The `zip` operation takes two lists and forms a list of pairs:

```
scala> abcde.indices zip abcde
res14: List[(Int, Char)] = List((0,a), (1,b), (2,c), (3,d),
(4,e))
```

If the two lists are of different length, any unmatched elements are dropped:

```
scala> val zipped = abcde zip List(1, 2, 3)
zipped: List[(Char, Int)] = List((a,1), (b,2), (c,3))
```

A useful special case is to zip a list with its index. This is done by the `zipWithIndex` method, which pairs every element of a list with the position where it appears in the list.

```
scala> abcde.zipWithIndex
res15: List[(Char, Int)] = List((a,0), (b,1), (c,2), (d,3),
(e,4))
```

Displaying lists: `toString` and `mkString`

The `toString` operation returns the canonical string representation of a list:

```
scala> abcde.toString
res16: String = List(a, b, c, d, e)
```

If you want a different representation you can use the `mkString` method. The operation `xs mkString (pre, sep, post)` takes four arguments: The list `xs` to be displayed, a prefix string `pre` to be displayed in front of all elements, a separator string `sep` to be displayed between successive elements, and a postfix string `post` to be displayed at the end. The result of the operation is the string

```
pre+xs(0).toString+sep+...+sep+xs(xs.length-1).toString+post
```

There is a second overloaded variant, which only takes a separator string:

```
xs mkString sep = xs mkString ("", sep, "")
```

Examples:

```
scala> abcde mkString ("[", ", ", "]")
res17: String = [a,b,c,d,e]
scala> abcde mkString ""
res18: String = abcde
scala> abcde mkString ("List(", ", ", ")")
res19: String = List(a, b, c, d, e)
```

There are also variants of the `mkString` methods called `addString` which append the constructed string to a `StringBuilder` object, rather than returning them as result:

```
scala> val buf = new StringBuilder
buf: StringBuilder =
scala> abcde addString (buf, "(", ";", ")")
res20: StringBuilder = (a;b;c;d;e)
```

The `mkString` and `addString` methods are inherited from `List`'s base class `Seq`, so they are applicable to all sorts of sequences.

Converting lists: `elements`, `toArray`, `copyToArray`

To convert data between the flat world of arrays and the recursive world of lists, you can use method `toArray` in class `List` and `toList` in class `Array`:

```
scala> val arr = abcde.toArray
arr: Array[Char] = Array(a, b, c, d, e)
scala> arr.toString
res21: String = Array(a, b, c, d, e)
scala> arr.toList
res22: List[Char] = List(a, b, c, d, e)
```

There's also a method `copyToArray` which copies list elements to successive array positions within some destination array. The operation

```
xs copyToArray (arr, start)
```

copies all elements of the list `xs` to the array `arr`, beginning with position `start`. You must ensure that the destination array `arr` is large enough to hold the list in full.

```
scala> val arr2 = new Array[Int](10)
arr2: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
scala> List(1, 2, 3) copyToArray (arr2, 3)
scala> arr2.toString
res24: String = Array(0, 0, 0, 1, 2, 3, 0, 0, 0, 0)
```

Finally, if you need to access list elements via an iterator, there is the `elements` method:

```
scala> val it = abcde.elements
it: Iterator[Char] = non-empty iterator
scala> it.next
res25: Char = a
scala> it.next
res26: Char = b
```

Example: Merge sort

The insertion sort presented earlier is simple to formulate, but it is not very efficient. Its average complexity is proportional to the square of the length

of the input list. A more efficient algorithm is *merge sort*, which works as follows.

First, if the list has zero or one elements, it is already sorted, so one returns the list unchanged. Longer lists are split into two sub-lists, each containing about half the elements of the original list. Each sub-list is sorted by a recursive call to the sort function, and the resulting two sorted lists are then combined in a merge operation.

For a general implementation of merge sort, you want to leave open the type of list elements to be sorted, and also want to leave open the function to be used for the comparison of elements. You obtain a function of maximal generality by passing these two items as parameters. This leads to the following implementation.

```
def msort[T](less: (T, T) => Boolean)(xs: List[T]): List[T] = {  
    def merge(xs: List[T], ys: List[T]): List[T] = (xs, ys) match {  
        case (Nil, _) => ys  
        case (_, Nil) => xs  
        case (x :: xs1, y :: ys1) =>  
            if (less(x, y)) x :: merge(xs1, ys) else y :: merge(xs, ys1)  
    }  
    val n = xs.length/2  
    if (n == 0) xs  
    else {  
        val (ys, zs) = xs splitAt n  
        merge(msort(less)(ys), msort(less)(zs))  
    }  
}
```

The complexity of `msort` is proportional ($n \log(n)$), where n is the length of the input list. To see why, note that splitting a list in two and merging two sorted lists each take time proportional to the length of the argument list(s). Each recursive call of `msort` halves the number of elements in its input, so there are about $\log(n)$ consecutive recursive calls until the base case of lists of length 1 is reached. However, for longer lists each call spawns off two further calls. Adding everything up we obtain that at each of the $\log(n)$ call levels, every element of the original lists takes part in one split operation and in one merge operation. Hence, every call level has a total cost proportional

to n . Since there are $\log(n)$ call levels, we obtain an overall cost proportional to $n \log(n)$. That cost does not depend on the initial distribution of elements in the list, so the worst case cost is the same as the average case cost. This property makes merge sort an attractive algorithm for sorting lists.

Here is an example how `msort` is used.

```
scala> msort((x: Int, y: Int) => x < y)(List(5, 7, 1, 3))
res27: List[Int] = List(1, 3, 5, 7)
```

Partial method applications

The `msort` function is a classical example of the currying discussed in [Chapter 9](#). The currying makes it easy to specialize the function for particular comparison functions. For instance,

```
scala> val intSort = msort((x: Int, y: Int) => x < y) _
intSort: (List[Int]) => List[Int] = <function>
scala> val reverseSort = msort((x: Int, y: Int) => x > y) _
reverseSort: (List[Int]) => List[Int] = <function>
```

As described in [Section 8.6](#), an underscore stands for a missing argument list. In this case, the only missing argument is the list that should be sorted.

14.7 Operations on lists Part II: Higher-order methods

Many operations over lists have a similar structure. One can identify several patterns that appear time and time again. Examples are: transforming every element of a list in some way, verifying whether a property holds for all elements of a list, extracting from a list elements satisfying a certain criterion, or combining the elements of a list using some operator. In Java, such patterns would usually be expressed by idiomatic combinations of for-loops or while-loops. In Scala, they can be expressed more concisely and directly using higher-order operators, which are implemented as methods in class `List`. These are discussed in the following.

Mapping over lists: map, flatMap and foreach

An operation `xs map f` takes as arguments a list `xs` of type `List[T]` and a function `f` of type `T => U`. It returns the list resulting from applying the function `f` to each list element in `xs`. For instance:

```
scala> List(1, 2, 3) map (_ + 1)
res28: List[Int] = List(2, 3, 4)

scala> val words = List("the", "quick", "brown", "fox")
words: List[java.lang.String] = List(the, quick, brown, fox)

scala> words map (_.length)
res29: List[Int] = List(3, 5, 5, 3)

scala> words map (_.toList.reverse.mkString(""))
res30: List[String] = List(eht, kciuq, nworb, xof)
```

The `flatMap` operator is similar to `map`, but it takes a function returning a list of elements as its right argument. It applies the function to each list and returns the concatenation of all function results. The difference between `map` and `flatMap` is illustrated in the following example:

```
scala> words map (_.toList)
res31: List[List[Char]] = List(List(t, h, e), List(q, u, i, c, k), List(b, r, o, w, n), List(f, o, x))

scala> words flatMap (_.toList)
res32: List[Char] = List(t, h, e, q, u, i, c, k, b, r, o, w, n, f, o, x)
```

You see that where `map` returns a list of lists, `flatMap` returns a single list in which all element lists are concatenated.

The interplay of `map` and `flatMap` is also demonstrated by the following expression, which constructs a list of all pairs (i, j) such that $1 \leq j < i < 5$:

```
scala> List.range(1, 5) flatMap (i => List.range(1, i) map (j => (i, j)))
res33: List[(Int, Int)] = List((2,1), (3,1), (3,2), (4,1), (4,2), (4,3))
```

The third map-like operation is `foreach`. Unlike `map` and `flatMap`, `foreach` takes a procedure (a method with result type `Unit`) as right argument. It simply applies the procedure to each list element. The result of the operation itself is again `Unit`; no list of results is assembled. As an example, here is a concise way of summing up all numbers in a list:

```
scala> var sum = 0
sum: Int = 0
scala> List(1, 2, 3, 4, 5) foreach (sum += _)
scala> sum
res35: Int = 15
```

Filtering lists: `filter`, `partition`, `find`, `takeWhile`, `dropWhile`, and `span`

The operation `xs filter p` takes as arguments a list `xs` of type `List[T]` and a predicate function `p` of type `T => Boolean`. It yields the list of all elements `x` in `xs` for which `p(x)` is `true`. For instance:

```
scala> List(1, 2, 3, 4, 5) filter (_ % 2 == 0)
res36: List[Int] = List(2, 4)
scala> words filter (_ .length == 3)
res37: List[java.lang.String] = List(the, fox)
```

The `partition` method is like `filter`, but returns a pair of lists. One list contains all elements for which the predicate is `true`, the other list contains all elements for which the predicate is `false`. It is defined by the equality:

```
xs partition p = (xs filter p(_), xs filter !p(_))
```

Example:

```
scala> List(1, 2, 3, 4, 5) partition (_ % 2 == 0)
res38: (List[Int], List[Int]) = (List(2, 4), List(1, 3, 5))
```

The `find` method is also similar to `filter` but it returns the first element satisfying a given predicate, rather than all such elements. The operation

`xs find p` takes a list `xs` and a predicate `p` as arguments. It returns an optional value. If there is an element `x` in `xs` for which `p(x)` is true, `Some(x)` is returned. Otherwise, if `p` is false for all elements, `None` is returned. Examples:

```
scala> List(1, 2, 3, 4, 5) find (_ % 2 == 0)
res39: Option[Int] = Some(2)

scala> List(1, 2, 3, 4, 5) find (_ <= 0)
res40: Option[Int] = None
```

The `takeWhile` and `dropWhile` operators also take a predicate as right argument. The operation `xs takeWhile p` takes the longest prefix of list `xs` such that every element in the prefix satisfies `p`. Analogously, the operation `xs dropWhile p` removes the longest prefix from list `xs` such that every element in the prefix satisfies `p`. Examples:

```
scala> List(1, 2, 3, -4, 5) takeWhile (_ > 0)
res41: List[Int] = List(1, 2, 3)

scala> words dropWhile (_ startsWith "t")
res42: List[java.lang.String] = List(quick, brown, fox)
```

The `span` method combines `takeWhile` and `dropWhile` in one operation, just like `splitAt` combines `take` and `drop`. It returns a pair of two lists, defined by the equality

$$\text{xs span } p = (\text{xs takeWhile } p, \text{xs dropWhile } p)$$

Like `splitAt`, `span` avoids to traverse the list `xs` twice.

```
scala> List(1, 2, 3, -4, 5) span (_ > 0)
res43: (List[Int], List[Int]) = (List(1, 2, 3), List(-4, 5))
```

Predicates over lists: `forall` and `exists`

The operation `xs forall p` takes as arguments a list `xs` and a predicate `p`. Its result is true if all elements in the list satisfy `p`. Conversely, the operation `xs exists p` returns true if there is an element in `xs` which satisfies the predicate `p`. For instance, to find out whether a matrix represented as a list of lists has a row with only zeroes as elements:

```
scala> def hasZeroRow(m: List[List[Int]]) =
    |   m exists (row => row forall (_ == 0))
hasZeroRow: (List[List[Int]])Boolean

scala> hasZeroRow(diag3)
res44: Boolean = false
```

Folding lists: ‘/:’ and ‘:\ ’

Another common kind of operations combine the elements of a list with some operator. For instance:

$$\begin{aligned} \text{sum(List(a, b, c))} &= 0 + a + b + c \\ \text{product(List(a, b, c))} &= 1 * a * b * c \end{aligned}$$

These are both special instances of a fold operation:

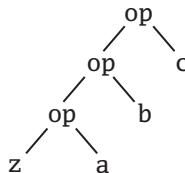
```
scala> def sum(xs: List[Int]): Int = (0 /: xs) (_ + _)
sum: (List[Int])Int

scala> def product(xs: List[Int]): Int = (1 /: xs) (_ * _)
product: (List[Int])Int
```

A *fold left* operation ($z /: xs$) (op) takes three arguments: A unit element z , a list xs , and a binary operation op . The result of the fold is op applied between successive elements of the list prefixed by z . For instance:

$$(z /: List(a, b, c)) (op) = op(op(op(z, a), b), c)$$

Or, graphically:



Here's another example that illustrates how ‘/:’ is used. To concatenate all words in a list of strings with spaces between them and in front, you can write:

```
scala> ("\" /: words) (_ + " " + _)
res45: java.lang.String = the quick brown fox
```

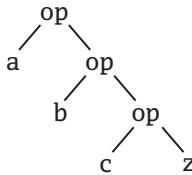
This gives an extra space at the beginning. To remove the space, you can use this slight variation:

```
scala> (words.head /: words.tail) (_ + " " + _)
res46: java.lang.String = the quick brown fox
```

The ‘`/:`’ operator produces left-leaning operation trees (its syntax with the slash rising forward is intended to be a reflection of that). The operator has ‘`:\
:`’ as an analog which produces right-leaning trees. For instance:

```
(List(a, b, c) :\  
 z) (op) = op(a, op(b, op(c, z)))
```

Or, graphically:



The ‘`:\
:`’ operator is pronounced *fold right*. It takes the same three arguments as fold left, but the first two appear in reversed order: The first argument is the list to fold, the second is the neutral element.

For associative operations `op`, fold left and fold right are equivalent, but there might be a difference in efficiency. Consider for instance an operation corresponding to the `List.flatten` method that is explained in this Chapter. The operation concatenates all elements in a list of lists. This could be implemented with either fold left ‘`/:`’ or fold right ‘`:\
:`’:

```
def flatten1[T](xss: List[List[T]]) =
  (List[T]() /: xss) (_ ::: _)
```

```
def flatten2[T](xss: List[List[T]]) =
  (xss :\  
 List[T]()) (_ ::: _)
```

Because list concatenation `xs ::: ys` takes time proportional to its first argument `xs`, the implementation in terms of fold right in `flatten2` is more efficient than the fold left implementation in `flatten1`. The problem is that `flatten1(xss)` copies the first element list `xss.head` $n - 1$ times, where n is the length of the list `xss`.

Note that both versions of flatten require a type annotation on the empty list which is the unary element of the fold. This is due to a limitation in Scala's type inferencer, which fails to infer the correct type of the list automatically. If you try to leave out the annotation, you get the following:

```
scala> def flatten2[T](xss: List[List[T]]) =  
|     (xss :\ List()) (_ ::: _)  
<console>:15: error: type mismatch;  
found   : List[T]  
required: List[Nothing]  
          (xss :\ List()) (_ ::: _)
```

To find out why the type inferencer goes wrong, you'll need to know about the types of the fold methods and how they are implemented. More on this in [Chapter 20](#).

Example: List reversal using fold

Earlier in the chapter you saw an implementation of method `reverse` whose run-time was quadratic in the length of the list to be reversed. Here is now a different implementation of `reverse` which has linear cost. The idea is to use a fold left operation based on the following program scheme.

```
def reverse2[T](xs: List[T]) = (z? /: xs)(op?)
```

It only remains to fill in the `z?` and `op?` parts. In fact, you can try to deduce these parts from some simple examples. To deduce the correct value of `z?`, you can start with the smallest possible list, `List()` and calculate as follows:

```
List()  
= // by the properties of reverse2  
reverse2(List())  
= // by the template for reverse2  
(z? /: List())(op?)  
= // by definition of /:  
z?
```

Hence, $z?$ must be `List()`. To deduce the second operand, let's pick the next smallest list as an example case. You know already that $z? = List()$, so you can calculate as follows:

```
List(x)
=    // by the properties of reverse2
reverse2(List(x))
=    // by the template for reverse2, with z? = List()
(List() /: List(x)) (op?)
=    // by definition of /:
op?(List(), x)
```

Hence, $op(List(), x)$ equals `List(x)`, which is the same as $x :: List()$. This suggests to take as op the ‘`::`’ operator with its operands exchanged (this operation is sometimes pronounced “snoc”, in reference to ‘`::`’, which is pronounced “cons”). We arrive then at the following implementation for `reverse2`.

```
def reverse2[T](xs: List[T]) =
(List[T]() /: xs) { (xs, x) => x :: xs }
```

(Again, the type annotation in `List[T]()` is necessary to make the type inferencer work.) If you analyze the complexity of `reverse2`, you find that it applies a constant-time operation (“snoc”) n times, where n is the length of the argument list. Hence, the complexity of `reverse2` is linear, as hoped for.

Sorting lists: `sort`

The operation `xs sort before` sorts the elements of list `xs` using the `before` function for element comparison. The expression `x before y` should return `true` if `x` should come before `y` in the intended ordering for the sort. For instance:

```
scala> List(1, -3, 4, 2, 6) sort (_ < _)
res47: List[Int] = List(-3, 1, 2, 4, 6)

scala> List(1, -3, 4, 2, 6) sort (_ > _)
res48: List[Int] = List(6, 4, 2, 1, -3)
```

Note that `sort` does the same thing as the `msort` algorithm in the last section, but it is a method of class `List` whereas `msort` was defined outside lists.

14.8 Operations on lists Part III: Methods of the `List` object

So far, all operations you have seen in this chapter are implemented as methods of class `List`, so you invoke them on individual list objects. There are also a number of methods in the globally accessible object `scala.List`, which is the companion object of class `List`. Some of these operations are factory methods that create lists. Others are operations that work on lists of some specific type of shape. Both kinds of methods will be presented in the following.

Creating lists from their elements: `List.apply`

You have already seen on several occasions list literals such as `List(1, 2, 3)`. There's nothing special about their syntax. A literal like `List(1, 2, 3)` is simply the application of the function object `List` to the elements 1, 2, 3. That is, it is equivalent to `List.apply(1, 2, 3)`.

```
scala> List.apply(1, 2, 3)
res49: List[Int] = List(1, 2, 3)
```

Creating a range of numbers: `List.range`

The `range` method creates a list consisting of a range of numbers. Its simplest form is `List.range(from, to)`; this creates a list of all numbers starting at `from` and going up to `to` minus one. So the end value `to` does not form part of the range.

There's also a version of `range` that takes a `step` value as third parameter. This operation will yield list elements that start at `from` and that are `step` values apart. The `step` can be positive as well as negative.

```
scala> List.range(1, 5)
res50: List[Int] = List(1, 2, 3, 4)

scala> List.range(1, 9, 2)
```

```
res51: List[Int] = List(1, 3, 5, 7)
scala> List.range(9, 1, -3)
res52: List[Int] = List(9, 6, 3)
```

Creating uniform lists: List.make

The `make` method creates a list consisting of zero or more copies of the same element. It takes two parameters: the length of the list to be created, and its element:

```
scala> List.make(5, 'a')
res53: List[Char] = List(a, a, a, a, a)
scala> List.make(3, "hello")
res54: List[java.lang.String] = List(hello, hello, hello)
```

Unzipping lists: List.unzip

The `unzip` operation is the inverse of `zip`. Where `zip` takes two lists and forms a list of pairs, `unzip` takes a list of pairs and returns two lists, one consisting of the first element of each pair, the other consisting of the second element.

```
scala> val zipped = "abcde".toList zip List(1, 2, 3)
zipped: List[(Char, Int)] = List((a,1), (b,2), (c,3))
scala> List.unzip(zipped)
res55: (List[Char], List[Int]) = (List(a, b, c), List(1, 2, 3))
```

You might wonder why `unzip` is a method of the global `List` object, instead of being a method of class `List`? The problem is that `unzip` does not work on any list but only on a list of pairs. Because `List` is a generic type, Scala's type system forces all methods of class `List` to be also generic in the `List` element type. `unzip` isn't generic in `List`'s element type, thus it has to go elsewhere. It might be possible to extend Scala's type system in the future so that it accepts non-generic methods in generic classes, but so far this has not been done.

Concatenating lists: List.flatten, List.concat

The flatten method takes a list of lists and concatenates all element lists of the main list.

```
scala> val xss = List(List('a', 'b'), List('c'), List('d', 'e'))
xss: List[List[Char]] = List(List(a, b), List(c), List(d,
e))

scala> List.flatten(xss)
res56: List[Char] = List(a, b, c, d, e)
```

flatten is packaged in the global List object for the same reason as unzip: It does not operate on any list, but only on lists with lists as elements, so it can't be a method of the generic List class.

The concat method is similar to flatten in that it concatenates a number of element lists. The element lists are given directly as repeated parameters. The number of lists to be passed to concat is arbitrary:

```
scala> List.concat(List('a', 'b'), List('c'))
res57: List[Char] = List(a, b, c)

scala> List.concat(List(), List('b'), List('c'))
res58: List[Char] = List(b, c)

scala> List.concat()
res59: List[Nothing] = List()
```

Mapping and testing pairs of lists: List.map2, List.forall2, List.exists2

The map2 method is similar to map, but it takes two lists as arguments together with a function that maps two element values to a result. The function gets applied to corresponding elements of the two lists, and a list is formed from the results:

```
List.map2(List(10, 20), List(3, 4, 5)) (_ * _)
```

The exists2 and forall2 methods are similar to exists and forall, respectively, but they also take two lists and a boolean test function that takes two arguments. The test function is applied to corresponding arguments.

```
scala> List.forall2(List("abc", "de"), List(3, 2)) (_.length == _)
res60: Boolean = true

scala> List.exists2(List("abc", "de"), List(3, 2)) (_.length != _)
res61: Boolean = false
```

14.9 Understanding Scala's type inference algorithm

One difference between the previous uses of `sort` and `msort` concerns the admissible syntactic forms of the comparison function. Compare

```
scala> msort((x: Char, y: Char) => x > y)(abcde)
res62: List[Char] = List(e, d, c, b, a)
```

with

```
scala> abcde sort (_ > _)
res63: List[Char] = List(e, d, c, b, a)
```

The two expressions are equivalent, but the first uses a longer form of comparison function with named parameters and explicit types whereas the second uses the concise form `(_ > _)` where named parameters are replaced by underscores. Of course, you could also use the first, longer form of comparison with `sort`. However, the short form cannot be used with `msort`:

```
scala> msort(_ > _)(abcde)
<console>:12: error: missing parameter type for expanded
  function ((x$1, x$2) => x$1.$greater(x$2))
           msort(_ > _)(abcde)
               ^
```

To understand why, you need to know some details of Scala's type inference algorithm. Type inference in Scala is flow based. In a method application `m(args)`, the inferencer first checks whether the method `m` has a known type. If it has, that type is used to infer the expected type of the arguments. For instance, in `abcde.sort(_ > _)`, the type of `abcde` is `List[Char]`, hence `sort` is known to be a method that takes arguments of type `(Char, Char) => Boolean` to results of type `List[Char]`. Since the correct parameter types of the closure argument are thus known, they need

not be written explicitly. With what it knows about `sort`, the inferencer can deduce that `(_ > _)` should expand to `((x: Char, y: Char) => x > y)` where `x` and `y` are some arbitrary fresh names.

Now consider the second case, `msort(_ > _)(abcde)`. The type of `msort` is a curried, polymorphic method type that takes an argument of type $(T, T) \Rightarrow \text{Boolean}$ to a function from `List[T]` to `List[T]` where `T` is some as-yet unknown type. The `msort` method needs to be instantiated with a type parameter before it can be applied to its arguments. Because the precise instance type of `msort` in the application is not yet known, it cannot be used to infer the type of its first argument. The type inferencer changes its strategy in this case; it first type-checks method arguments to determine the proper type-instance of the method. However, when tasked to type-check the shorthand closure `(_ > _)` it fails because it has no information about the types of the implicit closure parameters that are indicated by underscores.

One way to resolve the problem is to pass an explicit type parameter to `msort`, as in:

```
scala> msort[Char](_ > _)(abcde)
res64: List[Char] = List(e, d, c, b, a)
```

Because the correct instance type of `msort` is now known, it can be used to infer the type of the arguments.

Another possible solution is to rewrite the `msort` method so that its parameters are swapped:

```
scala> def msort1[T](xs: List[T])(less: (T, T) => Boolean): List[T] =
... // same implementation as msort, but with arguments swapped
```

Now type inference succeeds:

```
scala> msort1(abcde)(_ > _)
res65: List[Char] = List(e, d, c, b, a)
```

What has happened is that the inferencer used the known type of the first parameter `abcde` to determine the type parameter of `msort`. Once the precise type of `msort` was known, it could be used in turn to infer the type of the second parameter `(_ > _)`.

Generally, when tasked to infer the type parameters of a polymorphic method, the type inferencer consults the types of all value arguments in the

first argument section but no arguments beyond that. Since `msort1` is a curried method with two parameter sections, the second argument (*i.e.* the closure) did not need to be consulted to determine the type parameter of the method.

This inference scheme suggests the following library design principle: When designing a polymorphic method that takes some non-functional arguments and a closure argument, place the closure argument last in a curried parameter section by its own. That way, the method's correct instance type can be inferred from the non-functional arguments, and that type can in turn be used to type-check the closure. The net-effect is that users of the method need to give less type information and can write closures in more compact ways.

Now to the more complicated case of a *fold* operation. Why is there the need for an explicit type parameter in an expression like the body of the `flatten1` method above?

```
(xss :\ List[T]()) (_ ::: _)
```

The type of the right-fold operation is polymorphic in two type variables. Given an expression

```
(xs :\ z) (op)
```

The type of `xs` must be a list of some arbitrary type A, say `xs: List[A]`. The unit `z` can be of some other type B. The operation `op` must then take two arguments of type A and B and must return a result of type B, *i.e.* `op: (A, B) => B`. Because the type of `z` is not related to the type of the list `xs`, type inference has no context information for `z`. Now consider the erroneous expression in the method `flatten2` above:

```
(xss :\ List()) (_ ::: _)
```

The unit value `z` in this fold is an empty list `List()` so without additional type information its type is inferred to be a `List[Nothing]`. Hence, the inferencer will infer that the B type of the fold is `List[Nothing]`. Therefore, the operation `(_ ::: _)` of the fold is expected to be of the following type

```
(List[T], List[Nothing]) => List[Nothing]
```

This is indeed a possible type for the operation in that fold but it is not a very useful one! It says that the operation always takes an empty list as second argument and always produces an empty list as result. In other words, the type inference settled too early on a type for `List()`, it should have waited until it had seen the type of the operation `op`. So the (otherwise very useful) rule to only consider the first argument section in a curried method application for determining the method's type is at the root of the problem here. On the other hand, even if that rule were relaxed, the inferencer still could not come up with a type for `op` because its parameter types are not given. Hence, there is a Catch 22 situation which can only be resolved by an explicit type annotation from the programmer.

This example highlights some limitations of the local, flow-based type inference scheme of Scala. It is not present in the more global "Hindley/Milner"-style type-inference schemes for classical functional languages such as ML or Haskell. However, Scala's local type inference deals much more gracefully with object-oriented subtyping than Hindley/Milner-style type inference does. Fortunately, the limitations show up only in some corner cases, and are usually easily fixed by adding an explicit type annotation.

Adding type annotations is also a useful debugging technique when you get confused by type error messages related to polymorphic methods. If you are unsure what caused a particular type error, just add some type arguments or other type annotations, which you think are correct. Then you should be able to quickly see where the real problem is.

Chapter 15

Collections

Collections let you organize large numbers of objects. Scala has a rich collection library. In the simple cases, you can throw a few objects into a set or a list and not think much about it. For trickier cases, Scala provides a general library with several collection types, such as sequences, sets and maps. Each collection type comes in two variants—mutable and immutable. Most kinds of collections have several different implementations that have different tradeoffs of speed, space, and the requirements on their input data. You've seen many collection types in previous chapters. In this chapter we'll show you the big picture of how they relate to each other.

15.1 Overview of the library

Figure 10.2 shows the class hierarchy of the most frequently used kinds of collections in Scala's standard library. Each of these types is a trait, so each of them allows multiple implementations. All of them have a good default implementation available in the standard library.

At the top of the hierarchy is `Iterable`, the trait for possibly infinite groups of objects. The key property of an `Iterable` is that it is possible to iterate through the elements of the collection using a method named `elements`. Using this one abstract method, `Iterable` can implement dozens of other methods. Nonetheless, the trait too abstract for most programming situations. Because it is not guaranteed to be finite, you must be careful what methods you invoke on it. Imagine, for example, asking the infinite sequence of numbers from 10 through infinity—a perfectly valid `Iterable`—whether

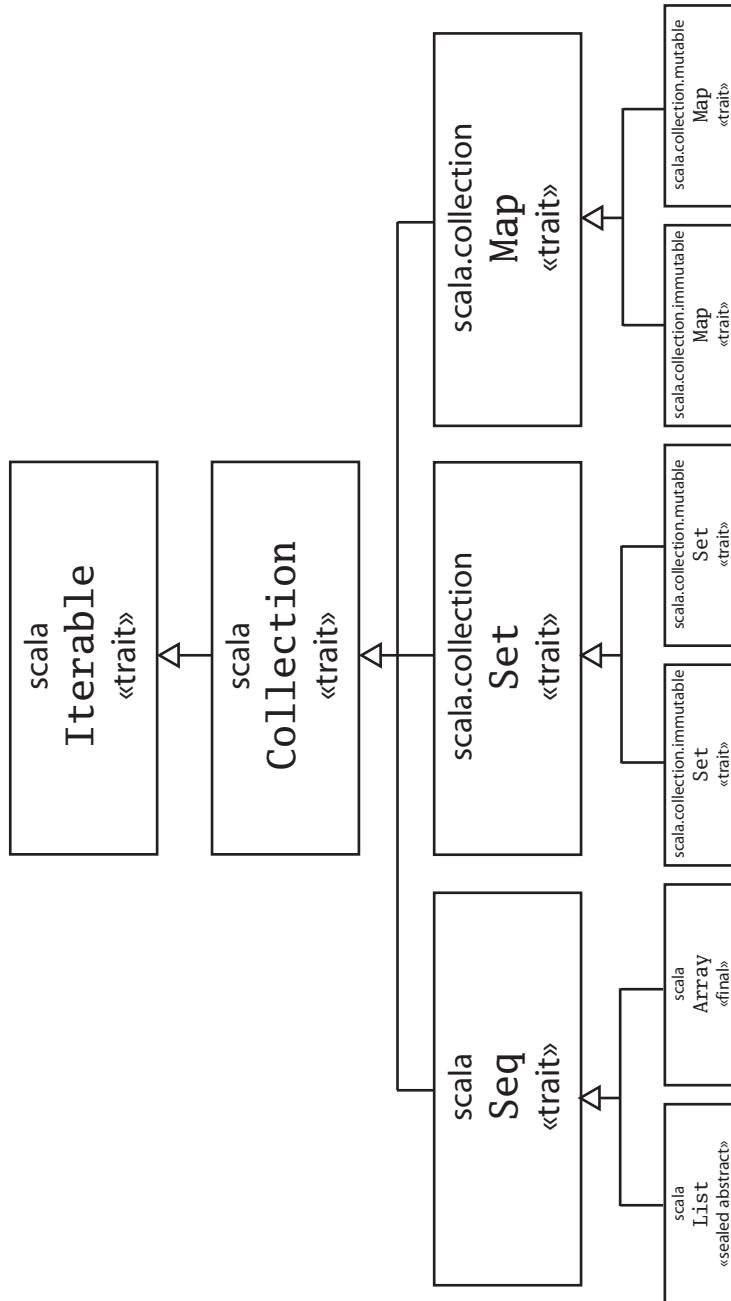


Figure 15.1: Scala collections class hierarchy.

it includes any negative numbers. The library would search forever.

Just below `Iterable` is `Collection`, the trait of finite collections of objects. In addition to supporting iteration through its elements, a `Collection` has a fixed, finite size. Because instances of `Collection` are known to be finite, they can be processed with less worry about infinite loops.

`Collection` is still quite abstract. Below `Collection` are three traits that are used more frequently: `Seq`, `Set`, and `Map`. `Seq` is the trait for all collections whose elements are numbered from zero up to some limit. The elements of a sequence can be queried by their associated number, so for example you can ask a sequence for its third or fifteenth element. `Set` is the trait for collections that can be efficiently queried for membership. You cannot ask a set for its fourth element, because the elements are not ordered, but you can efficiently ask it whether it includes the number 4. `Map` is the trait for lookup tables. You can use a map to associate values with some set of keys. For example, later in this chapter a map is used to associate each word in a string with the number of times that word occurs in the string. Thus the words are *mapped* to the number of times they occur.

Below these three traits, at the bottom of Figure TODO, are two variants of each of the three. Each of the three has a mutable variant that allows modifying collections in place, and each of them also has an immutable variant that instead has efficient methods to create new, modified collections out of old ones, without the old ones changing at all. Both the mutable and immutable variants have many uses, so the library provides both.

15.2 Sequences

Sequences, classes that inherit from the `Seq` trait, let you work with groups of data lined up in order. Because the elements are ordered, you can ask for the first element, the second element, the 103rd element, and so on. There are a few different kinds of sequences you should learn about early on: arrays, array buffers, and lists.

Arrays

Arrays are one of the most basic collections. They allow you to hold a sequence of objects in a row and efficiently access elements at an arbitrary

position in the sequence. You access the individual objects in an array via a 0-based index.

Arrays are normally created with the usual new keyword:

```
scala> val numbers = new Array[Int](5)
numbers: Array[Int] = Array(0, 0, 0, 0, 0)
```

You must specify two arguments when you create an array: the type of elements that are in the array and the length of the array. In this example, the numbers array holds 5 objects of type Int. As in the rest of Scala, note that type arguments like Int are placed in square brackets instead of parentheses.

To read an element from an array, put the index in parentheses. It looks, and acts, just like calling a method with the index as an argument.

```
scala> numbers(3)
res0: Int = 0
```

In this case the result is zero, because every element of this array holds its initial element. The initial element of an array in Scala is, as in Java, the zero value of the array's type: 0 for numeric types, false for booleans, and null for reference types.

Writing into an array uses a similar syntax:

```
scala> numbers(3) = 42
scala> numbers(3)
res2: Int = 42
```

Behind the scenes, such an assignment is treated the same as a call to a method named update:

```
scala> numbers.update(3, 420)
scala> numbers(3)
res4: Int = 420
```

Any class with an update method can use this syntax.

Looping over the elements of an array is easy:

```
scala> for (x <- numbers)
|   println(x)
```

```
0  
0  
0  
420  
0
```

If you need to use the index as you loop through, then you can loop like this:

```
scala> for (i <- 0 until numbers.length)  
|   println(i + " " + numbers(i))  
0 0  
1 0  
2 0  
3 420  
4 0
```

Scala arrays are represented in the same way as Java arrays. So, you can seamlessly use existing Java methods that return arrays. For example, Java can split a string into a sequence of tokens using the method `split`. The method knows nothing about Scala, but the array it returns can still be used in Scala.

```
scala> val words = "The quick brown fox".split(" ")  
words: Array[java.lang.String] = Array(The, quick, brown,  
fox)  
  
scala> for (word <- words)  
|   println(word)  
The  
quick  
brown  
fox
```

Array buffers

There are two other sequence types that you should know about in addition to arrays. First, class `ArrayBuffer` is like an array except that you can additionally add and remove elements from the beginning and end of the sequence. All Array operations are available, though they are a little slower

due to a layer of wrapping in the implementation. The new addition and removal operations are constant time on average, but occasionally require linear time due to the implementation needing to allocate a new array to hold the buffer's contents.

To use an ArrayBuffer, you must first import it from the mutable collections package:

```
scala> import scala.collection.mutable.ArrayBuffer  
import scala.collection.mutable.ArrayBuffer
```

Create an array buffer just like an array, except that you do not need to specify a size. The implementation will adjust the allocated space automatically.

```
scala> val buf = new ArrayBuffer[Int]()  
buf: scala.collection.mutable.ArrayBuffer[Int] =  
ArrayBuffer()
```

Then, you can append to the array using the `+=` method:

```
scala> buf += 12  
scala> buf += 15  
scala> buf  
res10: scala.collection.mutable.ArrayBuffer[Int] =  
ArrayBuffer(12, 15)
```

All the normal array methods are available. For example, you can ask it its size, or you can retrieve an element by its index:

```
scala> buf.length  
res11: Int = 2  
scala> buf(0)  
res12: Int = 12
```

Lists

The most important sequence type is class `List`, the immutable linked-list described in detail in the previous chapter. Lists support fast addition and removal of items to the beginning of the list, but they do not provide fast

access to arbitrary indexes because the implementation must iterate through the list linearly.

This combination of features might sound odd, but they hit a sweet spot that works well for many algorithms. The fast addition and removal of initial elements means that pattern matching works well, as described in [Chapter 12](#). The immutability of lists helps you develop a correct, efficient algorithm because you never need to make copies of a list.

As a default choice, many programmers new to Scala might choose arrays and array buffers because arrays are a familiar data structure. People with more experience programming in Scala, however, often have the opposite default and start with a list. Either way, the best choice depends on the specific circumstances, so it is good to learn how to use both.

Conversion

You can convert any collection. to an array or a list. Such conversion requires copying all of the elements of the collection, and thus is slow for large collections. Sometimes you need to do it, though, due to an existing API. Further, many collections only have a few elements anyway, in which case there is only a small speed penalty.

Use `toArray` to convert to an array and `toList` to convert to a list.

```
scala> buf.toArray
res13: Array[Int] = Array(12, 15)

scala> buf.toList
res14: List[Int] = List(12, 15)
```

15.3 Tuples

A tuple combines a fixed number of items together so that they can be passed around as a whole. Unlike an array, a tuple can hold objects with different types. Here is an example of a tuple holding an integer, a string, and the output console.

```
(1, "hello", Console)
```

Tuples save you the tedium of defining simplistic data-heavy classes. Even though defining a class is already easy, it does require a certain minimum effort, which sometimes serves no purpose. Tuples save you the effort of choosing a name for the class, choosing a scope to define the class in, and choosing names for the members of the class. If your class simply holds an integer and a string, there is no clarity added by defining a class named `AnIntegerAndAString`.

Tuples have a major difference from the other collections described in this chapter: they can combine objects of different types. Because of this difference, tuples do not inherit from `Collection`. If you find yourself wanting to group exactly one integer and exactly one string, then you want a tuple, not a `List` or `Array`.

A common application of tuples is to return multiple values from a method. For example, here is a method that finds the longest word in a collection and also returns its index.

```
def longestWord(words: Array[String]) = {  
    var word = words(0)  
    var idx = 0  
    for (i <- 1 until words.length)  
        if (words(i).length > word.length) {  
            word = words(i)  
            idx = i  
        }  
    (word, idx)  
}
```

Here is an example use of the method:

```
scala> val longest =  
|   longestWord("The quick brown fox".split(" "))  
longest: (String, Int) = (quick,1)
```

The `longestWord` function here computes two items: `word`, the longest word in the array, and `idx`, the index of that word. To keep things simple, the function assumes there is at least one word in the list, and it breaks ties by choosing the word that comes earlier in the list. Once the function has chosen which word and index to return, it returns both of them together using the tuple syntax `(word, idx)`.

To access elements of a tuple, you can use method `_1` to access the first element, `_2` to access the second, and so on.

```
scala> longest._1
res15: String = quick
scala> longest._2
res16: Int = 1
```

Additionally, you can assign each element of the tuple to its own variable:¹

```
scala> val (word, idx) = longest
word: String = quick
idx: Int = 1
scala> word
res17: String = quick
```

By the way, if you leave off the parentheses you get a different result:

```
scala> val word, idx = longest
word: (String, Int) = (quick,1)
idx: (String, Int) = (quick,1)
```

This syntax gives *multiple definitions* of the same expression. Each variable is initialized with its own evaluation of the expression on the right-hand side. That the expression evaluates to a tuple in this case does not matter. Both variables are initialized to the tuple in its entirety. See [Chapter 16](#) for some examples where multiple definitions are convenient.

As a note of warning, tuples are almost too easy to use. Tuples are great when you combine data that has no meaning beyond “an A and a B.” However, whenever the combination has some meaning, or you want to add some methods to the combination, it is better to go ahead and create a class. For example, do not use a 3-tuple for the combination of a month, a day, and a year. Make a Date class. It makes your intentions explicit, which both clears up the code for human readers and gives the compiler and language opportunities to help you.

¹This syntax is actually a special case of *pattern matching*, as described in detail in [Chapter 12](#).

15.4 Sets and maps

Two other kinds of collections you will use all the time when programming Scala are sets and maps. Sets and maps have a fast lookup algorithm, so they can quickly decide whether or not an object is in the collection.

It is easiest to explain using an example. Start by importing the package `scala.collection.mutable`, so you have easy access to the relevant classes.

```
scala> import scala.collection.mutable  
import scala.collection.mutable
```

Now you can create a new set using the `empty` method:

```
scala> val words = mutable.Set.empty[String]  
words: scala.collection.mutable.Set[String] = Set()
```

Note that you have to supply the type of objects this set will hold, which in this example is `String`. You can then add elements to the set using the `+=` method.

```
scala> words += "hello"  
scala> words += "there"  
scala> words += "there"  
scala> words  
res21: scala.collection.mutable.Set[String] = Set(there,  
hello)
```

Note that if an element is already included in the set, then it is not added a second time. That is why "there" only appears one time in the `words` set even though it was added twice.

As a longer example, you can use a set to count the number of different words in a string. The `split` method can separate the string into words, if you specify spaces and punctuation as word separators. The regular expression `[!,.]+` suffices: it indicates one or more space and/or punctuation characters.

```
scala> val text = "See Spot run. Run, Spot, Run!"
```

```
text: java.lang.String = See Spot run. Run, Spot, Run!
scala> for (w <- text.split("[ !,.]+"))
|   println(w)
See
Spot
run
Run
Spot
Run
```

As written, however, the same word can end up in the resulting collection. If you want to avoid double counting, a set can help. Simply convert the words to the same case and then add them to a set. By the nature of how sets work, each distinct word will appear exactly one time in the set.

```
scala> import scala.collection.mutable
import scala.collection.mutable
scala> val words = mutable.Set.empty[String]
words: scala.collection.mutable.Set[String] = Set()
scala> for (w <- text.split("[ !,.]+"))
|   words += w.toLowerCase()
scala> words
res24: scala.collection.mutable.Set[String] = Set(spot, run,
see)
```

The text includes exactly three (lowercased) words: spot, run, and see.

Some common set operations are shown in [Table 15.1](#). There are many more operations available; browse the API documentation for details.

Maps have the same fast lookup algorithm as sets, but additionally let you associate a value with each element of the collection. Using a map looks just like using an array, except that instead of indexing with integers counting from 0, you can use any kind of key. Several common map operations are shown in [Table 15.2](#).

Creating a mutable map looks like this:

```
scala> val map = mutable.Map.empty[String, Int]
map: scala.collection.mutable.Map[String,Int] = Map()
```

Table 15.1: Common operations for sets.

What it is	What it does
<code>import scala.collection.mutable</code>	make the mutable collections easy to access
<code>val words = mutable.Set.empty[String]</code>	create an empty set
<code>words += "the"</code>	add one object
<code>words -= "the"</code>	remove an object, if it exists
<code>words ++= List("do", "re", "mi")</code>	add multiple objects
<code>words --= List("do", "re")</code>	remove multiple objects
<code>words.size</code>	ask for the size of the set (returns 1)
<code>words.contains("mi")</code>	check for inclusion (returns true)

Note that when you create a map, you must specify two types. The first type is for the *keys* of the map, the second for the *values*. In this case, the keys are strings and the values are integers.

Setting entries in a map looks just like setting entries in an array:

```
scala> map("hello") = 1
scala> map("there") = 2
scala> map
res27: scala.collection.mutable.Map[String,Int] = Map(hello
-> 1, there -> 2)
```

Likewise, reading a map is like reading an array:

```
scala> map("hello")
res28: Int = 1
```

Putting it all together, here is a method that counts the number of times each word occurs in a text.

Table 15.2: Common operations for maps.

What it is	What it does
<code>import scala.collection.mutable</code>	make the mutable collections easy to access
<code>val words = mutable.Map.empty[Int, String]</code>	create an empty map
<code>words += (1 -> "one")</code>	add a map entry from 1 to "one"
<code>words -= 1</code>	remove a map entry, if it exists
<code>words ++= List(1 -> "one", 2 -> "two", 3 -> "three")</code>	add multiple map entries
<code>words --- List(1, 2)</code>	remove multiple objects
<code>words.size</code>	ask for the size of the set (returns 1)
<code>words.contains(3)</code>	check for inclusion (returns true)
<code>words(3)</code>	retrieve the value at a specified key (returns "three")
<code>words.keys</code>	list all keys (returns an Iterator over just the number 3.)

```
scala> def wordcounts(text: String) = {
|   val counts = mutable.Map.empty[String, Int]
|   for (rawWord <- text.split("[ ,!]+")) {
|     val word = rawWord.toLowerCase
|     val oldCount =
|       if (counts.contains(word))
|         counts(word)
|       else
|         0
|     counts += (word -> (oldCount + 1))
|   }
}
```

```
|   counts  
| }  
wordcounts: (String)scala.collection.mutable.Map[String,Int]
```

```
scala> wordcounts("See Spot run! Run, Spot, run!!")  
res29: scala.collection.mutable.Map[String,Int] = Map(see ->  
1, run -> 3, spot -> 2)
```

Given these counts, we see that this text talks a lot about running, but not so much about seeing.

The way this code works is that a map, `counts`, maps each word to the number of times it occurs in the text. For each word in the text, the word's old count is looked up, that count is incremented by one, and the new count is saved back into `counts`. Note the use of `contains` to check whether a word has been seen yet or not. If `counts.contains(word)` is not true, then the word has not yet been seen and zero is used for the count.

A few common map operations are shown in [Table 15.2](#).

15.5 Initializing collections

You have already seen the syntax `List(1,2,3)` for creating a list with its contents specified immediately. This notation works for sets and maps as well. You leave off a new statement and then put the initial contents in parentheses. Here are a few examples:

```
scala> List(1,2,3)  
res30: List[Int] = List(1, 2, 3)  
  
scala> mutable.Set(1,2,3)  
res31: scala.collection.mutable.Set[Int] = Set(3, 1, 2)  
  
scala> mutable.Map(1->"hi", 2->"there")  
res32: scala.collection.mutable.Map[Int,java.lang.String] =  
Map(2 -> there, 1 -> hi)  
  
scala> Array(1,2,3)  
res33: Array[Int] = Array(1, 2, 3)
```

Note the `->` notation that is available for initializing maps. Each entry to be added to the map is given by a *key->value* pair.

Behind the scenes, the parentheses here are expanded into a call to apply, much like the syntactic help for update methods on arrays and maps. For example, the following two expressions have the same result:

```
scala> List(1,2,3)
res34: List[Int] = List(1, 2, 3)

scala> List.apply(1,2,3)
res35: List[Int] = List(1, 2, 3)
```

So far, these examples have not specified a type in the code you enter. You write `List(1,2,3)`, and Scala creates a `List[Int]` for you automatically. What happens is that the compiler chooses a type for you based on the elements it sees. Often this is a perfectly fine type to choose, and you can leave it as is.

Sometimes, though, you want to create a collection literal and specify a different type from the one the compiler would choose. This is especially an issue with mutable collections:

```
scala> val stuff = mutable.Set(42)
stuff: scala.collection.mutable.Set[Int] = Set(42)

scala> stuff += "abracadabra"
<console>:7: error: type mismatch;
 found   : java.lang.String("abracadabra")
 required: Int
           stuff += "abracadabra"
               ^
```

The problem here is that `stuff` was given an element type of `Int`. If you want it to have an element type of `Any`, you have to specify so by putting the element type in square brackets, like this:

```
scala> val stuff = mutable.Set[Any](42)
stuff: scala.collection.mutable.Set[Any] = Set(42)
```

15.6 Immutable collections

Scala provides immutable versions of all of its collection types. These versions cannot be changed after they are initialized. You should use them

Table 15.3: Immutable analogs to mutable collections

mutable	immutable	code to convert
Array	List	x.toList
mutable.Map	Map	Map.empty ++ x
mutable.Set	Set	Set.empty ++ x

whenever you know a collection should not be changed, so that you do not accidentally change it later. These are hard bugs to find when they show up, because all you know is that some code somewhere has modified the collection.

The main immutable types are given in in [Table 15.3](#). Tuples are left out of the table because they do not have a mutable analog. Note that many of the immutable types are shorter to write. You write `Set` for an immutable set, but `mutable.Set` for a mutable one, and thus Scala quietly biases programmers toward using immutable collections.

The third column of [Table 15.3](#) shows one way to convert a mutable collection `x` to an immutable one. For example, if `x` is an `Array`, you can convert it to a `List` with `x.toList`. Conversion in the opposite direction is also possible: simply make an empty mutable collection of the desired type and then use `++=` to add to it all of the elements of the immutable collection.

Immutable collections have analogs to all of the methods available for mutable collections. There is one big difference, however. Instead of having operations to modify the collection in place, they have methods to create a new version of the collection that has had a change made. When such a method is used, the old version of the collection is still accessible and has exactly the same contents. As a visual reminder of this difference, the mutable methods use an `=` sign in the name while the immutable variants do not. For example, you add to an immutable set with `+` instead of `+=`. After the addition, the original set remains available, unchanged.

```
scala> val original = Set(1,2,3)
original: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> val updated = original + 5
```

Table 15.4: Common operations for immutable sets.

What it is	What it does
<code>val nums = Set(1, 2, 3, 3)</code>	create a set (returns <code>Set(1, 2, 3)</code>)
<code>nums + 5</code>	add one element (returns <code>Set(1, 2, 3, 5)</code>)
<code>nums - 1</code>	remove one element (returns <code>Set(2, 3)</code>)
<code>nums ++ List(5, 6)</code>	add multiple elements (returns <code>Set(1, 2, 3, 5, 6)</code>)
<code>nums -- List(1, 2)</code>	remove multiple elements (returns <code>Set(3)</code>)
<code>nums.size</code>	ask for the size of the set (returns <code>3</code>)
<code>nums.contains(3)</code>	check for inclusion (returns <code>true</code>)

```
updated: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 5)
scala> original
res37: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```

Some common operations for immutable sets and maps are shown in [Table 15.4](#) and [Table 15.5](#). As you can see, most operations are the same, and the ones that are not mostly differ in that they do not modify the receiver collection.

Switching to and from immutable collections

Because there are subtle implications of choosing mutable versus immutable collections, programmer often write code that starts with one kind of collection and then switches to the other later on. To help with this common kind of switch, Scala includes a little bit of syntactic sugar. Even though immutable sets and maps do not support a true `+=` method, Scala gives a useful alternate interpretation to `+=`. Whenever you write `a += b`, and `a` does not support a `+=` method, Scala will try interpreting it as `a = a + b`. For example, immutable sets do not support `+=`.

Table 15.5: Common operations for immutable maps.

What it is	What it does
<code>val words = Map(1 -> "one", 2 -> "two")</code>	create a map (returns <code>Map(1 -> "one", 2 -> "two")</code>)
<code>words + (6 -> "six")</code>	add an entry to a map (returns <code>Map(1 -> "one", 2 -> "two", 6 -> "six")</code>)
<code>words - 1</code>	remove one map entry (returns <code>Map(2 -> "two")</code>)
<code>words ++ List(1 -> "one", 2 -> "two", 6 -> "six")</code>	add multiple map entries (returns <code>Map(1 -> "one", 2 -> "two", 6 -> "six")</code>)
<code>words -- List(1, 2)</code>	remove multiple objects (returns <code>Map()</code>)
<code>words.size</code>	ask for the size of the set (returns 2)
<code>words.contains(2)</code>	check for inclusion (returns <code>true</code>)
<code>words(2)</code>	retrieve the value at a specified key (returns "two")
<code>words.keys</code>	list all keys (returns an <code>Iterator</code> over the numbers 1 and 2)

```
scala> authorized += "Bill"
scala> authorized
res40: scala.collection.immutable.Set[java.lang.String] =
Set(Nancy, Jane, Bill)
```

The same idea applies to any method ending in `=`, not just `+=`:

```
scala> authorized -= "Jane"
scala> authorized +++ List("Tom", "Harry")
scala> authorized
res43: scala.collection.immutable.Set[java.lang.String] =
```

```
Set(Nancy, Bill, Tom, Harry)
```

To see how this is useful, consider again the following example from [Chapter 1](#).

```
var capital = Map("US" -> "Washington",
                  "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

This code uses immutable collections. If you want to try using mutable collections instead, then all that is necessary is to import the mutable version of `Map`, thus overriding the default import of the immutable `Map`.

```
import scala.collection.mutable.Map // only change needed!
var capital = Map("US" -> "Washington",
                  "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

Not all examples are quite that easy to convert, but the special treatment of methods ending in an equals sign will often reduce the amount of code that needs changing.

By the way, this syntactic treatment works on any kind of value, not just collections. For example, here it is being used on floating-point numbers:

```
scala> var roughlyPi = 3.0
roughlyPi: Double = 3.0
scala> roughlyPi += 0.1
scala> roughlyPi += 0.04
scala> roughlyPi
res46: Double = 3.14
```

Lean towards immutable

There are problems where mutable collections work better, and other problems where immutable collections work better. Whenever you are in doubt, it is often better to start with an immutable collection and change it later if you need to. Mutation is simply a powerful feature, and powerful features can be hard to reason about. In much the same way, reassignable variables, vars, are powerful, but whenever you do not need that power it is better to disable it and use vals.

It is even worth going the opposite way. Whenever you find a collections-using program becoming complicated and hard to reason about, you should consider whether it would help to change some of the collections to be immutable. In particular, if you find yourself worrying about making copies of mutable collections in just the right places, or if you find yourself thinking a lot about who “owns” or “contains” a mutable collection, consider switching some of the collections to be immutable.

15.7 Conclusion

This chapter has shown you the most important kinds of Scala collections. While you can get by if you simply use arrays for everything, it is worth learning, over time, the map, set, and tuple classes. Using the right class at the right time can make your code shorter and cleaner. As you do so, keep an eye out for chances to use immutable equivalents of the collection classes you choose. They can make your code just a little more clean and easy to work with.

Chapter 16

Stateful Objects

Previous chapters have put the spotlight on functional objects. They have done that because the idea of objects without any mutable state deserves to be better known. However, it is also perfectly possible to define objects with mutable state in Scala. Such stateful objects often come up naturally when one wants to model objects in the real world that change over time.

This chapter explains what stateful objects are, and what Scala provides in term of syntax to express them. The second part of this chapter also introduces a larger case study on discrete event simulation, which is one of the application areas where stateful objects arise naturally.

16.1 What makes an object stateful?

The principal difference between a purely functional object and a stateful object can be observed even without looking at the object's implementation. When you invoke a method or dereference a field on some purely functional object, you will always get the same result. For instance, given a list of characters

```
val cs = List('a', 'b', 'c')
```

an application of `cs.head` will always return '`a`'. This is the case even if there is an arbitrary number of operations on the list `cs` between the point where it is defined and the point where the access `cs.head` is made.

For a stateful object, on the other hand, the result of a method call or field access may depend on what operations were performed on the object before.

A good example of a stateful object is a bank account. Here's a simplified implementation of bank accounts:

```
class BankAccount {  
    private var balance: Int = 0  
  
    def getBalance: Int = balance  
  
    def deposit(amount: Int) {  
        assume(amount > 0)  
        balance += amount  
    }  
  
    def withdraw(amount: Int): Boolean =  
        if (amount <= balance) { balance -= amount; true }  
        else false  
}
```

The `BankAccount` class defines a private variable `balance` and three public methods. The `getBalance` method returns the current balance, the `deposit` method adds a given amount to `balance`, and the `withdraw` method tries to subtract a given amount from `balance` while assuring that the remaining balance won't be negative. The return value of `@withdraw@` is a Boolean indicating whether the requested funds were successfully withdrawn.

Even if you know nothing about the inner workings of the `BankAccount` class, you can still tell that `BankAccounts` are stateful objects. All you need to do is create a `BankAccount` and invoke some of its methods:

```
scala> val account = new BankAccount  
account: BankAccount = BankAccount@eb06c3  
  
scala> account deposit 100  
  
scala> account withdraw 80  
res1: Boolean = true  
  
scala> account withdraw 80  
res2: Boolean = false
```

Note that the two final withdrawals in the above interaction returned different results. The first withdraw operation returned `true` because the bank account contained sufficient funds to allow the withdrawal. The second operation,

although the same as the first one, returned `false`, because the balance of the account had been reduced so that it no longer covers the requested funds. So, clearly bank accounts have mutable state, because the same operation can return different results at different times.

You might think that the statefulness of `BankAccount` is immediately apparent because it contains a `var` definition. State and `vars` usually go hand in hand, but things are not always so clear-cut. For instance, a class might be stateful without defining or inheriting any `vars` because it forwards method calls to other objects which have mutable state. The reverse is also possible: A class might contain `vars` and still be purely functional. An example is the `HashConsing` trait of [Chapter 11](#). This trait defines two `vars` but they are used only for optimizations, to speed up the operation of the public `hashCode` method. Seen from the outside, the trait is still purely functional, because its methods give the same result every time they are invoked.

16.2 Reassignable variables and properties

There are two fundamental operations on a reassignable variable: you can get its value or you can set it to a new value. In libraries such as JavaBeans, these operations are often encapsulated in separate getter and setter methods which need to be defined explicitly. In Scala, every variable which is a non-private member of some object implicitly defines a getter and a setter method with it. These getters and setters are named differently from their Java conventions, however. The getter of a variable `x` is just named `x`, while its setter is named `x_=`.

For example, if it appears in a class, the variable definition

```
var hour: Int = 12
```

generates a getter `hour` and setter `hour_=` in addition to a reassignable field. The field is always marked `private[this]`, which means it can be accessed only from the object that contains it. The getter and setter, on the other hand, get the same visibility as the original `var`. If the `var` definition is `public`, so are its getter and setter, if it is `protected` they are also `protected`, and so on.

For instance, consider the following class `Time` which defines two variables named `hour` and `minute`.

```
class Time {
    var hour = 12
    var minute = 0
}
```

This implementation is exactly equivalent to the following class definition:

```
class Time {
    private[this] var h = 12
    private[this] var m = 0
    def hour: Int = h
    def hour_=(x: Int) { h = x }
    def minute = m
    def minute_=(x: Int) { m = x }
}
```

In the above definitions, the names of the local fields `h` and `m` are arbitrarily chosen so as not to clash with any names already in use.

An interesting aspect about this expansion of `vars` into getters and setters is that you can also chose to define a getter and a setter directly instead of defining a `var`. By defining these access methods directly you can interpret the operations of variable access and variable assignment as you like. For instance, the following variant of class `Time` contains assumptions¹ that catch all assignments to `hour` and `minute` with illegal values:

```
class Time {
    private[this] var h = 12
    private[this] var m = 12
    def hour: Int = h
    def hour_=(x: Int) { assume(0 <= x && x < 24); h = x }
    def minute = m
    def minute_=(x: Int) { assume(0 <= x && x < 60); m = x }
}
```

Some languages have a special syntactic construct for these variable-like quantities that are not plain variables in that their getter or setter can be redefined. For instance, C# has properties, which fulfill this role. Scala's convention of always interpreting a variable as a pair of setter and getter methods

¹ Recall that `assume` was explained in [Section 10.4](#).

gives you in effect the same capabilities as C# properties without requiring special syntax. Properties can serve many different purposes. In the example above, the setters enforced an invariant, thus protecting the variable from being assigned illegal values. You could also use a property to log all accesses to getters or setters of a variable. Or you could integrate variables with events, for instance by notifying some subscriber methods each time a variable is modified.

It is also possible, and sometimes useful, to define a getter and a setter without an associated field. An example is the following class Thermometer which encapsulates a temperature variable that can be read and updated. Temperatures can be expressed in Celsius or Fahrenheit degrees. The class below allows you to get and set the temperature in either measure.

```
class Thermometer {  
    var celsius: Float =_  
    def fahrenheit =  
        celsius * 9 / 5 + 32  
    def fahrenheit_=(f: Float) =  
        celsius = (f - 32) * 5 / 9  
    override def toString =  
        fahrenheit+F+"/"+celsius+C"  
}
```

The first line in the body of this class defines a variable `celsius` that is supposed to contain the temperature in degrees Celsius. The variable's value is initially undefined. This is expressed by using '`_`' as the "initializing value" of `celsius`. More precisely, an initializer '`= _`' of a field assigns a default value to that field. Depending on the field type, this default value is either 0, 0.0, false, or null. It is the same as if the same variable was defined in Java without an initializer. Note that you cannot simply leave off the '`_`' initializer in Scala. If you had written

```
var celsius: Float
```

this would declare an abstract variable, not an uninitialized one.²

The `celsius` variable definition is followed by a getter `fahrenheit` and a setter `fahrenheit_=` that access the same temperature, but in degrees

²Abstract variables are explained in [Chapter 18](#)

Fahrenheit. There is no separate field that contains the current temperature value in Fahrenheit. Instead the getter and setter methods for Fahrenheit values automatically convert from and to degrees Celsius, respectively. Here is an example session that interacts with a `Thermometer` object:

```
scala> val t = new Thermometer
t: Thermometer = 32.0F/0.0C
scala> t.celsius = 100
scala> t
res4: Thermometer = 212.0F/100.0C
scala> t.fahrenheit = -40
scala> t
res6: Thermometer = -40.0F/-40.0C
```

16.3 Case study: discrete event simulation

The rest of this chapter shows by way of an extended example how stateful objects can be combined with first-class function values in interesting ways. You'll see the design and implementation of a simulator for digital circuits. This task is decomposed into several subproblems, each of which is interesting individually: First, you'll be presented a simple but general framework for discrete event simulation. The main task of this framework is to keep track of actions that are performed in simulated time. Second, you'll learn how discrete simulation programs are structured and built. The idea of such simulations is to model physical objects by simulated objects, and to use the simulation framework to model physical time. Finally, you'll see a little domain specific language for digital circuits. The definition of this language highlights a general method for embedding domain-specific languages in a host language like Scala.

The basic example is taken from the classic textbook “Structure and Interpretation of Computer Programs” by Abelson and Sussman [Abe96]. What's different here is that the implementation language is Scala instead of Scheme, and that the various aspects of the example are structured into four software layers: one for the simulation framework, another for the basic circuit simulation package, a third layer for a library of user-defined circuits

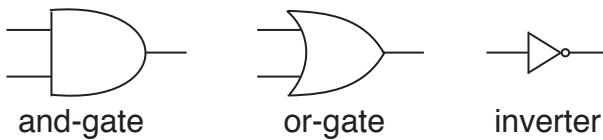


Figure 16.1: Basic gates.

and the last layer for each simulated circuit itself. Each layer is expressed as a class, and more specific layers inherit from more general ones. Understanding these layers in detail will take some time; if you feel you want to get on with learning more Scala instead, it's safe to skip ahead to the next chapter.

16.4 A language for digital circuits

Let's start with a little language to describe digital circuits. A digital circuit is built from *wires* and *function boxes*. Wires carry *signals* which are transformed by function boxes. Signals will be represented by booleans: `true` for signal-on and `false` for signal-off.

Figure 16.1 shows three basic function boxes (or: *gates*):

- An *inverter*, which negates its signal
- An *and-gate*, which sets its output to the conjunction of its input.
- An *or-gate*, which sets its output to the disjunction of its input.

These gates are sufficient to build all other function boxes. Gates have *delays*, so an output of a gate will change only some time after its inputs change.

We describe the elements of a digital circuit by the following set of Scala classes and functions.

First, there is a class `Wire` for wires. We can construct wires as follows.

```
val a = new Wire
val b = new Wire
val c = new Wire
```

or, equivalent but shorter:

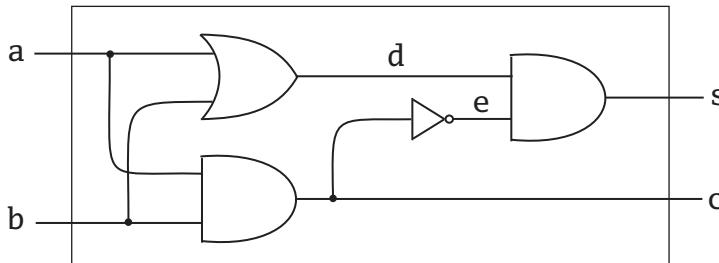


Figure 16.2: A half adder circuit.

```
val a, b, c = new Wire
```

Second, there are three procedures which “make” the basic gates we need.

```
def inverter(input: Wire, output: Wire)
def andGate(a1: Wire, a2: Wire, output: Wire)
def orGate(o1: Wire, o2: Wire, output: Wire)
```

What’s unusual, given the functional emphasis of Scala, is that these procedures construct the gates as a side-effect, instead of returning the constructed gates as a result. For instance, an invocation of `inverter(a, b)` places an inverter between the wires `a` and `b`. Its result is `Unit`. It turns out that this side-effecting construction makes it easier to construct complicated circuits gradually.

More complicated function boxes are built from the basic gates. For instance, the following method constructs a half-adder, which takes two inputs `a` and `b` and produces a sum `s` defined by $s = (a + b) \% 2$ and a carry `c` defined by $c = (a + b) / 2$.

```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) {
    val d, e = new Wire
    orGate(a, b, d)
    andGate(a, b, c)
    inverter(c, e)
    andGate(d, e, s)
}
```

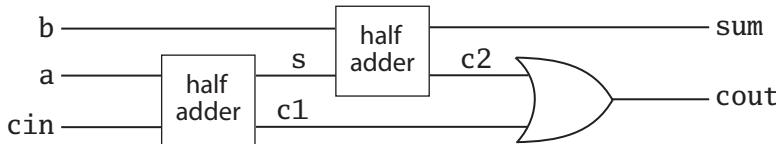


Figure 16.3: A full adder circuit.

A picture of this half-adder is shown in [Figure 16.2](#).

Note that `halfAdder` is a parameterized function box just like the three methods which construct the primitive gates. You can use the `halfAdder` method to construct more complicated circuits. For instance, the following code defines a full one bit adder, shown graphically in [Figure 16.3](#), which takes two inputs `a` and `b` as well as a carry-in `cin` and which produces a sum output defined by $\text{sum} = (\text{a} + \text{b} + \text{cin}) \% 2$ and a carry-out output defined by $\text{cout} = (\text{a} + \text{b} + \text{cin}) / 2$.

```

def fullAdder(a: Wire, b: Wire, cin: Wire,
              sum: Wire, cout: Wire)
{
    val s, c1, c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
}

```

Class `Wire` and functions `inverter`, `andGate`, and `orGate` represent a little language in which users can define digital circuits. It's a good example of a domain specific language (or DSL for short) that's defined as a library in some other language instead of being implemented on its own. Such languages are called *embedded DSLs*.

The implementation of the circuit DSL still needs to be worked out. Since the purpose of defining a circuit in that DSL is simulating the circuit, it makes sense to base the DSL implementation on a general API for discrete event simulation. The next two sections will present first the simulation API and then the implementation of the circuit DSL on top of it.

```
abstract class Simulation {  
    type Action = () => Unit  
  
    case class WorkItem(time: Int, action: Action)  
  
    private var curtime = 0  
    def currentTime: Int = curtime  
  
    private var agenda: List[WorkItem] = List()  
  
    private def insert(ag: List[WorkItem],  
                      item: WorkItem): List[WorkItem] =  
        if (ag.isEmpty || item.time < ag.head.time) item :: ag  
        else ag.head :: insert(ag.tail, item)  
  
    def afterDelay(delay: Int)(block: => Unit) {  
        val item = WorkItem(currentTime + delay, () => block)  
        agenda = insert(agenda, item)  
    }  
  
    private def next() {  
        (agenda: @unchecked) match {  
            case item :: rest =>  
                agenda = rest;  
                curtime = item.time  
                item.action()  
            }  
        }  
  
    def run() {  
        afterDelay(0) {  
            println("*** simulation started, time = "+  
                  currentTime+" ***")  
        }  
        while (!agenda.isEmpty) next()  
    }  
}
```

Figure 16.4: The `Simulation` class.

16.5 The Simulation API

The simulation API is shown in [Figure 16.4](#). It consists of class `Simulation` in package `simulator`. Concrete simulation libraries inherit this class and augment it with domain-specific functionality. The elements of the `Simulation` class are presented in the following.

Discrete event simulation performs user-defined *actions* at specified *times*. The actions, which are defined by concrete simulation subclasses, all share a common type:

```
type Action = () => Unit
```

The definition above defines `Action` to be an alias of the type of procedures that take an empty parameter list and that return `Unit`.

The time at which an action is performed is simulated time; it has nothing to do with the actual “wall-clock” time. Simulated times are represented simply as integers. The current simulated time is kept in a private variable

```
private var curtime: Int = 0
```

The variable has a public accessor method which retrieves the current time:

```
def currentTime: Int = curtime
```

This combination of private variable with public accessor is used to make sure that the current time cannot be modified outside the `Simulation` class.

An action which is to be executed at a specified time is called a *work item*. Work items are implemented by the following class:

```
case class WorkItem(time: Int, action: Action)
```

The `WorkItem` class is made a case class because of the syntactic conveniences this entails: you can use the factory method `WorkItem` to create instances of the class and you get accessors for the constructor parameters `time` and `action` for free. Note also that class `WorkItem` is nested inside class `Simulation`. Nested classes are treated similarly as in Java. [Chapter 18](#) gives more details.

The `Simulation` class keeps an *agenda* of all remaining work items that are not yet executed. The work items are sorted by the simulated time at which they have to be run:

```
private var agenda: List[WorkItem] = List()
```

The only way to add a work item to the agenda is with the following method:

```
def afterDelay(delay: Int)(block: => Unit) {
    val item = WorkItem(currentTime + delay, () => block)
    agenda = insert(agenda, item)
}
```

As the name implies, this method inserts an action (given by `block`) into the agenda so that it is scheduled for execution `delay` time units after the current simulation time. For instance, the invocation

```
afterDelay(delay) { count += 1 }
```

creates a new work item which will be executed at simulated time `currentTime + delay`. The code to be executed is contained in the method's second argument. The formal parameter for this argument has type “`=> Unit`,” *i.e.* it is a computation of type `Unit` which is passed by-name. Recall that call-by-name parameters are not evaluated when passed to a method. So in the call above `count` would be incremented only once the simulation framework calls the action stored in the work item.

The created work item is then inserted into the agenda. This is done by the `insert` method, which maintains the invariant that the agenda is time-sorted:

```
private def insert(ag: List[WorkItem],
                  item: WorkItem): List[WorkItem] =
    if (ag.isEmpty || item.time < ag.head.time) item :: ag
    else ag.head :: insert(ag.tail, item)
```

The core of the `Simulation` class is defined by the `run` method.

```
def run() {
    afterDelay(0) {
        println("*** simulation started, time = "+
               currentTime+" ***")
    }
    while (!agenda.isEmpty) next()
}
```

This method repeatedly takes the front item in the agenda and executes it until there are no more items left in the agenda to execute. Each step is performed by calling the next method, which is defined as follows.

```
private def next() {  
    agenda match {  
        case item :: rest =>  
            agenda = rest  
            curtime = item.time  
            item.action()  
    }  
}
```

The next method decomposes the current agenda with a pattern match into a front item item and a remaining list of work items rest. It removes the front item from the current agenda, sets the simulated time curtime to the work item's time, and executes the work item's action.

That's it. This seems surprisingly little code for a simulation framework. You might wonder how this framework could possibly support interesting simulations, if all it does is execute a list of work items? In fact the power of the simulation framework comes from the fact that actions stored in work items can themselves install further work items into the agenda when they are executed. That makes it possible to have long-running simulations evolve from simple beginnings.

Missing cases and the @unchecked annotation

Note that next can be called only if the agenda is non-empty. There's no case for an empty list, so you would get a MatchError exception if you tried to run next on an empty agenda.

In fact, the Scala compiler will warn you that you missed one of the possible patterns for a list:

```
Simulator.scala:19: warning: match is not exhaustive!  
missing combination           Nil
```

```
agenda match {  
    ^
```

```
one warning found
```

In this case, the missing case is not a problem, because you know that `next` is called only on a non-empty agenda. Therefore, you might want to disable the warning. You have seen in [Chapter 12](#) that this can be done by adding an `@unchecked` annotation to the selector expression of the pattern match:

```
private def next() {  
    (agenda: @unchecked) match {  
        case item :: rest =>  
            agenda = rest  
            curtime = item.time  
            item.action()  
    }  
}
```

16.6 Circuit Simulation

The next step is to use the simulation framework to implement the domain-specific language for circuits. Recall that the circuit DSL consists of a class for wires and methods that create and-gates, or-gates, and inverters. These are all contained in a class `BasicCircuitSimulation` which extends the simulation framework. Here's an outline of this class:

```
abstract class BasicCircuitSimulation extends Simulation {  
    def InverterDelay: Int  
    def AndGateDelay: Int  
    def OrGateDelay: Int  
    class Wire { ... }  
    def inverter(input: Wire, output: Wire) {...}  
    def andGate(a1: Wire, a2: Wire, output: Wire) {...}  
    def orGate(o1: Wire, o2: Wire, output: Wire) {...}  
    def probe(name: String, wire: Wire) {...}  
}
```

The class declares three abstract methods `InverterDelay`, `AndGateDelay` and `OrGateDelay` which represent the delays of the basic gates. The actual delays are not known at the level of this class because they would depend on

the technology of circuits that are simulated. That's why the delays are left abstract in class `BasicCircuitSimulation`, so that their concrete definition is delegated to a subclass.

The implementation of the other members of class `BasicCircuitSimulation` is described next.

The Wire class

A wire needs to support three basic actions.

`getSignal: Boolean` returns the current signal on the wire.

`setSignal(sig: Boolean)` sets the wire's signal to `sig`.

`addAction(p: Action)` attaches the specified procedure `p` to the *actions* of the wire. The idea is that all action procedures attached to some wire will be executed every time the signal of the wire changes. Typically actions are added to a wire by components connected to the wire. An attached action is executed once at the time it is added to a wire, and after that, every time the signal of the wire changes.

Here is an implementation of the `Wire` class:

```
class Wire {  
    private var sigVal = false  
    private var actions: List[Action] = List()  
    def getSignal = sigVal  
    def setSignal(s: Boolean) =  
        if (s != sigVal) {  
            sigVal = s  
            actions foreach (_ ())  
        }  
    def addAction(a: Action) = {  
        actions = a :: actions; a()  
    }  
}
```

Two private variables make up the state of a wire. The variable `sigVal` represents the current signal, and the variable `actions` represents the action

procedures currently attached to the wire. The only interesting method implementation is the one for `setSignal`: When the signal of a wire changes, the new value is stored in the variable `sigVal`. Furthermore, all actions attached to a wire are executed. Note the shorthand syntax for doing this: `actions foreach (_ ())` applies the function `_ ()` to each element in the `actions` list. As described in [Section 8.5](#), the function `_ ()` is a shorthand for `(f => f ())`, *i.e.* it takes a function (let's name it `f`) and applies it to the empty parameter list.

The Inverter Class

The only effect of creating an inverter is that an action is installed on its input wire. This action is invoked every time the signal on the input changes. The effect of the action is that the value of the inverter's output value is set (via `setSignal`) to the inverse of its input value. Since inverter gates have delays, this change should take effect only `InverterDelay` units of simulated time after the input value has changed and the action was executed. This suggests the following implementation.

```
def inverter(input: Wire, output: Wire) = {  
    def invertAction() {  
        val inputSig = input.getSignal  
        afterDelay(InverterDelay) {  
            output setSignal !inputSig  
        }  
    }  
    input addAction invertAction  
}
```

In this implementation, the effect of the `inverter` method is to add `invertAction` to the `input` wire. This action, when invoked, gets the input signal and installs another action that inverts the `output` signal into the simulation agenda. This other action is to be executed after `InverterDelay` units of simulated time. Note how the implementation uses the `afterDelay` method of the simulation framework to create a new work item that's going to be executed in the future.

The And-Gate Class

The implementation of and gates is analogous to the implementation of inverters. The purpose of an andGate is to output the conjunction of its input signals. This should happen at AndGateDelay simulated time units after any one of its two inputs changes. Hence, the following implementation:

```
def andGate(a1: Wire, a2: Wire, output: Wire) = {  
    def andAction() = {  
        val a1Sig = a1.getSignal  
        val a2Sig = a2.getSignal  
        afterDelay(AndGateDelay) {  
            output setSignal (a1Sig & a2Sig)  
        }  
    }  
    a1 addAction andAction  
    a2 addAction andAction  
}
```

The effect of the andGate method is to add andAction to both of its input wires a1 and a2. This action, when invoked, gets both input signals and installs another action that sets the output signal to the conjunction of both input signals. This other action is to be executed after AndGateDelay units of simulated time. Note that the output has to be recomputed if either of the input wires changes. That's why the same andAction is installed on each of the two input wires a1 and a2.

Exercise: Write the implementation of orGate.

Exercise: Another way is to define an or-gate by a combination of inverters and gates. Define a function orGate in terms of andGate and inverter. What is the delay time of this function?

Simulation output

To run the simulator, you still need a way to inspect changes of signals on wires. To accomplish this, it is useful to add a probe method that simulates the action of putting a probe on a wire.

```
def probe(name: String, wire: Wire) {
    def probeAction() {
        println(name+" "+currentTime+ " new-value = "+wire.getSignal)
    }
    wire addAction probeAction
}
```

The effect of the `probe` procedure is to install a `probeAction` on a given wire. As usual, the installed action is executed every time the wire's signal changes. In this case it simply prints the name of the wire (which is passed as first parameter to `probe`), as well as the current simulated time and the wire's new value.

Running the simulator

After all these preparations it's time to see the simulator in action. To define a concrete simulation, you need to inherit from a simulation framework class. To see something interesting, let's assume there is a class `CircuitSimulation` which extends `BasicCircuitSimulation` and contains definitions of half adders and full adders as they were presented earlier in the chapter:

```
abstract class CircuitSimulation
    extends BasicCircuitSimulation {
    def halfAdder(a: Wire, b: Wire,
                  s: Wire, c: Wire) { ... }
    def fullAdder(a: Wire, b: Wire, cin: Wire,
                  sum: Wire, cout: Wire) { ... }
}
```

A concrete circuit simulation will be an object that inherits from class `CircuitSimulation`. The object still needs to fix the gate delays according to the circuit implementation technology that's simulated. Finally, one also needs to define the concrete circuit that's going to be simulated. You can do these steps interactively using the Scala interpreter:

```
scala> import simulator._  
import simulator._
```

First, the gate delays. Define an object (call it `MySimulation`) that provides some numbers:

```
scala> object MySimulation extends CircuitSimulation {  
|   def InverterDelay = 1  
|   def AndGateDelay = 3  
|   def OrGateDelay = 5  
| }  
defined module MySimulation  
  
scala> import MySimulation._  
import MySimulation._
```

Next, the circuit. Define four wires, and place probes on two of them:

```
scala> val input1, input2, sum, carry = new Wire  
input1: MySimulation.Wire = simulator.BasicCircuitSimulation$Wire@111089b  
input2: MySimulation.Wire = simulator.BasicCircuitSimulation$Wire@14c352e  
sum: MySimulation.Wire = simulator.BasicCircuitSimulation$Wire@37a04c  
carry: MySimulation.Wire = simulator.BasicCircuitSimulation$Wire@1fd10fa  
  
scala> probe("sum", sum)  
sum 0 new-value = false  
  
scala> probe("carry", carry)  
carry 0 new-value = false
```

Note that the probes immediately print an output. This is a consequence of the fact that every action installed on a wire is executed a first time when the action is installed.

Now define a half-adder connecting the wires:

```
scala> halfAdder(input1, input2, sum, carry)
```

Finally, set one after another the signals on the two input wires to true and run the simulation:

```
scala> input1 setSignal true  
scala> run()  
*** simulation started, time = 0 ***  
sum 8 new-value = true
```

```
scala> input2 setSignal true
scala> run()
*** simulation started, time = 8 ***
carry 11 new-value = true
sum 15 new-value = false
```

16.7 Conclusion

This chapter has brought together two techniques that seem at first disparate: mutable state and higher-order functions. Mutable state was used to simulate physical entities whose state changes over time. Higher-order functions were used in the simulation framework to execute actions at specified points in simulated time. They were also used in the circuit simulations as *triggers* that associate actions with state changes. On the side, you have seen a simple way to define a domain-specific language as a library. That's probably enough for one chapter! If you feel like staying a bit longer, maybe you want to try more simulation examples. You can combine half-adders and full-adders to create larger circuits, or design new circuits from the basic gates defined so far and simulate them.

Chapter 17

Type Parameterization

This chapter explains some of the techniques for information hiding introduced in [Chapter 13](#) by means of concrete example: the design of a class for purely functional queues. It also explains type parameter variance as a new concept. There are some links between the two concepts in that information hiding may be used to obtain more general type parameter variance annotations – that's why they are presented together.

The chapter contains three parts. The first part develops a data structure for purely functional queues, which is interesting in its own right. The second part develops techniques to hide internal representation details of this structure. The final part explains variance of type parameters and how it interacts with information hiding.

17.1 Functional queues

A functional queue is a data structure with three operations.

`head` returns the first element of the queue

`tail` returns a queue without its first element

`append` returns a new queue with a given element appended at the end.

Unlike a standard queue, a functional queue does not change its contents when an element is appended. Instead, a new queue is returned which contains the element. You can try this out as follows.

```
scala> val q = Queue(1, 2, 3)
q: Queue[Int] = Queue(1, 2, 3)

scala> val q1 = q append 4
q1: Queue[Int] = Queue(1, 2, 3, 4)

scala> q
unnamed3: Queue[Int] = Queue(1, 2, 3)
```

If `Queue` was a standard queue implementation, the `append` operation in the second input line above would affect the contents of value `q`; in fact both the result `q1` and the original queue `q` would contain the sequence 1, 2, 3, 4 after the operation. But for a functional queue, the appended value shows up only in the result `q1`, not in the queue `q` being operated on.

Purely functional queues also have some similarity with lists. Both are so called *fully persistent* data structures, where old versions remain available even after extensions or modifications. Both support `head` and `tail` operations. But where a list is usually extended at the front, using a `:::` operation, a queue is extended at the end, using `append`.

How can this be implemented efficiently? Ideally, a functional queue should not have a fundamentally higher overhead than a standard, imperative one. That is, all three operations `head`, `tail`, and `append` should operate in constant time.

One simple approach to implement a functional queue would be to use a list as representation type. Then `head` and `tail` would just translate into the same operations on the list, whereas `append` would be concatenation. So this would give the following implementation:

```
class Queue1[T](elems: List[T]) {
    def head = elems.head
    def tail = elems.tail
    def append(x: T) = new Queue1(elems :: List(x))
}
```

The problem with this implementation is in the `append` operation – it takes time proportional to the number of elements stored in the queue. If you want constant time `append`, you could also try to reverse the order of the elements in the representation list, so that the last element that's appended comes first in the list. This would lead to the following implementation.

```
class Queue[T](leading: List[T], trailing: List[T]) {
    private def mirror =
        if (leading.isEmpty) new Queue(trailing.reverse, Nil)
        else this
    def head =
        mirror.leading.head
    def tail = {
        val q = mirror;
        new Queue(q.leading.tail, q.trailing)
    }
    def append[T](x: T) =
        new Queue(leading, x :: trailing)
}
```

Figure 17.1: Simple functional queues

```
class Queue2[T](smele: List[T]) {
    def head = smelete.last
    def tail = smelete.init
    def append(x: T) = new Queue2(x :: smelete)
}
```

Now `append` is constant time, but `head` and `init` are not. They now take time proportional to the number of elements stored in the queue.

Looking at these two examples, it does not seem easy to come up with an implementation that's constant time for all three operations. In fact, it looks doubtful that this is even possible! However, by combining the two operations one can get very close. The idea is to represent a queue by two lists, called `leading` and `trailing`. The `leading` list contains elements towards the front, whereas the `trailing` list contains elements towards the back of the queue in reversed order. The contents of the whole queue are at each instant equal to `leading :: trailing.reverse`.

Now, to append an element, one just conses it to the `trailing` list, so `append` is constant time. This means that, when an initially empty queue is constructed from successive `append` operations, the `trailing` list will grow

whereas the leading will stay empty. Then, before the first head or tail operation is performed on an empty leading list, the whole trailing list is copied to leading, reversing the order of the elements. This is done in an operation called `mirror`. Figure 17.1 shows an implementation of queues following this idea.

What is the complexity of this implementation of queues? The `mirror` operation might take time proportional to the number of queue elements, but only if list `leading` is empty. It returns directly if `leading` is non-empty. Because `head` and `tail` call `mirror`, their complexity might be linear in the size of the queue, too. However, the longer the queue gets, the less often `mirror` is called. Indeed, assume a queue of length n with an empty `leading` list. Then `mirror` has to reverse-copy a list of length n . However, the next time `mirror` will have to do any work is once the `leading` list is empty again, which will be the case after n `tail` operations. This means one can “charge” each of these n `tail` operation with one n ’th of the complexity of `mirror`, which means a constant amount of work. Assuming that `head`, `tail`, and `append` operations appear with about the same frequency, the *amortized* complexity is hence constant for each operation. So functional queues are asymptotically just as efficient as mutable ones.

Now, there’s some caveats in small print that need to be attached to this argument. First, the discussion only was about asymptotic behavior, the constant factors might well be somewhat different. Second, the argument rested on the fact that `head`, `tail` and `append` are called with about the same frequency. If `head` is called much more often than the other two operations, the argument is not valid, as each call to `head` might involve a costly re-organization of the list with `mirror`. The second caveat can be avoided; it is possible to design functional queues so that in a sequence of successive `head` operations only the first one might require a re-organization. You will find out at the end of this chapter how this is done.

17.2 Information hiding

Private constructors

The implementation of `Queue` is now quite good in what concerns efficiency. But one might object that this is paid for by exposing a needlessly detailed implementation. The `Queue` constructor, which is globally accessible, takes

two lists as parameters, where one is reversed – hardly an intuitive representation of a queue. What's needed is a way to hide this constructor from client code. In Java, this can be achieved by adding a `private` modifier to the constructor definition. In Scala the primary constructor does not have an explicit definition, it is defined implicitly by the class parameters and its body. Nevertheless, it is still possible to hide the primary constructor by adding a `private` modifier in front of the class parameter list, like this:

```
class Queue[T] private (leading: List[T], trailing: List[T]) {  
    ...  
}
```

The `private` modifier between the class name and its parameters indicates that the constructor of `Queue` is private; it can be accessed only from within the class itself and its companion object. The class name `Queue` is still public, so you can use it as a type, but you cannot call its constructor:

```
scala> new Queue(List(1, 2), List(3))  
<console>:4: error: constructor Queue cannot be accessed  
val unnamed0 = new Queue(List(1, 2), List(3) )  
^
```

Factory methods

Now that the primary constructor of class `Queue` can no longer be called from client code, there needs to be some other way to create new queues. One possibility is to add a secondary constructor that builds an empty queue, like this:

```
def this() = this(Nil, Nil)
```

As a refinement, the secondary constructor could take a list of initial queue elements as a variable length parameter:

```
def this(elems: T*) = this(elems.toList, Nil)
```

Another possibility is to add a factory method that builds a queue from such a sequence of initial elements. A neat way to do this is to define an object `Queue` which has the same name as the class being defined and which contains an `apply` method, like this:

```
object Queue {  
    // constructs a queue with initial elements as given by 'xs'  
    def apply[T](xs: T*) = new Queue[T](xs.toList, Nil)  
}
```

By placing this object in the same source file as class Queue you make the object a companion object of the class. You have seen in [Chapter 13](#) that a companion object has the same access rights as its class; that's why the `apply` method in object `Queue` could create a new `Queue` object, even though the constructor of class `Queue` is private.

Note that, because the factory method is called `apply`, clients can create queues with an expression such as `Queue(1, 2, 3)`. In fact, this expression expands to `Queue.apply(1, 2, 3)` using the normal rules of function objects. So `Queue` looks to clients as if it was a globally defined factory method. In reality, Scala has no globally visible methods; every method must be contained in an object or a class. However, using methods named `apply` inside global objects, one can support usage patterns that look like invocations of global methods.

An alternative: type abstraction

Private constructors and private members are one way to hide the initialization and representation of a class. Another, more radical way is to hide the class itself and only export a trait that reveals the public interface of the class. The code in [Figure 17.2](#) implements this design. There's a trait `Queue` which declares the methods `head`, `tail`, and `append`. All three methods are implemented in a subclass `QueueImpl`, which is itself a `private` inner class of object `Queue`. This exposes to clients the same information as before, but using a different technique: Now it's the whole implementation class which is hidden, instead of just individual constructors and methods.

17.3 Variance annotations

The combination of type parameters and subtyping poses some interesting questions. For instance, should `Queue[String]` be a subtype of `Queue[AnyRef]`? Intuitively, this seems OK, since a queue of `Strings` is a special case of a queue of `AnyRefs`. More generally, if `T` is a subtype of

```
trait Queue[T] {  
    def head: T  
    def tail: Queue[T]  
    def append(x: T): Queue[T]  
}  
  
object Queue {  
    def apply[T](xs: T*): Queue[T] =  
        new QueueImpl(xs.toList, Nil)  
  
    private class QueueImpl(  
        leading: List[T], trailing: List[T]) extends Queue[T] {  
  
        def normalize =  
            if (leading.isEmpty)  
                new QueueImpl(trailing.reverse, Nil)  
            else  
                this  
  
        def head: T =  
            normalize.leading.head  
  
        def tail: QueueImpl[T] = {  
            val q = normalize  
            new QueueImpl(leading.tail, q.trailing)  
        }  
  
        def append(x: T) =  
            new QueueImpl(leading, x :: trailing)  
    }  
}
```

Figure 17.2: Type abstraction for functional queues

type S then Queue[T] should be a subtype of Queue[S]. This property is called *co-variant* subtyping.

In Scala, generic types have by default non-variant subtyping. That is, with Queue defined as above, queues with different element types would never be in a subtype relation. However, you can demand co-variant subtyping of queues by changing the first line of the definition of class Queue as follows.

```
class Queue[+T] { ... }
```

Prefixing a formal type parameter with a + indicates that subtyping is co-variant in that parameter. Besides +, there is also a prefix - which indicates contra-variant subtyping. If Queue was defined

```
class Queue[-T] { ... }
```

then if T is a subtype of type S this would imply that Queue[S] is a subtype of Queue[T] (which in the case of queues would be rather surprising!).

In a purely functional world, many types are naturally co-variant. However, the situation changes once you introduce mutable data. To find out why, consider a simple type of one-element cells which can be read or written:

```
class Cell[T](init: T) {
    private[this] var current = init
    def get = current
    def set(x: T) { current = x }
}
```

This type is declared non-variant. For the sake of the argument, assume for a moment that it is declared covariant instead, as in `class Cell[+T] ...` and that this passes the Scala compiler (it doesn't, and we'll explain why shortly). Then you could construct the following problematic statement sequence:

```
val c1 = new Cell[String]("abc")
val c2: Cell[Any] = c1
c2.set(1)
val s: String = c1.get
```

Seen by itself, each of these four lines looks OK. Line 1 creates a cell of strings and stores in a value c1. Line 2 defines a new value c2 of type

Cell[Any] which is equal to c1. This is OK, since Cells are assumed to be covariant. Line 3 sets the value of cell c2 to 1. This is also OK, because the assigned value 1 is an instance of c2s element type Any. Finally, line 4 assigns the element value of c1 into a string. Nothing strange here, as both the sides are of the same type. But taken together, these four lines end up assigning the integer 1 to the string s. This is clearly a violation of type-soundness.

Which operation is to blame for the run-time fault? It must be the second one which uses co-variance. In fact, a Cell of String is *not* also a Cell of Any, because there are things one can do with a Cell of Any that one cannot do with a Cell of String: use set with an Int argument, for example.

In fact, if you pass the covariant version of Cell to the Scala compiler, you would get:

```
Cell.scala:7: error: covariant type T occurs in contravariant position
      in type T of value x
        def set(x: T) = current = x
                           ^
```

Variance and arrays

It's interesting to compare this behavior with arrays in Java. In principle, arrays are just like cells except that they can have more than one element. Nevertheless, arrays are treated as covariant in Java. You can try an example analogous to the cell interaction above with Java arrays:

```
// this is Java
String[] a1 = { "abc" }
Object[] a2 = a1
a2[0] = new Integer(17)
String s = a1[0]
```

If you try out this example, you will find that it typechecks, but executing the program will cause an `ArrayStore` exception to be thrown at the third line. In fact, Java stores the element type of the array at run-time. Then, every time an array element is updated, the new element value is checked against the stored type. If it is not an instance of that type, an `ArrayStore` exception is thrown.

You might ask why Java has adopted this design which seems both unsafe and expensive? When asked this question, James Gosling, the principal inventor of the Java language, answered that they wanted to have a simple means to treat arrays generically. For instance, they wanted to be able to write a method to sort all elements of an array, using a signature like the following:

```
void sort(Object[] a, Comparator cmp) { ... }
```

Covariance of arrays was needed so that arrays of arbitrary reference types could be passed to this `sort` method. Of course, with the arrival of Java generics, such a `sort` method can now be written with a type parameter, so the covariance of arrays is no longer necessary. However, it has persisted to this day.

Scala tries to be purer than Java in not treating arrays as covariant. Here's what you get if you translate the first two lines of the array example to Scala:

```
scala> val a1 = Array("abc")
a1: Array[java.lang.String] = [Ljava.lang.String;@14653a3
scala> val a2: Array[Any] = a1
<console>:5: error: type mismatch;
  found   : Array[java.lang.String]
  required: Array[Any]
      val a2: Array[Any] = a1
^
```

However, sometimes it is necessary to interact with legacy methods in Java that use an `Object` array as a means to emulate a generic array. For instance, you might want to call a `sort` method like the one described above with an array of `Strings` as argument. To make this possible, Scala lets you cast an array of `Ts` to an array of any supertype of `T`:

```
scala> val a2: Array[Object] = a1.asInstanceOf[Array[Object]]
a2: Array[Object] = [Ljava.lang.String;@14653a3
```

The cast will always succeed, but you might get `ArrayStore` exceptions afterwards, just as you would in Java.

Checking variance

Now that you have seen some examples where variance is unsound, you probably wonder which kind of class definitions need to be rejected and which definitions can be accepted. So far, all violations of type soundness involved some mutable field or array element. The purely functional implementation of queues, on the other hand, looks like a good candidate for a covariant data type. However, the following example shows that one can “engineer” an unsound situation even if there is no mutable field.

To set up the example, assume that queues as defined above are covariant. Then, create a subclass of queues which specializes the element type to `Int` and which overrides the `append` method:

```
class StrangeIntQueue extends Queue[Int] {  
    override def append(x: Int) = {  
        println(x * x)  
        super.append(x)  
    }  
}
```

The `append` method in `StrangeIntQueue` prints out the square of its (integer) argument before doing the append proper. Now, you can write a counter-example in two lines:

```
val x: Queue[Any] = new StrangeIntQueue  
x.append("abc")
```

The first of these two lines is valid, because `StrangeIntQueue` is a subclass of `Queue[Int]` and, assuming covariance of queues, `Queue[Int]` is a subtype of `Queue[Any]`. The second line is valid because one can append a `String` to a `Queue[Any]`. However, taken together these two lines have the effect of calling a non-existent multiply method on a string.

So, clearly, it’s not just mutable fields that make covariant types unsound. The problem is more general. One can show that as soon as a generic parameter type appears as the type of a method parameter, the containing data type may not be covariant. For queues, the `append` method violates this condition:

```
class Queue[+T] {
```

```
def append(x: T) =  
  ...  
}
```

Running a modified queue class like the one above through a Scala compiler would yield:

```
<console>:5: error: covariant type T occurs in contravariant position  
in type T of value x  
def append(x: T) =  
^
```

Mutable fields are a special case of the rule that disallows method parameters of covariant types. Remember that a mutable field `var x: T` is treated in Scala as a pair of a getter method `def x: T` and a setter method `def x_=(y: T)`. As you can see, the setter method has a parameter of the field's type `T`. So that type may not be covariant.

To verify correct use of variances, the Scala compiler classifies all positions in a class body as covariant, contravariant, or nonvariant. Positions are classified starting from the class definition itself and following paths towards all references to types in the class. Positions at the top-level of the class are classified as covariant. A *flip* operation changes the current classification from covariant into contravariant and *vice versa*. Flips are performed at the following two places:

- At a method parameter (both for value parameters and for type parameters).
- At a lower bound of an abstract type or type parameter.

A variance flip is also performed for a type argument `T` of a type such as `C[T]` if the corresponding parameter of `C` is defined to be contravariant. By contrast, if the formal parameter is defined to be covariant, the variance position stays as it is. Finally, if the formal parameter is defined to be nonvariant, every position inside `T` is also classified as nonvariant.

As a somewhat contrived example, consider the following class definition, where the variance of each type occurrence is annotated with `+` (for covariant) or `-` (for contravariant).

```
abstract class C[-T, +U] {
    def f[W >: List[U+]](x: T-, y: C[U+, T-])
        : C[C[U+, T-], U+]
}
```

The position of the type parameter W and the two value parameters x and y are all contravariant. The position of the lower bound $List[U]$ is covariant, because two flips bring a variance position back to itself. Looking at the result type of f , the position of the first $C[U, T]$ argument is contravariant, because C 's first type parameter T is defined contravariant. The type U inside this argument is again in covariant position (two flips), whereas the type T inside that argument is still in contravariant position. The variance positions of all the other occurrences of types T and U are computed in the same way.

The variance type checking rule itself is very simple: Every type parameter of a class which is declared covariant may only appear in covariant positions inside the class. A contravariant type parameter may only appear in contravariant positions. A nonvariant type parameter may appear anywhere. In the example above, both type parameters appear only in positions that match their variance declaration. So class C is type correct.

17.4 Lower bounds

Back to the `Queue` class. You have seen that the previous definition of `Queue[T]` cannot be made covariant in T because T appears as parameter type of the `append` method, and that's a contravariant position.

However, it's possible to generalize the `append` method using a lower bound:

```
class Queue[+T] {
    def append[U >: T](x: U) =
        new Queue(leading, x :: trailing)
    ...
}
```

In the revised definition of `append`, the position of the type T is covariant, because there are two flips between the class body and itself: One at the new type parameter U , the other at its lower bound. So, technically, `append` is

now variance correct. What's more, the new definition of `append` is arguably better (that is, more general) than the old one. Specifically, the new definition allows to append an arbitrary supertype `U` of the queue element type `T`. The result is then a queue of `Us`. Together with queue covariance, this gives the right kind of flexibility for modeling queues of different element types in a natural way.

For instance, assume there is a class `Fruit` with two subclasses, `Apple` and `Orange`. With the new definition of class `Queue`, it's possible to append an `Orange` to a `Queue[Apple]`. The result will be a `Queue[Fruit]`.

This shows that variance annotations and lower bounds play well together. They are a good example of *type-driven design*, where the types of an interface guide its detailed design and implementation. In the case of queues, you would probably not have thought of the refined implementation of `append` with a lower bound, but you might have decided to make queues covariant. In that case, the compiler would have pointed out the variance error for `append`. Correcting the variance error by adding a lower bound makes `append` more general and queues as a whole more usable.

This observation is also the primary reason why Scala prefers declaration-site variance over use-site variance as it is found in Java's wildcards. With use-site variance, you are on your own designing a class. It will be the clients of the class that need to put in the wildcards. If they get it wrong, some important instance methods will no longer be visible. Variance being a tricky business, users usually get it wrong. That explains why wildcards, and Java generics in general, are perceived by many to be overly complicated.

17.5 Contravariance

So far, all types were either covariant or nonvariant. But there are also cases where contravariance is natural. For instance, consider the following trait of output channels:

```
trait OutputChannel[-T] {  
    def write(x: T)  
}
```

Here, `T` is defined to be contravariant. So an output channel of `Objects`,

say, is a subtype of an output channel of `Strings`. This makes sense. To see why, consider what one can do with an `OutputChannel[String]`. The only supported operation is writing a `String` to it. The same operation can also be done on an `OutputChannel[Object]`. So it is safe to substitute an `OutputChannel[Object]` for an `OutputChannel[String]`. By contrast, it is not safe to use an `OutputChannel[String]` where an `OutputChannel[Object]` is required. After all, one can send any object to an `OutputChannel[Object]`, whereas an `OutputChannel[String]` requires that the written values are all strings.

This reasoning points to a general principle in type system design: It is safe to assume that a type `T` is a subtype of a type `U` if you can substitute a value of type `T` wherever a value of type `U` is required. This is called the *Liskov substitution principle*. The principle holds if `T` supports the same operations as `U` and all of `T`'s operations require less and provide more than the corresponding operations in `U`. In the case of output channels, an `OutputChannel[Object]` can be a subtype of an `OutputChannel[String]` because the two support the same `write` operation, and this operation requires less in `OutputChannel[Object]` than in `OutputChannel[String]`. “Less” means: the argument is only required to be an `Object` in the first case, whereas it is required to be a `String` in the second case.

Sometimes covariance and contravariance are mixed in the same type. A prominent example are Scala’s function traits. For instance, here is the trait of unary functions, which implements function types of the form `A => B`:

```
trait Function1[-S, +T] {  
    def apply(x: S): T  
}
```

The `Function1` trait is contravariant in the function argument type `S` and covariant in the result type `T`. This satisfies the Liskov substitution principle, because arguments are something that’s required, whereas results are something that’s provided.

17.6 Object-local data

The `Queue` class seen so far has a problem in that the `mirror` operation might repeatedly copy the trailing into the leading list if `head` is called several

```

class Queue[+T] private {
    private[this] var leading: List[T],
    private[this] var trailing: List[T]) {

    private def mirror() =
        if (leading.isEmpty) {
            while (!trailing.isEmpty) {
                leading = trailing.head :: leading
                trailing = trailing.tail
            }
        }

    def head: T = {
        mirror()
        leading.head
    }

    def tail: Queue[T] = {
        mirror()
        new Queue(leading.tail, trailing)
    }

    def append[U >: T](x: U) =
        new Queue(leading, x :: trailing)
}

```

Figure 17.3: Optimized functional queues

times in a row on a list where `leading` is empty. The wasteful copying could be avoided by adding some judicious side effects. Figure 17.3 presents a new implementation of `Queue` which performs at most one `trailing` to `leading` adjustment for any sequence of `head` operations.

What's different with respect to the previous version is that now `leading` and `trailing` are reassignable variables, and the `mirror` operation performs the reverse copy from `trailing` to `leading` as a side-effect on the current queue instead of returning a new queue. This side-effect is purely internal to the implementation of the `Queue` operation; since `leading` and `trailing`

are private variables, the effect is not visible to clients of Queue.

You might wonder whether this code passes the Scala type checker. After all, queues now contain two mutable fields of the covariant parameter type T. Is this not a violation of the variance rules? It would be indeed, except for the detail that `leading` and `trailing` have a `private[this]` modifier and are thus declared to be object-local. It turns out that variables that are accessed only from the object in which they are defined do not cause problems with variance. The intuitive explanation is that, in order to construct a case where variance would lead to type errors, one needs to have a reference to a value that has a statically weaker type than the type the value was defined with. For object-local values this is impossible.

Scala's variance checking rules contain a special case for object-local definitions. Such definitions are omitted when it is checked that a co- or contravariant type parameter occurs only in positions that have the same variance classification. Therefore, the code in [Figure 17.3](#) compiles without error. On the other hand, if you had left out the `[this]` qualifiers from the two `private` modifiers, you would see two type errors.

```
Queue.scala:1: error: covariant type T occurs in contravariant position
      in type List[T] of parameter of setter leading_=
          class Queue[+T] private (private var leading: List[T],
                                     ^
```

```
Queue.scala:1: error: covariant type T occurs in contravariant position
      in type List[T] of parameter of setter trailing_=
          private var trailing: List[T]) {
```

17.7 Conclusion

In this chapter you have seen several techniques for information hiding: Private constructors, factory methods, type abstraction, and object-local members. You have also learned how to specify data type variance and what it implies for class implementation. Finally, you have seen two techniques which help in obtaining rich variance annotations: lower bounds for method type parameters, and `private[this]` annotations for local fields and methods.

Chapter 18

Abstract Members and Properties

A member of a class or trait is *abstract* if the member does not have a complete definition in the class. Abstract members are supposed to be implemented in subclasses of the class in which they are defined. This idea is found in many object-oriented languages. For instance, Java lets you declare abstract methods. Scala also lets you declare such methods, as you have seen in chapter 10. But it goes beyond that and implements the idea in its full generality—besides methods you can also define abstract fields and even abstract types.

An example is the following trait `Abstract` which defines an abstract type `T`, an abstract method `transform`, an abstract value `initial`, and an abstract variable `current`.

```
trait Abstract {  
    type T  
    def transform(x: T): T  
    val initial: T  
    var current: T  
}
```

A concrete implementation of `Abs` needs to fill in definitions for each of its abstract members. Here is an example implementation that provides these definitions.

```
class Concrete extends Abstract {  
    type T = String  
    def transform(x: String) = x + x
```

```
val initial = "hi"
var current = initial
}
```

The implementation gives a concrete meaning to the type name T by defining it as an alias of type String. The transform operation concatenates a given string with itself, and the initial and current values are both set to "hi".

This example gives you a rough first idea of what kinds of abstract members exist in Scala. The next sections will present the details and explain what the new forms of abstract members are good for.

18.1 Abstract vals

An abstract val definition has a form like

```
val initial: String
```

It gives a name and type for a val, but not its value. This value has to be provided by a concrete value definition in a subclass. For instance, class Concrete implemented the value using

```
val initial = "hi"
```

You use an abstract value definition in a class when you do not know the correct value in the class, but you do know that the variable will have an unchangeable value in each instance of the class.

An abstract value definition resembles an abstract parameterless method definition such as

```
def initial: String
```

Client code refers to both the value and the method in exactly the same way, i.e. obj.initial. However, if initial is an abstract value, the client is guaranteed that obj.initial will yield the same value everytime it is referenced. If initial is an abstract method, that guarantee would not hold, because in that case initial could be implemented by a concrete method that returned a different value everytime it was called.

In other words, an abstract value constrains its legal implementation: Any implementation must be a val definition; it may not be a var or def

definition. Abstract method definitions, on the other hand, may be implemented by both concrete method definitions and concrete value definitions. So given a class,

```
abstract class A {  
    val v: String // 'v' for value  
    def m: String // 'm' for method  
}
```

the following class would be a legal implementation

```
class C1 extends A {  
    val v: String  
    val m: String // OK to override a 'def' with a 'val'  
}
```

But the following class would be in error:

```
class C2 extends A {  
    def v: String // ERROR: cannot override a 'val' with a 'def'  
    def m: String  
}
```

18.2 Abstract vars

Like an abstract val, an abstract var defines just a name and a type, but not an initial value. For instance, here is a class `AbstractTime` which defines two abstract variables named `hour` and `minute`.

```
trait AbstractTime {  
    var hour: Int  
    var minute: Int  
}
```

What should be the meaning of an abstract var like `hour` or `minute`? You have seen in [Chapter 16](#) that vars that are members of classes come equipped with getter and setter methods. This holds for concrete as well as abstract variables. If you define an abstract var `x`, you implicitly define a getter

method `x` and a setter method `x_=`. In fact, an abstract var is just a shorthand for a pair of getter and setter methods. There's no reassignable field to be defined – that will come in subclasses which define the concrete implementation of the abstract var. For instance, the definition of `AbstractTime` above is exactly equivalent to the following definition.

```
trait AbstractTime {  
    def hour: Int          // getter for 'hour'  
    def hour_=(x: Int)      // setter for 'hour'  
    def minute: Int         // getter for 'minute'  
    def minute_=(x: Int)    // setter for 'minute'  
}
```

18.3 Abstract types

In the beginning of this chapter you saw an abstract type declaration

```
type T
```

The rest of this chapter discusses what such an abstract type declaration means and what it's good for. Like all other abstract declarations this is a placeholder for something that will be defined concretely in subclasses. In this case, it is a type that will be defined further down the class hierarchy. So `T` above refers to a type that is at yet unknown at the point where it is defined. Different subclasses can provide different realizations of `T`.

Here is a well-known example where abstract types show up naturally. Suppose you are given the task to model eating habits of animals. You might start with a class `Food` and a class `Animal` with an `eat` method:

```
class Food {}  
abstract class Animal {  
    def eat(food: Food)  
}
```

You would then specialize these two classes to a class of Cows which eat Grass:

```
class Grass extends Food {}  
class Cow extends Animal {  
    override def eat(food: Grass) {}  
}
```

However, if you tried to compile the new classes you'd get compilation errors:

```
error: class Cow needs to be abstract, since method eat is not  
defined  
class Cow {  
    ^  
  
error: method eat overrides nothing  
override def eat(food: Grass)  
    ^
```

What happened is that the `eat` method in class `Cow` does not override the `eat` method in class `Animal` because its parameter type is different—it's `Grass` in class `Cow` vs. `Food` in class `Animal`.

Some people have argued that the type system is unnecessarily strict in refusing these classes. They have said that it should be OK to specialize a parameter of a method in a subclass. However, if the classes were allowed as written, you could get yourself in unsafe situations very quickly. For instance, the following would pass the type checker.

```
class Fish extends Food  
val cow: Animal = new Cow  
cow eat (new Fish)
```

The program would compile, because Cows are Animals and Animals do have an `eat` method that accepts any kind of Food, including Fish. But surely it would do a cow no good to eat a fish!

What you need to do instead is apply some more precise modelling. Animals do eat Food but what kind of Food depends on the Animal. This can be neatly expressed with an abstract type:

```
abstract class Animal {  
    type SuitableFood <: Food  
    def eat(food: SuitableFood)
```

```
}
```

With the new class definition, an Animal can eat only food that's suitable. What food is suitable cannot be determined on the level of the Animal class. That's why SuitableFood is modelled as an abstract type. The type has an upper bound Food, which is expressed by the `<: Food` clause. This means that any concrete instantiation of SuitableFood in a subclass must be a subclass of Food. For example, you would not be able to instantiate SuitableFood with class IOException.

With Animal defined, you can now progress to cows:

```
class Cow extends Animal {  
    type SuitableFood = Grass  
    def eat(food: Grass) {}  
}
```

Class Cow fixes its SuitableFood to be Grass and also defines a concrete eat method for this kind of food. These new class definitions compile without errors. If you tried to run the “cows-that-eat-fish” counterexample with the new class definitions you'd get the following:

```
scala> class Fish extends Food  
defined class Fish  
scala> val cow: Animal = new Cow  
cow: Animal = Cow@1fb069  
  
scala> cow eat (new Fish)  
<console>:7: error: type mismatch;  
      found   : Fish  
      required: cow.SuitableFood  
              cow eat (new Fish)  
                  ^
```

Path-dependent types

Have a look at the last error message: What's interesting about it is the type required by the eat method: `cow.SuitableFood`. This type consists of an object reference (`cow`) which is followed by a type field `SuitableFood` of

the object. So this shows that objects in Scala can have types as members. `cow.SuitableFood` means “the type `SuitableFood` which is a member of the `cow` object,” or otherwise said, the type of food that’s suitable for `cow`. A type like `cow.SuitableFood` is called a *path dependent type*. The word “path” here means a reference to an object. It could be a single name or a longer access path such as in `swiss.cow.SuitableFood`.

As the term “path-dependent type” says, the type depends on the path: in general, different paths give rise to different types. For instance, if you had two animals `cat` and `dog`, their `SuitableFood`s would not be the same: `cat.SuitableFood` is incompatible with `dog.SuitableFood`. The case is different for Cows however. Because their `SuitableFood` type is defined to be an alias for class `Grass`, the `SuitableFood` types of two cows are in fact the same.

A path-dependent type resembles a reference to an inner class in Java, but there is a crucial difference: A path-dependent type names an outer *object*, whereas a reference to an inner class names an outer *class*. References to inner classes as in Java can also be expressed in Scala, but they are written differently. Assume two nested classes `Outer` and `Inner`:

```
class Outer {  
    class Inner  
}
```

In Scala, the inner class is addressed using the expression `Outer # Inner` instead of `Outer.Inner` in Java. The `.` syntax is reserved for objects. For instance, assume two objects:

```
val o1, o2 = new Outer
```

Then `o1.Inner` and `o2.Inner` would be two path-dependent types (and they would be different types). Both of these types would conform to the more general type `Outer # Inner` which represents the `Inner` class with an arbitrary outer object of type `Outer`.

18.4 Case study: Currencies

The rest of this chapter presents a case-study which explains how abstract types can be used in Scala. The task is to design a class `Currency`. A

typical instance of `Currency` would represent an amount of money in Dollars or Euros or Yen, or in some other currency. It should be possible to do some arithmetic on currencies. For instance, you should be able to add two amounts of the same currency. Or you should be able to multiply currency amount by a factor representing an interest rate.

These thoughts lead to the following first design for a currency class:

```
// A first (faulty) design of the Currency class
abstract class Currency {
    val amount: Long
    def designation: String
    override def toString = amount + " " + designation
    def +(that: Currency): Currency = ...
    def *(x: Double): Currency = ...
}
```

The amount of a currency is the number of currency units it represents. This is a field of type `Long` so that very large amounts of money such as the market capitalization of Google or Microsoft can be represented. It's left abstract here, waiting to be defined when one talks about concrete amounts of money. The designation of a currency is a string that identifies it. The `toString` method of class `Currency` indicates an amount and a designation. It would yield results such as

```
79 USD
11000 Yen
99 Euro
```

Finally, there are methods ‘+’ for adding currencies and ‘*’ for multiplying a currency with a floating point number. A concrete currency value could be created by instantiating `amount` and `designation` with concrete values. For instance:

```
new Currency {
    val amount = 99.95
    def designation = "USD"
}
```

This design would be OK if all we wanted to model was a single currency such as only Dollars or only Euros. But it fails once we need to deal with several currencies. Assume you model Dollars and Euros as two subclasses of class `Currency`.

```
abstract class Dollar extends Currency {  
    def designation = "USD"  
}  
abstract class Euro extends Currency {  
    def designation = "Euro"  
}
```

At first glance this looks reasonable. But it would let you add Dollars to Euros. The result of such an addition would be of type `Currency`. But it would be a funny currency that was made up of a mix of Euros and Dollars. What you want instead is a more specialized version of ‘+’: When implemented in class `Dollar`, it should take `Dollar` arguments and yield a `Dollar` amount; when implemented in class `Euro`, it should take `Euro` arguments and yield `Euro` results. So the type of the addition method changes depending in which class you are in. However, you would like to write the addition method just once, not each time a new currency is defined.

In Scala, there’s a simple technique to deal with situations like this: If something is not known at the point where a class is defined, make it abstract in the class. This applies to both values and types. In the case of currencies, the exact argument and result type of the addition method are not known, so it is a good candidate for an abstract type. This would lead to the following sketch of class `AbstractCurrency`:

```
// A second (still imperfect) design of the Currency class  
abstract class AbstractCurrency {  
    type Currency <: AbstractCurrency  
    val amount: Long  
    def designation: String  
    override def toString = amount+" "+designation  
    def +(that: Currency): Currency = ...  
    def *(x: Double): Currency = ...  
}
```

The only difference to the situation before is that the class is now called `AbstractCurrency`, and that it contains an abstract type `Currency`, which represents the real currency in question. A concrete subclass of `AbstractCurrency` would need to fix the `Currency` type to refer to the concrete subclass itself, thereby “tying the knot”. For instance, here is a new version of `Dollar` which extends `AbstractCurrency`.

```
class Dollar extends AbstractCurrency {  
    type Currency = Dollar  
    def designation = "USD"  
}
```

This design is workable, but it is still not perfect. One problem is hidden by triple dots which indicate the missing method definitions of ‘+’ and ‘*’ in class `AbstractCurrency`. In particular, how should addition be implemented in this class? It’s easy enough to calculate the correct amount of the new currency as `this.amount + that.amount`, but how to convert the amount into a currency of the right type? You might try something like

```
def +(that: Currency): Currency = new Currency {  
    val amount = this.amount + that.amount  
}
```

However, this would not compile:

```
error: class type required  
def +(that: Currency): Currency = new Currency {  
    ^
```

One of the restrictions of Scala’s treatment of abstract types is that you cannot create an instance of an abstract type, or have an abstract type as a supertype of another class.¹ So the compiler refused the example code above.

However, you can work around this restriction using a *factory method*. Instead of creating an instance of an abstract type directly, define an abstract method which does it. Then, wherever the abstract type is fixed to be some concrete type, you also need to give a concrete implementation of the factory method. For class `AbstractCurrency`, this would look as follows:

¹ There’s some promising recent research on *virtual classes*, which would allow this, but this is not currently supported in Scala.

```
abstract class AbstractCurrency {  
    type Currency <: AbstractCurrency // abstract type  
    def make(amount: Long): Currency // factory method  
    ...  
        // rest of class  
}
```

A design like this could be made to work, but it looks rather suspicious. Why place the factory method *inside* class `AbstractCurrency`? This looks like a code smell, for at least two reasons. First, if you have some amount of currency (say: one Dollar), you also hold in your hand the ability to make more of the same currency, using code such as:

```
myDollar.make(100) // here are a hundred more!
```

In the age of color copying this might be a tempting scenario, but hopefully not one which you would be able to do for very long without being caught. The second problem with this code is that you can make more `Currency` objects if you already have a reference to a `Currency` object, but how do you get the first object of a given `Currency`? You'd need another creation method, which does essentially the same job as `make`. So you have a case of code duplication, which is a sure sign of a code smell.

The solution, of course, is to move the abstract type and the factory method outside class `AbstractCurrency`. You need to create another class which contains the `AbstractCurrency` class, the `Currency` type, and the `make` factory method. Let's call this object a `CurrencyZone`:

```
abstract class CurrencyZone {  
    type Currency <: AbstractCurrency  
    def make(x: Long): Currency  
    abstract class AbstractCurrency {  
        val amount: Long  
        def designation: String  
        override def toString = amount+" "+designation  
        def +(that: Currency): Currency =  
            make(this.amount + that.amount)  
        def *(x: Double): Currency =  
            make((this.amount * x).toLong)  
    }  
}
```

```
}
```

An example of a concrete `CurrencyZone` is the US. You could define this as follows:

```
object US extends CurrencyZone {  
    abstract class Dollar extends AbstractCurrency {  
        def designation = "USD"  
    }  
    type Currency = Dollar  
    def make(x: Long) = new Dollar { val amount = x }  
}
```

Here, `US` is an object that extends `CurrencyZone`. It defines a class `Dollar` which is a subclass of `AbstractCurrency`. So the type of money in this zone is `US.Dollar`. The `US` object also fixes the type `Currency` to be an alias for `Dollar`, and it gives an implementation of the `make` factory method to return a dollar amount.

This is a workable design. There are only some refinements to be added. The first refinement concerns subunits. So far, every currency was measured in a single unit: Dollars, Euros, or Yen. However, most currencies have sub-units: for instance, in the US, it's dollars and cents. The most straightforward way to model cents is to have the `amount` field in `US.Currency` represent cents instead of Dollars. To convert back to Dollars, it's useful to introduce a field `CurrencyUnit` in class `CurrencyZone` which contains the amount of currency of one standard unit in that currency. Class `CurrencyZone` gets thus augmented like this:

```
class CurrencyZone {  
    ...  
    val CurrencyUnit: Currency  
}
```

The `US` object then defines the quantities `Cent`, `Dollar`, and `CurrencyUnit` as follows:

```
object US extends CurrencyZone {  
    abstract class Dollar extends AbstractCurrency  
    type Currency = Dollar
```

```
def make(x: Long) = new Dollar {  
    val amount = x  
    def designation = "USD"  
}  
val Cent = make(1)  
val Dollar = make(100)  
val CurrencyUnit = Dollar  
}
```

This definition is just like the previous definition of the US object, except that it adds three new fields: The field `Cent` represents an amount of 1 `US.Currency`. It's an object analogous to a copper coin of one cent. The field `Dollar` represents an amount of 100 `US.Currency`. So the `US` object now defines the name `Dollar` in two ways: The *type* `Dollar` represents the generic name of the `Currency` valid in the `US` currency zone. By contrast the *variable* `Dollar` represents a single `US` `Dollar`, analogous to a greenback bill. The third field definition of `CurrencyUnit` specifies that the standard currency unit in the `US` zone is the `Dollar`.

The `toString` method in class `Currency` also needs to be adapted to take subunits into account. For instance, the sum of ten dollars and twenty three cents should print as a decimal number: 10.23 USD. To achieve this, you implement `Currencys` `toString` method as follows:

```
override def toString =  
    amount format "%." + decimals(CurrencyUnit.amount) + "f"
```

Here, `format` is a method which Scala adds to the standard `String` class. It returns a string which is formatted according to a format string which is given as the method's right-hand operand. Format strings are as for Java's `String.format` method. For instance, the format string `%.2f` formats a number with two decimal digits. The format string above is assembled by calling the `decimals` method on `CurrencyUnit.amount`. This method returns the number of decimal digits of a decimal power minus one. For instance, `decimals(10)` is 1, `decimals(100)` is 2, and so on. The `decimals` method is implemented by a simple recursion:

```
private def decimals(l: Long): Int =  
    if (l == 1) 0 else 1 + decimals(l / 10)
```

Here are some other currency zones:

```
object European extends CurrencyZone {  
    abstract class Euro extends AbstractCurrency  
    type Currency = Euro  
    def make(x: Long) = new Euro {  
        val amount = x  
    }  
    def designation = "EUR"  
}  
val Euro = make(100)  
val Cent = make(1)  
val CurrencyUnit = Euro  
}  
  
object Japan extends CurrencyZone {  
    abstract class Yen extends AbstractCurrency  
    type Currency = Yen  
    def make(x: Long) = new Yen {  
        val amount = x  
        def designation = "JPY"  
    }  
    val Yen = make(1)  
    val CurrencyUnit = Yen  
}
```

As another refinement you can add a currency conversion feature to the model. As a first step, write a Converter object which contains applicable exchange rates between currencies. For instance:

```
object Converter {  
    var exchangeRate = Map(  
        "USD" -> Map("USD" -> 1.0, "EUR" -> 0.7596,  
                      "JPY" -> 1.211, "CHF" -> 1.223),  
        "EUR" -> Map("USD" -> 1.316, "EUR" -> 1.0,  
                      "JPY" -> 1.594, "CHF" -> 1.623),  
        "JPY" -> Map("USD" -> 0.8257, "EUR" -> 0.6272,  
                      "JPY" -> 1.0, "CHF" -> 1.018),  
        "CHF" -> Map("USD" -> 0.8108, "EUR" -> 0.6160,
```

```
    "JPY" -> 0.982 , "CHF" -> 1.0 )  
)  
}
```

Then, add a conversion method `from` to class `Currency`, which converts from a given source currency into the current `Currency` object:

```
def from(other: CurrencyZone#AbstractCurrency): Currency =  
  make(Math.round(other.amount.toDouble * Converter.exchangeRate  
    (other.designation)(this.designation)))
```

The `from` method takes another completely arbitrary currency as argument. That's expressed by its formal parameter type is `CurrencyZone#AbstractCurrency`, which stands for an `AbstractCurrency` type in some arbitrary and unknown `CurrencyZone`. It produces its result by multiplying the amount of the other currency with the exchange rate between the other and the current currency.

Here's the code of the final version of the `CurrencyZone` class:

```
abstract class CurrencyZone {  
  type Currency <: AbstractCurrency  
  protected def make(x: Long): Currency  
  val CurrencyUnit: Currency  
  private def decimals(l: Long): Long =  
    if (l == 1) 0 else 1 + decimals(l / 10)  
  abstract class AbstractCurrency {  
    val amount: Long  
    def designation: String  
    def +(that: Currency): Currency =  
      make(this.amount + that.amount)  
    def -(that: Currency): Currency =  
      make(this.amount - that.amount)  
    def *(that: Double) =  
      make((this.amount * that).toLong)  
    def /(that: Double) =  
      make((this.amount / that).toLong)  
    def /(that: Currency) =  
      this.amount.toDouble / that.amount
```

```
override def toString =  
  (amount.toDouble / CurrencyUnit.amount.toDouble) format  
    "%." + decimals(CurrencyUnit.amount) + "f"  
def from(other: CurrencyZone#AbstractCurrency): Currency =  
  make(Math.round(  
    other.amount.toDouble *  
    Converter.exchangeRate(other.designation)(this.designation)))  
}  
}
```

You can test the class in the Scala command shell. Let's assume that the `CurrencyZone` class and all concrete `CurrencyZone` objects are defined in a package `currencies`. The first step is to import everything in this package into the command shell:

```
scala> import currencies._
```

We can then do some currency conversions:

```
scala> Japan.Yen from US.Dollar * 100  
res0: currencies.Japan.Currency = 12110  
scala> European.Euro from res0  
res1: currencies.European.Currency = 75.95  
scala> US.Dollar from res1  
res2: currencies.US.Currency = 99.95
```

The fact that we obtain almost the same amount after three conversions implies that these are some pretty good exchange rates!

You can also add up values of the same currency:

```
scala> US.Dollar * 100 + res2  
res4: currencies.US.Currency = 199.95
```

On the other hand, you cannot add amounts of different currencies:

```
scala> US.Dollar + European.Euro  
<console>:7: error: type mismatch;  
       found   : currencies.European.Euro  
       required: currencies.US.Currency
```

US.Dollar + European.Euro

So the type abstraction has done its job—it prevents us from performing calculations which are unsound. Failures to convert correctly between different units may seem like trivial bugs, but they have caused many serious systems faults. An example is the crash of the Mars Climate Orbiter spacecraft on Sep 23rd, 1999, which was caused because one engineering team used metric units while another used English units. If units had been coded in the same way as Currencys are coded in this chapter, this error would have been detected by a simple compilation run. Instead, it caused the crash of the orbiter after a near 10-month voyage.

18.5 Conclusion

Scala offers systematic and very general support for object-oriented abstraction: It enables you to not only abstract over methods, but also to abstract over values, variables, and types. This chapter has shown how to make use of abstract members. You have seen that they support a simple yet effective principle for systems structuring: When designing a class make everything which is not yet known at the level of a class into an abstract member. This principle applies to all sorts of members: methods and variables as well as types.

The chapter has also shown how variables which are members of some class come equipped with setters and getters. You have seen how these getters and setters can be used to implement properties.

Chapter 19

Implicit Conversions and Parameters

There's a fundamental difference between your own code and libraries of other people: You can change or extend your own code as you wish, but if you want to use someone else's libraries you usually have to take them as they are.

A number of constructs have sprung up in programming languages to alleviate this problem. Ruby has modules, and Smalltalk lets packages add to each other's classes. These are very powerful, but also dangerous, in that you modify the behavior of a class for an entire application, some parts of which you might not know. C# 3.0 has static extensions methods, which are more local, but also more restrictive in that you can only add methods, not fields or interfaces to a class.

Scala has implicit parameters and conversions. They can make existing libraries much more pleasant to deal with by letting you leave out tedious code that is more obvious than useful. Used tastefully, this results in code that is focused on the interesting, non-trivial parts of your program. This chapter shows you how implicits work, and presents some of the most common ways they are used.

19.1 Implicit conversions

Here's a first example. One of the central collection traits in Scala is `RandomAccessSeq[T]`, which describes random access sequences over elements of type T. `RandomAccessSeqs` have most of the utility methods which you know from arrays or lists: `take`, `drop`, `map`, `filter`, `exists`,

or `mkString` are just some examples.

To make a new random access sequence, you simply extend trait `RandomAccessSeq`. You only need to define two methods which are abstract in the trait: `length` and `apply`. You then get implementations of all the other useful methods in the trait “for free”.

So far so good. This works fine if you are about to define new classes, but what about existing ones? Maybe you’d like to also treat classes in other people’s libraries as random access sequences, even if the designers of those libraries had not thought of making their classes extend `RandomAccessSeq`. For instance, a `String` in Java would make a good example of a `RandomAccessSeq[Char]`, but unfortunately Java’s `String` class does not inherit from Scala’s `RandomAccessSeq` trait.

In situations like this, implicits can help. To make a `String` into a `RandomAccessSeq`, you can define an implicit conversion between those two types:

```
implicit def stringWrapper(s: String) =  
  new RandomAccessSeq[Char] {  
    def length = s.length  
    def apply(i: Int) = s.charAt(i)  
  }
```

That’s it.¹ The implicit conversion is just a normal method, the only thing that’s special is the `implicit` modifier at the start. You can apply the conversion explicitly to transform `Strings` to `RandomAccessSeqs`:

```
scala> stringWrapper("abc") exists ('c' == _)  
res1: Boolean = true
```

But you can also leave out the conversion and *still* get the same behavior:

```
scala> "abc" exists ('c' == _)  
res2: Boolean = true
```

What goes on here under the covers is that the Scala compiler inserts the `stringWrapper` conversion for you. So in effect it converts the last expres-

¹In fact, the standard `Predef` object defines already a `stringWrapper` conversion with similar functionality, so in practice you can use this conversion instead of defining your own.

sion above to the one before. But on the surface, it's as if Java's Strings had acquired all the useful methods of trait RandomAccessSeq.

This aspect of implicits is similar to extension methods in C#, which also allow you to add new methods to existing classes. However, implicits can be far more concise than extension methods. For instance, you only need to define the `length` and `apply` methods in the `stringWrapper` conversion, and this gives you all other methods in `RandomAccessSeq` for free. With extension methods you'd need to define every one of these methods again. This duplication makes code harder to write, and, more importantly, harder to maintain. Imagine someone adds a new method to `RandomAccessSeq` sometimes in the future. If all you have is extension methods, you'd have to chase down all `RandomAccessSeq` "copycats" one by one, and add the new method in each. If you forget one of the copycats, your system will become inconsistent. Talk about a maintenance nightmare! By contrast, with Scala's implicits, all conversions pick up the newly added method automatically.

Another advantage of implicit conversions is that they support conversions into the target type. For instance, suppose you write a method `printWithSpaces` which prints all characters in a given random access sequence with spaces in between them:

```
scala> def printWithSpaces(seq: RandomAccessSeq[Char]) =  
    seq mkString " "
```

Because Strings are implicitly convertible to `RandomAccessSeq`s, you can pass a string to `printWithSpaces`:

```
scala> printWithSpaces("xyz")  
res3: String = x y z
```

The last expression is equivalent to the following one, where the conversion shows up explicitly:

```
scala> printWithSpaces(stringWrapper("xyz"))  
res4: String = x y z
```

This section has shown you some of the power of implicit conversions, and how they let you "dress up" existing libraries. In the next sections you'll learn the rules that determine when implicit conversions are tried and how they are found.

19.2 The fine print

Implicit definitions are those that the compiler is allowed to insert into a program in order to fix any of its type errors. For example, if `x + y` does not type check, then the compiler might change it to `convert(x) + y`. If `convert` changes `x` into something that has a `+` method, then this change might fix a program so that it type checks and runs correctly. If `convert` really is just a simple conversion function, then leaving it out of the source code can be a clarification.

Implicit conversions are governed by the following general rules.

Marking Rule: Only definitions marked implicit are available. The `implicit` keyword is used to mark which declarations the compiler may use as implicits. You can use it to mark any variable, function, or object definition, just like this:

```
implicit def int2string(x: Int) = x.toString
```

The compiler will only change `x + y` to `convert(x) + y` if `convert` is marked as `implicit`. This way, you avoid the confusion that would result if the compiler picked random functions that happen to be in scope and inserted them as “conversions.” The compiler will only select among the things you have explicitly marked as conversions.

Scope Rule: An inserted implicit conversion must be a single identifier or be associated with the source or target type of the conversion. The compiler will usually not insert a conversion of the form `foo.convert`. It will not expand `x + y` to `foo.convert(x) + y`. Any conversion must be available in the current scope via a single identifier. If you want to make `foo.convert` available as an implicit, then you need to import it. In fact, it is common for libraries to include a `Preamble` object including a number of useful implicit conversions. Code that uses the library can then do a single `insert Preamble._` to access the library’s implicit conversions.

There’s one exception to this “single identifier” rule. To pick up an implicit conversion the compiler will also look in the companion modules of the source or expected target types of the conversion. For instance, you could package an implicit conversion from `X` to `Y` in the companion module of class `X`:

```
object X {  
    implicit def XToY(x: X): Y = ...  
}  
class X { ... }
```

In that case, the conversion `XToY` is said to be *associated* to the type `X`. The compiler will find such an associated conversion everytime it needs to convert from an instance of type `X`. There's no need to import the conversion separately into your program.

The Scope Rule helps with modular reasoning. When you read code in one file, the only things you need to consider from other files are those that are either imported or are explicitly referenced through a fully qualified name. This benefit is at least as important for implicits as it is for explicitly written code. If implicits took effect system-wide, then to understand a file you would have to know about every implicit introduced anywhere in the program!

Non-Ambiguity Rule: **An implicit conversion is never inserted unless there is no other possible conversion to insert.** If the compiler has two options to fix `x + y`, say using either `convert1(x) + y` or `convert2(x) + y`, then it will report an error and refuse to choose between them. It would be possible to define some kind of “best match” rule that prefers some conversions over others. However, such choices lead to really obscure code. Imagine the compiler chooses `convert2`, but you are new to the file and are only conscious of `convert1`—you could spend a lot of time thinking a different conversion had been applied!

In cases like this, one option is to remove one of the imported implicits so that the ambiguity is removed. If you prefer `convert2`, then remove the import of `convert1`. Alternatively, you can write your desired conversion explicitly: `convert2(x) + y`.

One-at-a-time Rule: **Only one implicit is tried.** The compiler will never convert `x + y` to `convert1(convert2(x)) + y`. Doing so would cause compile times to increase dramatically on erroneous code, and it would increase the difference between what the programmer writes and what the program actually does. For sanity’s sake, the compiler does not insert further implicit conversions when it is already in the middle of trying another implicit.

However, it's possible to circumvent this restriction by having implicits take implicit parameters; see below.

Explicit-First Rule: Whenever code type checks as it is written, no implicits are attempted. The compiler will not change code that already works. A corollary of this rule is that you can always replace implicit identifiers by explicit ones, thus making the code longer but with less apparent ambiguity. You can trade between these choices on a case by case basis. Whenever you see code that seems repetitive and verbose, implicit conversions can help you decrease the tedium. Whenever code seems terse to the point of obscurity, you can insert conversions explicitly. The amount of implicits you leave the compiler to insert is ultimately a matter of style.

Naming an implicit conversion. Implicit conversions can have arbitrary names. The name of an implicit conversion matters only in two situations: if you want to write it explicitly in a method application, and for determining which implicit conversions are available at any place in the program.

To illustrate the second point, say you want to turn strings automatically into text labels of a GUI. To do this, you define an implicit conversion like the following:

```
import javax.swing._  
trait GUIFramework {  
    implicit def stringToLabel(s: String): JLabel =  
        new JLabel(s)  
    ...  
}
```

You have put the conversion in a trait `GUIFramework`. Whenever your application inherits that trait, the implicit conversion becomes available as a single identifier, and you can use strings directly as labels.

However, you might want to change the way strings are converted to labels in some other program component. For instance, you might define a subtrait `WindowsFramework` of `GUIFramework`. Inside `WindowsFramework` all implicitly generated labels should always have the Windows “look and feel”, which is represented by the `WindowsLabelUI` value. So you write:

```
trait WindowsFramework extends GUIFramework {  
    val WindowsLabelUI = ...  
    implicit def stringToLabel(s: String): JLabel = {  
        val label = super.stringToLabel(s)  
        label.setUI(WindowsLabelUI)  
        label  
    }  
    ...  
}
```

Now, every class inheriting from `WindowsFramework` will pick up the new `stringToLabel` conversion, which overrides the old one in `GUIFramework`. Here it is important that the names of the two implicit conversions match, because that way, the `GUIFramework` conversion is hidden by one in `WindowsFramework`. If you had picked a different name for the conversion in `WindowsFramework`, there would be *two* conversions available everytime a string needs to be converted to a label. You would then get compiler errors signalling violations of the non-ambiguity rule.

Where implicits are tried. There are three places implicits are used in the language: conversions to an expected type, conversions of the receiver of a selection, and implicit parameters. Implicit conversions to an expected type let you use one type in a context where a different type is expected. For example, you might have an `String` but want to use it as a `RandomAccessSeq[Char]`. Conversions of the receiver let you adapt the receiver of a method call if the method is not applicable on the original type. An example is `"abc".exists`, which is converted to `stringWrapper("abc").exists` because the `exists` method is not available on `Strings` but is available on `RandomAccessSeqs`. Implicit parameters, on the other hand, are usually used to provide more information to the callee about what the caller wants. Implicit parameters are especially useful with generic functions, where the callee might otherwise know nothing at all about the type of one or more arguments. The following three sections will each discuss one of these kinds of implicits.

19.3 Implicit conversion to an expected type

Implicit conversions to an expected type are the first place that the compiler will use implicits. The rule is simple. Whenever the compiler sees an X, but needs a Y, it will look for an implicit function that converts X's to Y's. For example, normally a double cannot be used as an integer, because it loses precision:

```
scala> val i: Int = 3.5
<console>:8: error: type mismatch;
           found   : Double(3.5)
           required: Int
    val i: Int = 3.5
           ^
```

However, you can define an implicit conversion to smooth this over:

```
scala> implicit def double2int(x: Double) = x.toInt
double2int: (Double)Int

scala> val i: Int = 3.5
i: Int = 3
```

What happens here is that the compiler sees a double, specifically 3.5, in a context where it requires an integer. Before giving up, it searches for an implicit conversion from doubles to integers. In this case, it finds one: `double2int`. It then inserts a call to `double2int` automatically. Behind the scenes, the code becomes:

```
val i: Int = double2int(3.5)
```

This is literally an *implicit* conversion. The programmer does not explicitly ask for conversion. Instead, you mark `double2int` as an available implicit conversion, and then the compiler automatically uses it wherever it needs to convert from a double to an integer.

19.4 Converting the receiver

Implicit conversions also apply to the receiver of a method call, giving two major applications that might not be obvious. These receiver conversions

allow smoother integration of a new class into an existing class hierarchy, and they also support writing domain-specific languages (DSLs) within the language.

To see how it works, suppose you write down `foo.doit`, and `foo` does not have a method named `doit`. The compiler will try to insert conversions before giving up. In this case, the conversion needs to apply to the receiver, `foo`. The compiler will act as if the expected “type” of `foo` were “has a method named `doit`.” This “has a `doit`” type cannot be expressed in Scala notation, but it is there conceptually and is why the compiler will insert an implicit conversion in this case.

Interoperating with new types

Receiver conversions have two major applications. One of them supports defining a new type, where you want to let people freely use some existing type as if it were also a member of the new type. Take for example the type of Rational numbers defined in [Chapter 6](#). Here’s an outline of that class again:

```
class Rational(n: Int, d: Int) {  
    ...  
    def +(that: Rational): Rational = ...  
    def +(that: Int): Rational = ...  
}
```

Class `Rational` has two overloaded variants of the ‘`+`’ method, which take `Rationals` and `Ints`, respectively, as arguments. So you can either add two rational numbers or a rational number and an integer:

```
scala> val oneHalf = new Rational(1, 2)  
oneHalf: Rational = 1/2  
scala> oneHalf + oneHalf  
res6: Rational = 1/1  
scala> oneHalf + 1  
res7: Rational = 3/2
```

What about an expression like `1 + oneHalf`, however? This expression is tricky because the receiver, `1`, does not have a suitable `+` method. So the following gives an error:

```
scala> 1 + oneHalf
<console>:6: error: overloaded method value + with alternatives
  (Double)Double <and> ... cannot be applied to (Rational)
        1 + oneHalf
               ^
```

To allow this kind of mixed arithmetic, you need to define an implicit conversion from `Int` to `Rational`:

```
scala> implicit def intToRational(x: Int) = new Rational(x, 1)
      intToRational: (Int)Rational
```

With the conversion in place, converting the receiver does the trick.

```
scala> 1 + oneHalf
      res10: Rational = 3/2
```

What happens behind the scene here is that Scala compiler first tries to type-check the expression `1 + oneHalf` as it is. This fails because `Int` has several `'+'` methods, but none that takes a `Rational` argument. Next, the compiler searches an implicit conversion from `Int` to another type that has a `'+'` method which can be applied to a `Rational`. It finds your conversion and applies it, yielding

```
intToRational(1) + oneHalf
```

Simulating new syntax

Another major application of implicit conversions is to simulate adding new syntax. Recall that you can make a `Map` using syntax like this:

```
Map(1->"one", 2->"two", 3->"three")
```

Have you wondered how the `->` is supported? It is not syntax! Instead, `->` is a method of a class called `RichAny`, located in package `scala.runtime`. The standard Scala preamble (`scala.Predef`) defines an implicit conversion from `Any` to `RichAny`, and `RichAny` includes a `->` method.

This `RichFoo` pattern is common in libraries that provide a syntax-like extension to the language, so you should be ready to recognize it when you see it. Whenever you see someone calling methods that appear not to exist in the receiver class, they are probably using implicits. Likewise, whenever you see a class named `RichSomething`, you can expect that the programmer is adding syntax-like methods to type `Something`, so it may be worth a quick skim to see what methods it has. You have seen it already for the basic types described in [Chapter 5](#). The `RichFoo` pattern applies more widely, however.

Stepping back, this simulated new syntax means you can define a DSL right within Scala. Instead of writing a parser and interpreter, you can provide a DSL by writing a library.

19.5 Implicit parameters

The other place the compiler inserts implicits is within parameter lists. The compiler will sometimes replace `foo(x)` with `foo(x)(y)`, or `new Foo(x)` with `new Foo(x)(y)`, thus adding a missing parameter list to complete a function call. For this usage, not only must the inserted identifier `(y)` be marked `implicit`, but also the formal parameter list in `foo`'s or `Foo`'s definition be marked as `\implicit@`.

Here is a simple example. The following `printSomething` function prints whatever its argument is.

```
scala> def printSomething(implicit x: Int) = println(x)
printSomething: (implicit Int)Unit
```

This function can be called just like any other function:

```
scala> printSomething(10)
10
```

However, you can also set a parameter implicitly:

```
scala> implicit val favoriteNumber = 4
```

```
favoriteNumber: Int = 4
```

```
scala> printSomething  
4
```

The most common use of these implicit parameters is to provide information about a type, similarly to the type classes of Haskell. For example, the following method returns the maximum of a list of items.

```
def maxList[T](nums: List[T])  
    (implicit orderer: T=>Ordered[T]): T =  
    nums match {  
        case List() => throw new Error("empty list!")  
        case List(x) => x  
        case x :: rest =>  
            val maxRest = maxList(rest)(orderer)  
            if (orderer(x) > maxRest) x  
            else maxRest  
    }
```

For `maxList` to do its work, it needs to have a way to decide whether one `T` is larger or smaller than another. The `Ordered` trait discussed in [Chapter 11](#) gives such a definition, but how does `maxList` get an instance of `Ordered` for `T`, even if such an instance exists?

In this case the method uses an implicit parameter. The `orderer` parameter in this example is used to describe the ordering of `Ts`. In the body of `maxList`, this ordering is used in two places: a recursive call to `maxList`, and in an `if` expression that checks whether the head of the list is larger than the maximum element of the rest of the list.

This pattern is so common that the standard Scala library already comes with `orderer` methods for many simple types. You can thus use this `maxList` method with a variety of types:

```
scala> maxList(List(1,5,10,3))  
res0: Int = 10  
  
scala> maxList(List(1.5, 5.2, 10.7, 3.14159))  
res1: Double = 10.7
```

In the first case, the compiler inserts an `orderer` function for `Ints`, and in the second case, the compiler inserts one for `Doubles`.

A style rule for implicit parameters . As a style rule, it is best to use a customized named type in the types of implicit parameters. The `maxList` function could just as well have been written with the following type signature:

```
def maxList[T](nums: List[T])
              (implicit orderer: (T,T)=>Boolean): T
```

To use this version of the function, though, the caller would have to supply an `orderer` parameter of type `(T,T)=>Boolean`. This is a fairly generic type that includes any function from two `Ts` to a boolean. It does not indicate anything at all about what the type is for; it could be an equality test, a less-than test, a greater-than test, or something else entirely.

The actual code given above is more stylistic. It uses an `orderer` parameter of type `T=>Ordered[T]`. The word `Ordered` in this type indicates exactly what the implicit parameter is used for: it is for ordering elements of `T`. Because this `orderer` type is more explicit, it becomes no trouble to add implicit conversions for this type in the standard library. To contrast, imagine the chaos that would ensue if you added a method of type `(T,T)=>Boolean` in the standard library, and the compiler started sprinkling it around in people's code. You would end up with code that compiles and runs, but that does fairly arbitrary tests against pairs of items!

Thus the style rule: use at least one role-determining name within the type of an implicit parameter.

19.6 View bounds

The previous example had an opportunity to use an implicit but did not. Note that when you use `implicit` on a parameter, then not only will the compiler try to *supply* that parameter with an implicit value, but the compiler will also *use* that parameter as an available implicit in the body of the method! Thus, both uses of `orderer` within the body of the method can be left out:

```
def maxList2[T](nums: List[T])
```

```
(implicit orderer: T=>Ordered[T]): T =
nums match {
  case Nil => throw new Error("empty list!")
  case x :: Nil => x
  case x :: rest =>
    val maxRest = maxList2(rest) // (orderer) is redundant
    if (x > maxRest) x       // orderer(x) is redundant
    else maxRest
}
```

When the compiler examines the above code, it will see that the types do not match up. For example, `x` of type `T` does not have a `>` method, and so `x > maxRest` does not work. The compiler will not immediately stop, however! It will first look for implicit conversions to repair the code. In this case, it will notice that `orderer` is available, so it can convert the code to `ordered(x) > maxRest`. Likewise for the expression `maxList2(rest)`, which can be converted to `maxList2(rest)(ordered)`. After this the method fully type checks.

Look closely at `maxList2`, now. There is not a single mention of the `ordered` parameter in the text of the method! This coding pattern is actually fairly common. The implicit parameter is used only for conversions, and so it can itself be used implicitly. Now, because the parameter name is never used explicitly, the name could have been anything else. For example, `maxList` would behave identically if you left its body alone but changed the parameter name:

```
def maxList2[T](nums: List[T])
  (implicit converter: T=>Ordered[T]): T =
// same body...
```

For that matter, it could just as well be:

```
def maxList2[T](nums: List[T])
  (implicit icecream: T=>Ordered[T]): T =
// same body...
```

If you want, you can leave out the name of this parameter and shorten the method header by using a *view bound*. Using a view bound, you would write the signature of `maxList` like this:

```
def maxList3[T <% Ordered[T]](nums: List[T]): T =  
    // same body...
```

Mentally, you can think of this code as saying, “I can use any T, so long as it can be treated as an `Ordered[T]`.” This is different from saying that the argument *is* an `Ordered[T]`. It says that the user will supply a conversion so that it can be *treated as* an `Ordered[T]`.

The “treated as” approach is strictly more permissive than “is a”, due to some help from the standard library. The standard library includes the identity function as an available implicit. Thus, if the argument happens to already be an `Ordered[T]`, then the compiler can supply the identity function as the “conversion.” In this case, the conversion is a no-op, which simply returns whatever object it is given.

19.7 Debugging implicits

Implicits are an extremely powerful feature in Scala, but one which is sometimes difficult to get right and debug. So you should use implicits with moderation. Before adding a new implicit conversion, you should ask yourself whether you can achieve the same effect through other means, such as inheritance, mixin composition or method overloading. Only when these fail are implicits warranted.

Sometimes you might wonder why the compiler did not find an implicit conversion which you think should apply. In that case it helps writing the conversion out explicitly. If that also gives an error message, you then know why the compiler could not apply your implicit. For instance, assume that you mistakenly took `stringWrapper` to be a conversion from `Strings` to `Lists`, instead of `RandomAccessSeqs`. So you would wonder why the following scenario does not work:

```
scala> val chars: List[Char] = "xyz"  
<console>:7: error: type mismatch;  
      found   : java.lang.String("xyz")  
      required: List[Char]  
              val chars: List[Char] = "xyz"  
                           ^
```

```
object ImplicitsExample {  
    implicit val favoriteNumber = 4  
    def printSomething(implicit x: Int) = println(x)  
    printSomething  
}
```

Figure 19.1: Sample code that uses an implicit parameter.

```
$ scalac -Xprint:typer ImplicitsExample.scala  
[[syntax trees at end of typer]]  
// Scala source: ImplicitsExample.scala  
package <empty> {  
    final object ImplicitsExample extends AnyRef  
    with ScalaObject  
    def this(): object ImplicitsExample = {  
        ImplicitsExample.super.this();  
        ()  
    };  
    private[this] val favoriteNumber: Int = 4;  
    implicit <stable> <accessor> def favoriteNumber: Int =  
        ImplicitsExample.this.favoriteNumber;  
    def printSomething(implicit x: Int): Unit =  
        scala.this.Predef.println(x);  
    ImplicitsExample.this.printSomething(  
        ImplicitsExample.this.favoriteNumber)  
    }  
}  
$
```

Figure 19.2: The example from Figure 19.1, after type checking and insertion of implicits. The implicit parameter is in bold face.

In that case it helps to write the `stringWrapper` conversion explicitly, to find out what went wrong.

```
<console>:8: error: type mismatch;
  found   : RandomAccessSeq[Char]
  required: List[Char]
        val chars: List[Char] = stringWrapper("xyz")
                                         ^
```

With this, you have found the cause of the error: `stringWrapper` has the wrong return type. On the other hand, it's also possible that inserting the conversion explicitly will make the error go away. In that case you know that some of the other rules (most likely, the Scope Rule) has prevented the implicit from being applied.

When you are debugging a program, it can sometimes help to see what implicit conversions the compiler is inserting. The `-Xprint:typer` option to the compiler is useful for this. If you run `scalac` with this option, then the compiler will show you what your code looks like after all implicit conversions have been added by the type checker. An example is shown in [Figure 19.1](#) and [Fig. 19.2](#).

If you are brave, try `scala -Xprint:typer` to get an interactive shell that prints out the post-typing source code it uses internally. If you do so, be prepared to see an enormous amount of boilerplate surrounding the meat of your code.

Chapter 20

Implementing Lists

Lists have been ubiquitous in this book. Class `List` is probably the most commonly used structured data type in the majority of Scala programs. This chapter “opens up the covers” and explains a bit how lists are implemented in Scala. Knowing the internals of the `List` class is useful for several reasons: You gain a better idea of the relative efficiency of list operations, which will help you in writing fast and compact code using lists. You also learn a toolbox of techniques that you can apply in the design of your own libraries. Finally, the `List` class is a sophisticated application of Scala’s type system in general and its generics concepts in particular. So studying class `List` will deepen your knowledge in these areas.

20.1 The `List` class in principle

Lists are not “built-in” as a language construct in Scala; they are defined by an abstract class `List` in the `scala` package, which comes with two subclasses for `:::` and `Nil`. In the following we present a quick tour through class `List`. This section presents a somewhat simplified account of the class, compared to its real implementation in the Scala standard library, which is covered in [Section 20.3](#).

```
package scala
abstract class List[+T] {
```

`List` is an abstract class, so one cannot define elements by calling the empty `List` constructor. For instance the expression `new List` would be illegal.

The class has a type parameter T. The ‘+’ in front of this type parameter specifies that lists are covariant. Because of this property, you can assign a value of type List[Int], say, to a variable of type List[Any]:

```
scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)

scala> var ys: List[Any] = xs
ys: List[Any] = List(1, 2, 3)
```

All list operations can be defined in terms of three basic methods

```
def isEmpty: Boolean
def head: T
def tail: List[T]
```

These methods are all abstract in class List. They are defined in the subobject Nil and the subclass ‘::’.

The Nil object

The Nil object defines an empty list. Here is its definition:

```
case object Nil extends List[Nothing] {
    override def isEmpty = true
    def head: Nothing =
        throw new NoSuchElementException("head of empty list")
    def tail: List[Nothing] =
        throw new NoSuchElementException("tail of empty list")
}
```

The Nil object inherits from type List[Nothing]. Because of covariance, this means that Nil is compatible with every instance of the List type.

The three abstract methods of class List are implemented in the Nil object in a straightforward way: The isEmpty method returns true and the head and tail methods both throw an exception. Note that throwing an exception is not only reasonable, but practically the only possible thing to do for head: Because Nil is a List of Nothing, the result type of head must be Nothing. Since there is no value of this type, this means that head cannot return a normal value. It has to return abnormally by throwing an exception

(to be precise, the types would also permit for `head` to always go into an infinite loop instead of throwing an exception, but this is clearly not what's wanted).

The ‘`::`’ class

Class ‘`::`’ (pronounced “cons”) represents non-empty lists. It's named that way in order to support pattern matching with the infix ‘`::`’. You have seen in [Section 14.5](#) that every infix operation in a pattern is treated as a constructor application of the infix operator to its arguments. So the pattern `x :: xs` is treated as `::(x, xs)` where ‘`::`’ is a case class. Here is the definition of class ‘`::`’.

```
final case class ::[T](hd: T, tl: List[T]) extends List[T] {  
    def head = hd  
    def tail = tl  
    override def isEmpty: Boolean = false  
}
```

The implementation of the ‘`::`’ class is straightforward. It takes two parameters `hd` and `tl`, representing the head and the tail of the list to be constructed. The definitions of the `head` and `tail` method simply return the corresponding parameter. In fact, this pattern can be abbreviated by letting the parameters directly implement the `head` and `tail` methods of the superclass `List`, as in the following equivalent but shorter definition of class ‘`::`’:

```
final case class ::[T](head: T, tail: List[T]) extends List[T] {  
    override def isEmpty: Boolean = false  
}
```

This works because every case class parameter is implicitly also a field of the class (it's like the parameter declaration was prefixed with `val`). Scala allows you to implement an abstract parameterless method such as `head` or `tail` with a field. So the code above directly uses the parameters `head` and `tail` as implementations of the abstract methods `head` and `tail` that were inherited from class `List`.

Some more methods

All other List methods can be written using the basic three. For instance:

```
def length: Int =  
    if (isEmpty) 0 else 1 + tail.length
```

or

```
def take(n: Int) =  
    if (isEmpty) Nil  
    else if (n == 0) this  
    else tail.take(n-1)
```

or

```
def map[U](f: T => U): List[U] =  
    if (isEmpty) Nil  
    else f(head) :: tail.map(f)
```

List construction

The list construction methods ‘`::`’ and ‘`::::`’ are special. Because they end in a colon, they are bound to their right operand. That is, an operation such as `x :: xs` is treated as the method call `xs.::(x)`, not `x.::(xs)`. In fact, `x.::(xs)` would not make sense, as `x` is of the list element type, which can be arbitrary, so we cannot assume that this type would have a ‘`::`’ method.

For this reason, the ‘`::`’ method should take an element value and should yield a new list. What is the required type of the element value? One might be tempted to say, it should be the same as the list’s element type, but in fact this is more restrictive than necessary. To see why, consider a class hierarchy of arithmetic expressions similar to the one defined in [Chapter 12](#) on pattern matching.

```
scala> trait Expr  
defined trait Expr  
  
scala> case class Number(n: Int) extends Expr  
defined class Number
```

```
scala> case class Var(s: String) extends Expr
defined class Var

scala> val exprs1 = Number(1) :: Nil
exprs1: List[Number] = List(Number(1))

scala> val exprs2 = Var("x") :: exprs1
exprs2: List[Expr] = List(Var(x), Number(1))
```

The `exprs1` value is treated as a `List` of `Numbers`, as expected. However, the definition of `exprs2` shows that it's still possible to add an element of a different type to that list. The element type of the resulting list is `Expr`, which is the most precise common supertype of the original list element type (*i.e.* `Number`) and the type of the element to be added (*i.e.* `Var`).

This flexibility is obtained by defining the “cons” method ‘`::`’ as follows.

```
def ::[U >: T](x: U): List[U] = new scala.::(x, this)
```

Note that the method is itself polymorphic—it takes a type parameter named `U`. Furthermore, `U` is constrained in `[U >: T]` to be a supertype of the list element type `T`. The element to be added is required to be of type `U` and the result is a `List[U]`.

You can check how with this formulation of ‘`::`’ the above definition of `exprs2` works out type-wise: In that definition the type parameter `U` of ‘`::`’ is instantiated to `Expr`. The lower-bound constraint of `U` is satisfied, because the list `exprs1` has type `List[Number]` and `Expr` is a supertype of `Number`. The argument to the ‘`::`’ is `Var("x")`, which conforms to type `Expr`. Therefore, the method application is type-correct with result type `List[Expr]`.

In fact, the polymorphic definition of ‘`::`’ with the lower bound `T` is not only convenient; it is also necessary to render the definition of class `List` type-correct. This is because `Lists` are defined to be covariant. Assume for a moment that we had defined ‘`::`’ like this:

```
def ::(x: T): List[T] = new scala.::(x, this)
```

You have seen in [Chapter 17](#) that method parameters count as contravariant positions, so the list element type `T` is in contravariant position in the definition above. But then `List` cannot be declared covariant in `T`. The lower

bound $[U >: T]$ kills, therefore, two birds with one stone: It removes a typing problem and leads to a ‘`:::`’ method that’s more flexible to use.

The list concatenation method ‘`:::`’ is defined in a similar way to ‘`::`’:

```
def :::[U >: T](prefix: List[U]): List[U] =
  if (prefix.isEmpty) this
  else prefix.head :: prefix.tail :::: this
```

Like `cons`, concatenation is polymorphic. The type of the result is “widened” as necessary to include the types all list elements. Note also that again the order of the arguments is swapped between an infix operation and an explicit method call. Because both ‘`:::`’ and ‘`::`’ end in a colon, they both bind to the right and are both right associative. For instance, the `else` part of the definition of ‘`:::`’ above contains infix operations of both ‘`::`’ and ‘`::::`’. These infix operations are expanded to equivalent method calls as follows:

```
prefix.head :: prefix.tail :::: this
= prefix.head :: (prefix.tail :::: this)
= (prefix.tail :::: this).:::(prefix.head)
= this.::::(prefix.tail).:::(prefix.head)
```

20.2 The ListBuffer class

The typical access pattern for a list is recursive. For instance, to increment every element of a list without using `map` we could write:

```
def incAll(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case x :: xs1 => x + 1 :: incAll(xs1)
}
```

One shortcoming of this program pattern is that it is not tail-recursive. Note that the recursive call to `incAll` above occurs inside a ‘`::`’-operation. Therefore each recursive call requires a new stack-frame. On today’s virtual machines this means that you cannot apply `incAll` to lists of much more than about 30000 to 50000 elements. This is a pity.

How do you write a version of `incAll` that can work on lists of arbitrary size (as much as heap-capacity allows)? The obvious approach is to use a loop:

```
for (x <- xs) ??
```

But what should go in the loop body? Note that where `incAll` above constructs the list by prepending elements to the result of the recursive call, the loop needs to append new elements at the end of the result list. One, very inefficient possibility is to use the list append operator ‘`:::`’.

```
var result = List[Int]()
for (x <- xs) result = result :: List(x + 1)
result
```

This has terrible efficiency, though. Because ‘`:::`’ takes time proportional to the length of its first operand, the whole operation takes time proportional to the square of the length of the list. This is clearly unacceptable.

A better alternative is to use a list buffer. List buffers let you accumulate the elements of a list. To do this, you use an operation such as `buf += elem` which appends the element `elem` at the end of the list buffer `buf`. Once you are done appending elements, you can turn the buffer into a list using the `toList` operation.

`ListBuffer` is a class in package `scala.collection.mutable`. To use the simple name only, you can import that package:

```
import scala.collection.mutable.ListBuffer
```

Using a list buffer, the body of `incAll` can now be written as follows:

```
val buf = new ListBuffer[Int]
for (x <- xs) buf += x + 1
buf.toList
```

This is a very efficient way to build lists. In fact, the list buffer implementation is organized so that both the append operation ‘`+=`’ and the `toList` operation take (very short) constant time.

20.3 The List class in practice

The implementations of list methods given in [Section 20.1](#) are concise and clear, but suffer from the same stack overflow problem as the non-tail recursive implementation of `incAll`. Therefore, most methods in the real implementation of class `List` avoid recursion and use loops with list buffers instead. For example, here is the real implementation of `map` in class `List`:

```
final override def map[U](f: T => U): List[U] = {  
    val b = new ListBuffer[U]  
    var these = this  
    while (!these.isEmpty) {  
        b += f(these.head)  
        these = these.tail  
    }  
    b.toList  
}
```

This revised implementation traverses the list with a simple loop, which is highly efficient. A tail recursive implementation would be similarly efficient, but a general recursive implementation would be slower and less scalable. But what about the operation `b.toList` at the end? What is its complexity? In fact, the call to the `toList` method takes only a small number of cycles, which is independent of the length of the list.

To understand why, take a second look at class `::` which constructs non-empty lists. In practice, this class does not quite correspond to its idealized definition given previously in [Section 20.1](#). There's one peculiarity:

```
final case class ::[U](hd: U,  
                      private[scala] var tl: List[U])  
extends List[U] {  
    def head = hd  
    def tail = tl  
    override def isEmpty: Boolean = false  
}
```

It turns out that the second “`tl`” argument is a variable! This means that it is possible to modify the tail of a list after the list is constructed. However, because the variable `tl` has the modifier `private[scala]`, it can be

accessed only from within package `scala`. Client code outside this package can neither read nor write `t1`.

Since the `ListBuffer` class is contained in package `scala.collection.mutable`, it can access the `t1` field of a cons-cell. In fact the elements of a list buffer are represented as a list and appending new elements involves a modification of `t1` field of the last ‘`::`’ cell in that list. Here’s the start of class `ListBuffer`:

```
package scala.collection.immutable
final class ListBuffer[T] extends Buffer[T] {
    private var start: List[T] = Nil
    private var last0: ::[T] = _
    private var exported: Boolean = false
    ...
}
```

You see three private fields that characterize a `ListBuffer`.

`start` points to the list of all elements stored in the buffer.

`last0` points to the last ‘`::`’-cell in that list.

`exported` indicates whether the buffer has been turned into a list using a `toList` operation.

The `toList` operation is very simple:

```
override def toList: List[T] = {
    exported = !start.isEmpty
    start
}
```

It returns the list of elements referred to by `start` and also sets `exported` to true if that list is nonempty. So `toList` is very efficient, because it does not copy the list which is stored in a listbuffer. But what happens if the list is further extended after the `toList` operation? Of course, once a list is returned from `toList`, it must be immutable. However, appending to the `last0` element will modify the list which is referred to by `start`.

To maintain the correctness of the list buffer operations, one needs to work on a fresh list instead. This is achieved by the first line in the implementation of the ‘`+=`’ operation:

```
override def += (x: T) {  
    if (exported) copy()  
    if (start.isEmpty) {  
        last0 = new scala.::(x, Nil)  
        start = last0  
    } else {  
        val last1 = last0  
        last0 = new scala.::(x, Nil)  
        last1.tl = last0  
    }  
}
```

You see that ‘`+=`’ copies the list pointed to by `start` if `exported` is true. So, in the end, there is no free lunch. If you want to go from lists which can be extended at the end to immutable lists, there needs to be some copying. However, the implementation of `ListBuffer` is such that copying is necessary only for list buffers that are further extended after they have been turned into lists. This case is quite rare in practice. Most use cases of list buffers construct elements incrementally which is followed by one `toList` operation at the end. In such cases, no copying is necessary.

20.4 Conclusion

This section has shown key elements of the implementation of Scala’s `List` and `ListBuffer` classes. You have seen that lists are purely functional “at the outside” but have an imperative implementation using list buffers “inside”. This is a typical strategy in Scala programming: trying to combine purity with efficiency by carefully delimiting the effects of impure operations. You might ask, why insist on purity? Why not just open up the definition of lists, making the `tail` field, and maybe also the `head` field, mutable? The disadvantage of such an approach is that it would make programs much more fragile. Note that constructing lists with ‘`::`’ re-uses the tail of the constructed list. So when you write

```
val ys = 1 :: xs  
val zs = 2 :: xs
```

the tails of lists `ys` and `zs` are shared; they point to the same data structure. This is essential for efficiency; if the list `xs` was copied everytime you added a new element onto it, this would be much slower. Because sharing is pervasive, changing list elements, if it were possible, would be quite dangerous. For instance, taking the code above, if you wanted to truncate list `ys` to its first two elements by writing

```
val ys.drop(2).tail = Nil // can't do this in Scala!
```

you would also truncate lists `zs` and `xs` as a side-effect. Clearly, it would be quite difficult to keep track of what gets changed. That's why Scala goes for pervasive sharing and no mutation for lists. The `ListBuffer` class still allows you to build up lists imperatively and incrementally, if you wish to. But since list buffers are not lists, the types keep mutable buffers and immutable lists separate.

Scala's `List`/`ListBuffer` design is quite similar to what's done in Java's pair of classes `String` and `StringBuffer`. This is no coincidence. In both situations the designers wanted to maintain a pure immutable data structure but also wanted to provide an efficient way to construct this structure incrementally. For Java and Scala strings, `StringBuffers` (or, in Java 5, `StringBuilder`s) provide a way to construct a string incrementally. For Scala's lists, you have a choice: You can either construct lists incrementally by adding elements to the beginning of a list using `:::`, or you use a list buffer for adding elements to the end. Which one is preferable depends on the situation. Usually, `:::` lends itself well to recursive algorithms in the divide-and-conquer style. List buffers are often used in a more traditional loop-based style.

Chapter 21

Object Equality

Comparing two values for equality is ubiquitous in programming. It is also more tricky than it looks at first glance. This chapter studies object equality in detail and gives some recommendations to consider when you design your own equality tests.

The definition of equality is different in Scala and Java. Java knows two equality methods: ‘`==`’, which is the natural equality for value types and object identity for reference types, and `equals` which is (user-defined) canonical equality for reference types. This convention is problematic, because the more natural symbol ‘`==`’ does not always correspond to the natural notion of equality. When programming in Java, one of the most frequently encountered pitfalls is to compare objects with ‘`==`’ when they should have been compared with `equals`. For instance, comparing two strings `x` and `y` using `(x == y)` might well yield `false` in Java, even if `x` and `y` are the same, meaning they consist of exactly the same characters in the same order.

Scala also has an equality like ‘`==`’ in Java, but it is written `eq`. `(x eq y)` is true if `x` and `y` reference the same object, or if `x` and `y` are the same primitive value. The ‘`==`’ equality is reserved in Scala for the “natural” equality of each type. For value types, ‘`==`’ is the same as `eq`. For reference types, ‘`==`’ is the same as `equals`. You can redefine the behavior of ‘`==`’ for new types by overriding the `equals` method, which is always inherited from class `Any`. The inherited `equals`, which takes effect unless overridden, is object identity, as is the case in Java. So `equals` (and with it, ‘`==`’) is by default the same as `eq`, but you can change its behavior by overriding the `equals` method in the classes you define. It is not possible to override ‘`==`’ directly,

as it is defined as a final method in class Any. That is, Scala treats ‘`==`’ as if was defined as follows in class Any:

```
final def ==(other: Any): Boolean = this.equals(other)
```

21.1 Writing an equality method

How should the `equals` method be defined? It turns out that writing a correct equality method is surprisingly difficult in object-oriented languages. Mardana Vaziri and Frank Tip have recently done a study of a large body of Java code, and concluded that almost all implementations of `equals` methods are faulty.

This is problematic, because equality is at the basis of many other things. For one, a faulty equality method for a type C might mean that you cannot reliably put an object of type C in a collection. You might have two elements `elem1`, `elem2` of type C which are equal, *i.e.* `(elem1 equals elem2)` yields `true`. Nevertheless, with commonly occurring faulty implementations of the `equals` method you could still see behavior like the following:

```
val set = new collection.immutable.HashSet
set += elem1
set get elem2    // returns None!
```

Here are four common pitfalls that can cause inconsistent behavior of `equals`.

1. Defining `equals` with the wrong signature.
2. Changing `equals` without also changing `hashCode`.
3. Defining `equals` in terms of mutable fields
4. Failing to define `equals` as an equivalence relation.

These four pitfalls are discussed in the following.

Pitfall #1: Defining equals with the wrong signature.

Consider adding an equality method to the following class of simple points:

```
class Point(val x: Int, val y: Int) { ... }
```

A seemingly obvious, but wrong way would be to define it like this:

```
/** An utterly wrong definition of equals */
def equals(other: Point): Boolean =
    this.x == other.x && this.y == other.y
```

What's wrong with this method? At first glance, it seems to work OK:

```
scala> val p1, p2 = new Point(1, 2)
p1: Point = Point@109558d
p2: Point = Point@1cfcc277

scala> val q = new Point(2, 3)
q: Point = Point@107f8ba

scala> p1 equals p2
res0: Boolean = true

scala> p1 equals q
res1: Boolean = false
```

However, trouble starts once you start putting points into a collection:

```
scala> import scala.collection.mutable.~
import scala.collection.mutable.~

scala> val coll = HashSet(p1)
coll: scala.collection.mutable.HashSet[Point] = HashSet(p1)

scala> coll contains p2
res2: Boolean = false
```

How to explain that `coll` does not contain `p2`, even though `p1` was added to it, and `p1` and `p2` are equal objects? The reason becomes clear in the following interaction, where the precise type of one of the compared points is masked. Define `p2a` as an alias of `p2`, but with type `Any` instead of `Point`:

```
scala> val p2a: Any = p2
p2a: Any = Point@1cfc277
```

Now, repeating the first comparison but with the alias p2a instead of p2 you get:

```
scala> p1 equals p2a
res7: Boolean = false
```

What went wrong? In fact, the version of `equals` given above does not override the standard method `equals`, because its type is different. Here is the type of the `equals` method as it is defined in the root class `Any`¹:

```
def equals(other: Any): Boolean
```

Because the `equals` method in `Point` takes a `Point` instead of an `Any` as an argument, it does not override `equals` in `Any`. Instead, it is just an overloaded alternative. Now, overloading in Scala and in Java is resolved by the static type of the argument, not the run-time type. So as long as the static type of the argument is `Point`, the `equals` method in `Point` is called. However, once the static argument is of type `Any`, it's the `equals` method in `Any` which is called instead. This method has not been overridden, so it is still object identity. That's why the comparison (`p1 equals p2a`) yields `false` even though points `p1` and `p2a` have the same coordinates. That's also why the `contains` method in `HashSet` returned `false`. Since that method operates on generic sets, it calls the generic `equals` method in `Object` instead of the overloaded variant in `Point`.

A better `equals` method is the following:

```
/** A better definition, but still not perfect */
override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y == that.y
    case _ => false
}
```

Now `equals` has the correct type – it takes a value of type `Any` as parameter and it yields a Boolean result. The implementation of this method uses a pattern match. It first tests whether the `other` object is also of type `Point`.

¹ to do: explain that `Object` is mapped to `Any` for `equals` imported from Java

If it is, it compares the coordinates of the two points. Otherwise the result is false.

A related pitfall is to define ‘==’ with a wrong signature. Normally, if you try to redefine == with the correct signature, which takes an argument of type Any, the compiler will give you an error because you try to override a final member of type Any. However, newcomers to Scala sometimes make two errors at once: They try to override ‘==’ *and* they give it the wrong signature. For instance:

```
def ==(other: Point): Boolean = // don't do this!
```

In that case, the user-defined ‘==’ method is treated as an overloaded variant of the same-named method class Any, and the program compiles. However, the behavior of the program would be just as dubious as if you had defined equals with the wrong signature.

Pitfall #2: Changing equals without also changing hashCode

If you repeat the comparison of p1 and p2a with the latest definition of Point defined previously, you will get true, as expected. However, if you repeat the HashSet.contains test, you will probably still get false

```
val p1, p2 = new Point(1, 2)
p1: Point = Point@3f4a21
p2: Point = Point@11bda67

HashSet(p1) contains p2
res2: Boolean = false
```

In fact, this outcome is not 100% certain – you might also get true from the experiment. If you do, you can try with some other points with coordinates 1 and 2. Eventually, you’ll get one which is not contained in the set. What goes wrong here is that Point redefined equals without also redefining hashCode.

Note that the collection in the example above is a HashSet. This means elements of the collection are put in “hash-buckets” determined by their hash code. The contains test first determines a hash bucket to look in and then compares the given elements with all elements in that bucket. Now, the last version of class of Point did redefine equals, but it did not at the same

time redefine hashCode. So hashCode is still what it was in its version in class Object: some function of the address of the allocated object. The hash codes of p1 and p2 are almost certainly different, even though the fields of both points are the same. Different hash codes mean with high probability different hash buckets in the set. The contains test will look for a matching element in the bucket which corresponds to p2's hash code. In most cases, point p1 will be in another bucket, so it will never be found. p1 and p2 might also end up by chance in the same hash bucket. In that case the test would return true.

The problem was that the last implementation of Point violated the contract on hashCode as stated in the JavaDoc documentation of class java.lang.Object: *If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.*

In fact, it's well known in Java that hashCode and equals should always be redefined together. Furthermore, hashCode may only depend on fields that equals depends on. For the Point class, the following would be a suitable definition of hashCode:

```
class Point(val x: Int, val y: Int) {  
    override def hashCode = x * 41 + y  
    override def equals(other: Any) = ... // as before  
}
```

This is just one of many possible implementations of hashCode. Multiplying one field with a prime number such as 41 and adding the other field to the result gives a reasonable distribution of hash codes at a low cost in running time and code size.

Adding hashCode fixes the problems of equality when defining classes like Point. However, there are still other troublespots to watch out for.

Pitfall #3: Defining equals in terms of mutable fields

Consider the following slight variation of class Point:

```
class Point(var x: Int, var y: Int) {  
    override def hashCode = ... // as before  
    override def equals(other: Any) = ... // as before
```

```
}
```

The only difference is that the fields `x` and `y` are now mutable. The `equals` and `hashCode` methods are now defined in terms of these mutable fields, so their results change when the fields change. This can have strange effects once you put points in collections:

```
val p1 = new Point(1, 2)
p1: Point = Point@2b

val coll = HashSet(p1)
coll: scala.collection.mutable.Set[Point] = HashSet(Point@2b)

coll contains p1
res5: Boolean = true
```

Now, if you change a field in point `p1`, does the collection still contain the point? Let's try:

```
scala> p1.x += 1
scala> coll contains p1
res6: Boolean = false
```

This looks strange. Where did `p1` go? More strangeness results if you check whether the `elements` iterator of the set contains `p1`:

```
coll.elements contains p1
res7: Boolean = true
```

So here's a set which does not contain `p1`, yet `p1` is among the elements of the set! What happened, of course, is that after the change to the `x` field, the point `p1` ended up in the wrong hash bucket of the set `coll`. That is, its original hash bucket no longer corresponded to the new value of its hash code. In a manner of speaking, the point `p1` “dropped out of sight” in the set `coll` even though it still belonged to its elements.

The lesson to be drawn from this example is that `equals` and `hashCode` should never be defined in terms of mutable fields. If you need a comparison that takes the current state of an object into account, you should name it something else, not `equals`. Considering the last definition of `Points`, it would have been preferable to omit a redefinition of `hashCode` and to name

the comparison method `equalContents`, or some other name different from `equals`. `Point` would then have inherited the default implementation of `equals` and `hashCode`. So `p1` would have stayed locatable in `coll` even after the modification to its `x` field.

Pitfall #4: Failing to define `equals` as an equivalence relation

The contract for `equals` found on the JavaDoc page for class `java.lang.Object` specifies that it must be an equivalence relation. It reads as follows:

The `equals` method implements an equivalence relation on non-null object references.

- *It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return true.*
- *It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.*
- *It is transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.*
- *It is consistent: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in `equals` comparisons on the objects is modified.*
- *For any non-null reference value `x`, `x.equals(null)` should return false.*

Checking the previous definition of `equals` in class `Point`, one finds that this method satisfies the contract for `equals`. However, things become more complicated once subclasses are considered. Say there is a subclass `ColoredPoint` of `Point` which adds a field `color` of an enumeration type `Color.Value`.

```
object Color extends Enumeration {
    val red, green, blue = Value
}
```

Class ColoredPoint re-implements equals to also take the new color field into account:

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
  extends Point(x, y) {
    override def equals(other: Any) = other match {
        case that: ColoredPoint =>
            this.color == that.color && super.equals(that)
        case _ => false
    }
}
```

This is what most experienced programmers would write. Note that ColoredPoint need not override hashCode. Because the new definition of equals on ColoredPoint is stricter than the overridden definition in Point (meaning it equates fewer pairs of objects), the contract for hashCode stays valid. If two colored points are equal, they must have the same coordinates, so their hash codes are guaranteed to be equal as well.

Taking the class ColoredPoint by itself, its definition of equals looks OK. However, the contract for equals is broken once points and colored points are mixed. Consider:

```
scala> val p = new Point(1, 2)
p: Point = Point@2b

scala> val cp = new ColoredPoint(1, 2, Color.red)
cp: ColoredPoint = ColoredPoint@dab0d4

scala> p equals cp
res20: Boolean = true

scala> cp equals p
res21: Boolean = false
```

The comparison (p equals cp) invokes p's equals method, which is defined in class Point. This method only takes into account the coordinates of the two points. Consequently, the comparison yields true. On the other

hand, the comparison (`cp equals p`) invokes `cp`'s `equals` method, which is defined in class `ColoredPoint`. This method returns `false`, because `p` is not a `ColoredPoint`. So the relation defined by `equals` is not symmetric.

The loss in symmetry can have unexpected consequences for collections. Here's an example:

```
scala> HashSet[Point](p) contains cp
res38: Boolean = true

scala> HashSet[Point](cp) contains p
res39: Boolean = false
```

So even though `p` and `cp` are equal, one `contains` test succeeds whereas the other one fails.

How can you change the definition of `equals` so that it becomes symmetric? Essentially there are two ways. You can either make the relation more general or stricter. Making it more general means that a pair of two objects `x` and `y` is taken to be equal if either comparing `x` with `y` or comparing `y` with `x` yields `true`. Here's code that does this:

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
  extends Point(x, y) {
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (this.color == that.color) && super.equals(that)
    case that: Point =>
      that.equals(this)
    case _ =>
      false
  }
}
```

The new definition of `equals` in `ColoredPoint` has one more case than the old one: If the other object is a `Point` but not a `ColoredPoint`, the method forwards to the `equals` method of `Point`. This has the desired effect of making `equals` symmetric. Now, both (`cp equals p`) and (`p equals cp`) give `true`. However, the contract for `equals` is still broken. Now the problem is that the new relation is no longer transitive! Here's a sequence of statements

which demonstrates this. Define a point and two colored points of different colors, all at the same position:

```
scala> val redp = new ColoredPoint(1, 2, Color.red)
redp: ColoredPoint = ColoredPoint@2b
scala> val bluep = new ColoredPoint(1, 2, Color.blue)
bluep: ColoredPoint = ColoredPoint@2b
```

Taken individually, `redp` is equal to `p` and `p` is equal to `bluep`:

```
scala> redp == p
res40: Boolean = true
scala> p == bluep
res41: Boolean = true
```

However, comparing `redp` and `bluep` yields `false`:

```
scala> redp == bluep
res42: Boolean = false
```

Hence, there is a violation of the transitivity clause of `equal`'s contract.

Making the `equals` relation more general seems to lead into a dead end. Let's try to make it stricter instead. One way to make `equals` stricter is to always treat objects of different classes as different. That could be achieved by modifying the `equals` methods in classes `Point` and `ColoredPoint` as follows.

```
class Point(val x: Int, val y: Int) {
    override def hashCode = x * 41 + y
    override def equals(other: Any) = other match {
        case that: Point =>
            this.x == that.x && this.y == that.y &&
            this.getClass == that.getClass
        case _ => false
    }
}
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) {
    override def equals(other: Any) = other match {
```

```
        case that: ColoredPoint =>
          (this.color == that.color) && super.equals(that) &&
          this.getClass == that.getClass
        case _ =>
          false
      }
}
```

The new definitions satisfy symmetry and transitivity because now every comparison between objects of different classes yields false. So a colored point can never be equal to a point. This convention looks reasonable, but one could argue that the new definition is too strict.

Consider the following slightly roundabout way to define a point at coordinates (1, 2):

```
val p1a = new Point(1, 1) { override val y = 2 }
p1: Point = anon0@2b
```

Is p1a equal to p1? The answer is no because the class objects associated with p1 and p1a are different. For p1 it is Point whereas for p1a it is an anonymous subclass of Point. But clearly, p1a is just another point at coordinates (1, 2). It does not seem reasonable to treat it as being different from p1.

So it seems we are stuck. Is there a sane way to redefine equality on several levels of the class hierarchy while keeping its contract? In fact, there is such a way, but it requires one more method to redefine together with equals and hashCode. The idea is that as soon as a class redefines equals (and hashCode), it should also explicitly state that objects of this class are never equal to objects of some superclass which implement a different equality method. This is achieved by adding a method isComparable to every class which redefines equals. Here's the method's signature:

```
def isComparable(other: Any): Boolean
```

The method should yield true if the other object is an instance of the class in which isComparable is (re-)defined, false otherwise. It is called from equals to make sure that the objects are comparable both ways. Here's a new (and last) implementation of class Point along these lines:

```
class Point(val x: Int, val y: Int) {  
    override def hashCode = x * 41 + y  
    override def equals(other: Any) = other match {  
        case that: Point =>  
            (this.x == that.x) && (this.y == that.y) &&  
            (that.isComparable this)  
        case _ =>  
            false  
    }  
    def isComparable(other: Any) = other.isInstanceOf[Point]  
}
```

The `equals` test in class `Point` contains a third condition: that the other object is comparable to this one. The implementation `isComparable` in `Point` states that all instances of `Point` are comparable. Here is the corresponding implementation of class `ColoredPoint`:

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)  
    extends Point(x, y) {  
    override def hashCode = super.hashCode * 41 + color.hashCode  
    override def equals(other: Any) = other match {  
        case that: ColoredPoint =>  
            super.equals(that) && this.color == that.color  
        case _ =>  
            false  
    }  
    override def isComparable(other: Any) = other.isInstanceOf[ColoredPoint]  
}
```

It can be shown that the new definition of `Point` and `ColoredPoint` keeps the contract of `equals`. Equality is symmetric and transitive. Comparing a `Point` to a `ColoredPoint` always yields `false`. Indeed, for all points `p` and colored point `cp` it's the case that (`p.equals(cp)`) returns `false` because `cp.isComparable(p)` is `false`. The reverse comparison (`cp.equals(p)`) also returns `false`, because `p` is not a `ColoredPoint`, so the first pattern match in the body of `equals` in `ColoredPoint` fails.

On the other hand, instances of different subclasses of `Point` can be equal, as long as none of the classes redefines the `equals` method. For

instance, with the new class definitions, the comparison of `p1` and `p1a` would yield true.

Here's a summary of all the things to consider when redefining the `equals` method:

1. Every class redefining `equals` also needs to define `isComparable`. If the inherited definition of `equals` is from `Object` (that is, `equals` was not redefined higher up in the class hierarchy), the definition of `isComparable` is new, otherwise it overrides a previous definition of a method with the same name.
2. `isComparable` always yields true if the argument object is an instance of the current class (*i.e.* the class in which `isComparable` is defined), false otherwise.
3. The `equals` method of the class that first introduced `isComparable` also contains a test of the form (that `isComparable this`) where `this` is the argument of the equality method.
4. Overriding redefinitions of `equals` also add this test, unless they contain a call to `super.equals`. In the latter case, the `isComparable` test is already done by the superclass call.

If you keep to these rules, equality is guaranteed to be an equivalence relation, as is required by `equal`'s contract.

In retrospect, defining a correct implementation of `equals` has been surprisingly subtle. You might prefer to define your classes of comparable objects as case classes, because then the Scala compiler will add an `equals` method with the right properties automatically.

Defining equality for parameterized types

This section will appear in a future version of the PrePrint PDF.

Chapter 22

Working with XML

This chapter introduces Scala’s support for XML. After discussing semi-structured data in general, it shows the essential functionality in Scala for manipulating XML: how to make nodes with XML literals, how to save and load XML to files, and how to take apart XML nodes by using query methods and by using pattern matching. This chapter is just a brief introduction to what is possible with XML, but it shows enough to get you started.

22.1 Semi-structured data

XML is a form of *semi-structured data*. It is more structured than plain strings, because it organizes the contents of the data into a tree. With XML, you can always take two fragments and combine them as part of a new node. Later you can take those two apart and reliably get the same two nodes again. With strings there are no such operations, so you must design delimiters and escapes yourself.

Semi-structured data is very helpful any time you need to serialize program data for saving in a file or shipping across a network. It provides enough structure that it is easier to parse and to produce than plain strings. Additionally, it is more flexible than raw object data, because if the data model changes you can always just make the parser a little bit more complex.

There are many semi-structured data formats, but XML is important because it is widely used and thus there are many helpful tools already. Practically every language your program might communicate with has a parser

and generator for XML. Additionally, there are a host of separate tools for XML, facilities that other formats do not enjoy simply because they are less popular.

For all of these reasons, Scala includes special support for processing XML. This chapter shows you Scala’s support for: constructing XML, analyzing XML with methods, and analyzing XML via pattern matching. Along the way, the chapter shows several common coding patterns for using XML in Scala.

22.2 Creating XML

You can type an XML node anywhere that an expression is valid. Simply type an open tag and then continue writing XML content. When the compiler sees the last close tag, it will go back to reading arbitrary Scala code.

```
scala> <a>
|   This is some XML.
|   Here is a tag: <atag/>
| </a>
res0: scala.xml.Elem =
<a>
    This is some XML.
    Here is a tag: <atag></atag>
</a>
```

The result of this expression is of type `Elem`, meaning it is an XML node that has a label (“`a`”) and children (“This is some XML,” *etc.*). Some other important XML classes are:

- Class `Node` is the abstract superclass of all XML node classes.
- Class `Text` is a node holding just text. The “stuff” part of `<a>stuff` is of class `Text`.
- Class `NodeSeq` holds a sequence of nodes. Many methods in the XML library process `NodeSeq`’s in places you might expect them to process individual `Node`’s. You can still use such methods with individual nodes, however, since `Node` extends from `NodeSeq`. This may sound

weird, but it works out well for XML. You can think of an individual Node as a one-element NodeSeq.

You can also use {} if you want to insert XML that is computed at run-time. Inside the {} you can put arbitrary Scala code:

```
scala> <a> {2+2} </a>
res1: scala.xml.Elem = <a> 4 </a>
```

You can even nest XML literals further inside a {} escape, thus allowing your code to switch back and forth between XML as the nesting level increases.

```
scala> val yearMade = 1955
yearMade: Int = 1955
scala> <a> { if (year < 2000) <old>{year}</old>
|           else xml.NodeSeq.Empty }
|   </a>
res2: scala.xml.Elem =
<a> <old>1955</old>
      </a>
```

If the code inside the {} evaluates to either an XML node or a sequence of XML nodes, then those nodes are inserted directly as is. In the above example, if year is less than 2000, then it is wrapped in <old> tags and added to the <a> element. If year is newer than 2000, then nothing is added. To denote “nothing” as an XML node, use `xml.NodeSeq.Empty`. The empty XML sequence makes no sense as a top-level XML document, but it is frequently useful when optionally inserting something to the middle of a tree of XML.

A {} escape is not required to evaluate to an XML node. It can evaluate to any Scala value. In such a case, result is converted to a string and inserted as a text node.

```
scala> <a> {3+4} </a>
res3: scala.xml.Elem = <a> 7 </a>
```

Any <, >, and & characters in the text will be escaped if you print the node back out.

```
scala> <a> {"</a>potential security hole<a>"} </a>
res4: scala.xml.Elem = <a> &lt;/a&gt;potential security
hole&lt;a&gt; </a>
```

XML literals plus {} escapes make it easy to write conversions from internal data structures to XML. For example, suppose you are implementing a database to keep track of your extensive collection of vintage Coca-Cola thermometers. You might make the following class to hold one entry in the catalog:

```
abstract class CCTherm {
    val description: String
    val yearMade: Int
    val dateObtained: String
    val bookPrice: Int // in pennies
    val purchasePrice: Int // in pennies
    val condition: Int // 1-10
    override def toString = description
}
```

Converting this class to XML is easy. Simply write down an XML literal and use code escapes to insert the data that is particular to each instance. Here is a `toXML` method that does the trick:

```
abstract class CCTherm {
    ...
    def toXML =
        <cctherm>
            <description>{description}</description>
            <yearMade>{yearMade}</yearMade>
            <dateObtained>{dateObtained}</dateObtained>
            <bookPrice>{bookPrice}</bookPrice>
            <purchasePrice>{purchasePrice}</purchasePrice>
            <condition>{condition}</condition>
        </cctherm>
}
```

Here is the method in action:

```
scala> val therm = new CCTherm {  
|   val description = "hot dog thermometer"  
|   val yearMade = 1952  
|   val dateObtained = "March 14, 2006"  
|   val bookPrice = 2199  
|   val purchasePrice = 500    // sucker!  
|   val condition = 9  
| }  
therm: CCTherm = hot dog thermometer  
scala> therm.toXML  
res5: scala.xml.Elem =  
<cctherm>  
          <description>hot dog thermometer</description>  
          <yearMade>1952</yearMade>  
          <dateObtained>March 14, 2006</dateObtained>  
          <bookPrice>2199</bookPrice>  
          <purchasePrice>500</purchasePrice>  
          <condition>9</condition>  
</cctherm>
```

By the way, if you want to include a “{” or “}” as XML text, as opposed to using them to escape to Scala code, simply write two of them in a row:

```
scala> <a> }}}}brace yourself!{{{ </a>  
res6: scala.xml.Elem = <a> }}brace yourself!{{ </a>
```

22.3 Taking XML apart

Among the many methods available for the XML classes, there are three of them that you should particularly be aware of. They allow you to take apart XML without thinking too much about the precise way XML is represented in Scala.

Extracting text. Send `text` to any XML node to retrieve all of the text within that node, minus any element tags.

```
scala> <a>blah blah <tag/> blah</a>.text
res7: String = blah blah  blah
```

Any encoded characters are decoded automatically.

```
scala> <a> input ---&gt; output </a>.text
res8: String =  input ---> output
```

Extracting sub-elements. If you want to find a sub-element by tag name, simply call ‘\’ with the name of the tag.

```
scala> <a><b><c>hello</c></b></a> \ "b"
res9: scala.xml.NodeSeq = <b><c>hello</c></b>
```

You can do a “deep search”, and look through sub-sub-elements, *etc.*, by using \\ instead of \:

```
scala> <a><b><c>hello</c></b></a> \ "c"
res10: scala.xml.NodeSeq =
scala> <a><b><c>hello</c></b></a> \\ "c"
res11: scala.xml.NodeSeq = <c>hello</c>
```

Extracting attributes. You can extract tag attributes using the same \ and \\ methods. Simply put an “at” sign before the attribute name.

```
scala> val joe = <employee
|   name="Joe"
|   rank="code monkey"
|   serial="123"/>
joe: scala.xml.Elem = <employee rank="code monkey"
name="Joe" serial="123"></employee>
scala> joe \ "@name"
res12: scala.xml.NodeSeq = Joe
scala> joe \ "@serial"
res13: scala.xml.NodeSeq = 123
```

Using these methods, you can easily parse XML back into a CCTerm with the following code:

```
def fromXML(node: xml.Node): CCTherm =  
  new CCTherm {  
    val description = (node \ "description").text  
    val yearMade =  
      Integer.parseInt((node \ "yearMade").text)  
    val dateObtained = (node \ "dateObtained").text  
    val bookPrice =  
      Integer.parseInt((node \ "bookPrice").text)  
    val purchasePrice =  
      Integer.parseInt((node \ "purchasePrice").text)  
    val condition =  
      Integer.parseInt((node \ "condition").text)  
  }
```

Here is this method in action:

```
scala> val node = therm.toXML  
node: scala.xml.Elem =  
<cctherm>  
          <description>hot dog thermometer</description>  
          <yearMade>1952</yearMade>  
          <dateObtained>March 14, 2006</dateObtained>  
          <bookPrice>2199</bookPrice>  
          <purchasePrice>500</purchasePrice>  
          <condition>9</condition>  
        </cctherm>  
  
scala> fromXML(node)  
res14: CCTherm = hot dog thermometer
```

22.4 Loading and saving

Converting XML to and from in-memory data structures is only half of the conversion needed to use XML as an interchange format. You also need to convert the XML to a sequence of bytes that can be saved to a file or sent

over the network. This second conversion is automatic, so all you need to think about is the precise way you invoke the converters.

To convert XML to a string, you can simply use `toString()`. That is why printing XML in the Scala shell works effectively. However, it is better to convert directly from XML to bytes when you can, because then the resulting bytes can include a description of the character encoding. If you convert to a string, and then serialize the string, there is a chance that the software that later loads the XML will use the wrong character encoding.

To convert from XML to a file of bytes, you can use the `XML.saveFull` command. The last item describes the “document type” of this XML node. You can specify `null` to leave the document type unspecified.

```
xml.XML.saveFull("therm1.xml", node, "UTF-8", true, null)
```

After running the above command, the resulting file `therm1.xml` looks like the following:

```
<?xml version='1.0' encoding='UTF-8'?>
<cctherm>
    <description>hot dog thermometer</description>
    <yearMade>1952</yearMade>
    <dateObtained>March 14, 2006</dateObtained>
    <bookPrice>2199</bookPrice>
    <purchasePrice>500</purchasePrice>
    <condition>9</condition>
</cctherm>
```

Loading is simpler, and can be accomplished with the command `XML.load`.

```
scala> val loadnode = xml.XML.loadFile("therm1.xml")
loadnode: scala.xml.Elem =
<cctherm>
    <description>hot dog thermometer</description>
    <yearMade>1952</yearMade>
    <dateObtained>March 14, 2006</dateObtained>
    <bookPrice>2199</bookPrice>
    <purchasePrice>500</purchasePrice>
    <condition>9</condition>
</cctherm>
```

```
scala> fromXML(loadnode)
res14: CCTherm = hot dog thermometer
```

Those are the basic methods you need. There are many variations on these loading and saving methods, including methods for reading and writing to various kinds of readers, writers, input and output streams.

22.5 Pattern matching

So far you have seen how to dissect XML using the XPath-like `text`, `\`, and `\\"` methods. These are good when you know exactly what kind of XML structure you are taking apart. Sometimes, though, there are a few possible structures the XML could have. Maybe there are multiple kinds of records within the data, for example because you have extended your thermometer collection to include clocks and sandwich plates. Maybe you simply want to skip over any white space between tags. Whatever the reason, you can use the pattern matcher to sift through the possibilities.

An XML pattern looks just like an XML literal. The main difference is that if you insert a `{}` escape, then the code inside the `{}` is not an expression but a pattern! A pattern embedded in `{}` can use the full Scala pattern language, including binding new variables, performing type tests, and ignoring content using the `_` and `_*` patterns.

Here is a simple example to give the idea.

```
scala> <a>blahblah</a> match {
    |   case <a>{contents}</a>  => "yes! " + contents
    |   case _ => "no! "
    | }
res15: java.lang.String = yes! blahblah
```

In the “yes” case, the pattern checks for an `<a>` tag with a single element. It then binds that element to a new variable named `cont`. This code is probably not exactly what you would want, however, because it checks that there is precisely one element within the `<a>`. If there are multiple elements—or if there are zero elements!—then the “yes” case fails and the “no” branch executes:

```
scala> <a></a> match {
```

```
|   case <a>{contents}</a> => "yes! " + contents
|   case _ => "no! "
| }
res16: java.lang.String = no!
```

You probably want to match on a sequence of items, not a single item, but if you write a variable as above it will only match on a single item. To fix this, there's the “any sequence” pattern, which matches any number of arguments. You can use this pattern to match a sequence of nodes and then bind the result to the pattern variable `conts`:

```
scala> <a></a> match {
|   case <a>{contents @ _*}</a> => "yes! " + contents
|   case _ => "no! "
| }
res17: java.lang.String = yes! Array()
```

As a tip, be aware that XML patterns work very nicely with `for` expressions as a way to iterate through some parts of an XML tree while ignoring other parts. For example, suppose you wish to skip over the white space between records in the following XML structure:

```
val catalog =
<catalog>
  <cctherm>
    <description>hot dog thermometer</description>
    <yearMade>1952</yearMade>
    <dateObtained>March 14, 2006</dateObtained>
    <bookPrice>2199</bookPrice>
    <purchasePrice>500</purchasePrice>
    <condition>9</condition>
  </cctherm>
  <cctherm>
    <description>Sprite Boy thermometer</description>
    <yearMade>1964</yearMade>
    <dateObtained>April 28, 2003</dateObtained>
    <bookPrice>1695</bookPrice>
    <purchasePrice>595</purchasePrice>
```

```
<condition>5</condition>
</cctherm>
</catalog>
```

Visually, it looks like there are two sub-elements of the `<catalog>` element. Actually, though, there are five. There is white space before, after, and between the two elements! If you do not consider this white space, you might incorrectly process the thermometer records as follows:

```
scala> catalog match {
|   case <catalog>{therms @ _*}</catalog> =>
|     for (therm <- therms)
|       println("processing: " +
|             (therm \ "description").text)
|   }
processing:
processing: hot dog thermometer
processing:
processing: Sprite Boy thermometer
processing:
```

Notice all of the lines that try to process white space as if it were a true thermometer record. What you would really like to do is ignore the white space, and process only those sub-elements that are inside a `<cctherm>{_*}</cctherm>`, and you can restrict the for expression to iterating over items that match that pattern:

```
scala> catalog match {
|   case <catalog>{therms @ _*}</catalog> =>
|     for (therm @ <cctherm>{_*}</cctherm> <- therms)
|       println("processing: " +
|             (therm \ "description").text)
|   }
processing: hot dog thermometer
processing: Sprite Boy thermometer
```

22.6 Conclusion

This chapter has only scratched the surface of what you can do with XML. There are a multitude of tools that work with XML, some customized for Scala, some for Java, and others not dependent on any specific programming language.

What you should walk away with is how to use semi-structured data for interchange purposes, and how to use Scala's support for XML as semi-structured data. With Scala, you can create XML using XML literals, take it apart using three simple methods, sift through it using pattern matching, and save and load it using fully automatic routines.

Chapter 23

Actors and Concurrency

23.1 Overview

Sometimes it helps in designing a program to specify that certain things happen independently, in parallel. Java includes support for these notions, most notably its threads and locks. This support is sufficient, but it turns out to have problems in practice as programs get larger and more complex.

Scala augments Java's native support by adding *actors*. Actors avoid a lot of the problems with threads and locks by providing a safe message-passing system. If you can design your program in an actors style, then you will avoid the deadlocks and race conditions of Java's native concurrency support. This chapter shows you how.

23.2 Locks considered harmful

Java provides threads and locks and monitors, so you might ask, why not use Java's support directly?

As a Scala programmer, you are certainly free to use Java's concurrency constructs. Java provides you independent threads, locks, and monitors. Your strategy would then be to hold a lock (or enter a monitor) whenever you access shared data. It sounds simple.

In practice it is impractically tricky for larger programs. At each program point, you must reason about which locks are currently held. At each method call, you must reason about which locks it will try to hold, and convince yourself that it does not overlap the set of locks already held. Compounding

the problem, the locks you reason about are not simply a finite list in the program, because the program is free to create new locks at run time as it progresses.

Making things worse, testing is not reliable with locks. Since threads are non-deterministic, you might successfully test a program one thousand times, yet still the program could go wrong the first time it runs on a customer's machine. With locks, you must get the program correct through reason alone.

Over-engineering also does not solve the problem. Just as you cannot use no locks, you cannot put a lock around every operation, either. The problem is that new lock operations remove possibilities for race conditions, but simultaneously add possibilities for deadlocks. A correct lock-using program must have neither of these.

Overall, there is no good way to fix locks and make them practical to use. That is why Scala provides an alternative concurrency approach, one based on message-passing actors.

23.3 Actors and message passing

An actor is a kind of thread that has a mailbox for receiving messages. To implement an actor, subclass `scala.actors.Actor` and implement an `act()` method.

```
import scala.actors._  
object sillyActor extends Actor {  
    def act() {  
        for (i <- 1 to 5) {  
            println("I'm acting!")  
            Thread.sleep(1000)  
        }  
    }  
}
```

An actor can then be started with the `start()` method, just as if it were a normal Java thread:

```
scala> sillyActor.start()
```

```
I'm acting!  
res4: scala.actors.Actor = sillyActor@1945696  
  
scala> I'm acting!  
I'm acting!  
I'm acting!  
I'm acting!
```

Notice that the “I’m acting!” output is interleaved with the Scala shell’s output (“res4:”, *etc.*). This interleaving is due to the `sillyActor` actor running independently from the thread running the shell. Actors run independently from each other, too. For example, here are two actors running at the same time.

```
object seriousActor extends Actor {  
    def act() {  
        for (i <- 1 to 5) {  
            println("To be or not to be.")  
            Thread.sleep(1000)  
        }  
    }  
}  
  
scala> { sillyActor.start(); seriousActor.start() }  
res3: scala.actors.Actor = seriousActor@1689405  
  
scala> To be or not to be.  
I'm acting!  
To be or not to be.  
I'm acting!
```

It’s also possible to create an actor directly, using a method called `actor` in object `scala.actors.Actor`:

```
scala> import scala.actors.Actor._  
scala> val seriousActor2 = actor {  
|   for (i <- 1 to 5)  
|     println("That is the question.")  
|     Thread.sleep(1000)  
| }  
  
scala> That is the question.  
That is the question.  
That is the question.  
That is the question.  
That is the question.
```

The value definition above creates an actor which executes the actions defined in the block following the `actor` method. The actor starts immediately when it is defined. There is no need to call a separate `start` method.

All well and good. More interesting, though, is when actors send each other messages. A message is sent by using the `!` method, like this:

```
scala> sillyActor ! "hi there"
```

Nothing happens in this case, because `sillyActor` is too busy acting to read from its mailbox. Here is a new actor that waits for a message in its mailbox and prints out whatever it receives. It receives a message by calling `receive` and giving it a pattern-match expression.¹

```
val echoActor = actor {  
  while (true) {  
    receive {  
      case msg =>  
        println("received message: " + msg)  
    }  
  }  
}
```

¹ More precisely, any partial function will do. In practice people usually make the partial function using a pattern-match expression.

```
scala> echoActor ! "hi there"
received message: hi there
scala> echoActor ! 15
scala> received message: 15
```

23.4 Treating native threads as actors

The actor subsystem manages one or more native threads for its own use. So long as you work with an explicit actor that you define, you do not need to think much about how they map to threads.

The other direction is also supported by the subsystem: Every native thread is also usable as an actor. However, you cannot use `Thread.current` directly, because it does not have the necessary methods. Instead, you should access `Actor.self` if you want to view the current thread as an actor.

This facility is especially useful for debugging actors from the interactive shell.

```
scala> import scala.actors._
import scala.actors._

scala> Actor.self ! "hello"

scala> Actor.self.receive { case x => x }
res1: Any = hello

scala>
```

If you use this technique, it is better to use a timeout of 0 so that your shell does not block forever.

```
scala> Actor.self.receiveWithin(0) { case x => x }
res2: Any = TIMEOUT

scala>
```

23.5 Tips for better actors

At this point you have seen everything you need to write your own actors. Consider a few tips, though on how to make those programs shorter and easier to write.

Sharing threads with event-based actors

Actors are implemented on top of normal Java threads. As described so far, in fact, every actor must be given its own thread, so that all the `act()` methods get their turn.

Unfortunately, threads are not cheap in Java. Threads use enough memory that typical Java virtual machines can have millions of objects but only a couple of thousands of threads. Worse, switching threads often takes hundreds if not thousands of processor cycles. If you want your program be as efficient as possible, then it is important to be sparing with thread creation and thread switching.

To help you with this task, Scala provides an alternative to the usual `receive` method called `react`. Unlike `receive`, `react` does not return immediately to the caller. Instead, it evaluates the message handler and then terminates the actor. Thus, the message handler you pass to `react` must not only process that message, but arrange to do all of the actor's remaining work!

This remaining work can often be accomplished by arranging to have a top-level `work` method that the message handler calls when it finishes. Do not worry about the apparent infinite recursion; the actors system clears an actor's stack every time it calls `react`. Here is an example of using this approach.

```
object NameResolver extends Actor {  
    import java.net.{InetAddress, UnknownHostException}  
  
    def act() {  
        react {  
            case (name: String, actor: Actor) =>  
                actor ! getip(name)  
                act()  
            case msg =>
```

```
    println("Unhandled message: " + msg)
    act()
}
}

def getip(name: String): Option[InetAddress] = {
  try {
    Some(InetAddress.getByName(name))
  } catch {
    case _:UnknownHostException => None
  }
}
```

This actor waits for strings that are host names, and it returns an IP address for that host name if there is one. Here is an example session using it:

```
scala> NameResolver.start()
res3: scala.actors.Actor = NameResolver$@dfcc0e
scala> NameResolver ! ("www.scala-lang.org", Actor.self)
scala> Actor.self.receiveWithin(5000) { case x => x }
res5: Any = Some(www.scala-lang.org/128.178.154.102)
scala> NameResolver ! ("wwwwwww.scala-lang.org", Actor.self)
scala> Actor.self.receiveWithin(5000) { case x => x }
res7: Any = None
```

Writing an actor to use `react` instead of `receive` is challenging, but pays off in performance. Because `react` does not return, the calling actor's call stack can be discarded, freeing up the thread's resources for a different actor. At the extreme, if all of the actors of a program use `react`, then they can be implemented on a single thread.

This coding pattern is so common with event-based actors, there is special support in the library for it. The `Actor.loop` function executes a block of code repeatedly, even if the code calls `react`. Using `loop`, the above code can be written like this:

```
def act() {
```

```
loop {  
    react {  
        case (name: String, actor: Actor) => actor ! getip(name)  
        case msg => println("Unhandled message: " + msg)  
    }  
}  
}
```

Messages should not block

A well-written actor does not block while processing a message, not even for one second. The problem is that while the actor blocks, some other actor might make a request on it that it could handle. If the actor is blocked on the first request, it will not even notice the second request. In the worst case, a deadlock can even result, with every actor blocked as it waits on some other actor to respond.

Instead of blocking, the actor should arrange for some message to arrive designating that action is ready to be taken. Often this rearrangement will require the help of other actors. For example, here is an actor to help with timing:

```
case class SendAfter(time: Int, to: Actor, msg: Any)  
  
val timerActor = actor {  
    loop {  
        react {  
            case SendAfter(time, to, msg) =>  
                actor {  
                    Thread.sleep(time)  
                    to ! msg  
                }  
        }  
    }  
}
```

This actor processes timing requests from other actors. For each request, it starts a helper actor to process that one request and then terminate. The

helper actor does indeed block, but since it will never receive a message, it is okay in this case. The main `timerActor`, notice, remains available to answer new requests.

You then use the timer actor like this.

```
val sillyActor2 = actor {  
    var emoted = 0  
    timerActor ! SendAfter(1000, Actor.self, "Emote")  
  
    while (emoted < 5) {  
        receive {  
            case "Emote" =>  
                println("I'm acting!")  
                emoted += 1  
                timerActor ! SendAfter(1000, Actor.self, "Emote")  
            }  
        }  
    }  
}
```

Use immutable data

To use actors effectively, you need to arrange that every object is accessed by only one actor at a time. Since locks are so difficult to use in practice, this means you should arrange your program so that each object is owned by only one actor. You can arrange for objects to be transferred from one actor to another if you like, but you need to make sure that at any given time, it is clear which actor owns the object and is allowed to access it. Whenever you design an actors-based system, you need to decide which parts of memory are assigned to which actor.

There is an exception, though: immutable data does not play by this rule! Any data that is immutable can be safely accessed by multiple actors. Since the data does not change, there do not need to be any locks, and you do not need to assign the data to a particular object.

Immutable data is convenient in many cases, but it really shines for parallel systems. When you design a program that might involve parallelism in the future, you should try especially hard to make data structures immutable. All of the value types are immutable, as are strings. Most case classes are,

too. Obviously, the “immutable” collection types live up to their name. Lists, as well as the default Scala map and set classes, are all immutable.

On the other hand, arrays are mutable, and thus any array you use must only be accessed by one actor at a time. Similarly, any object that has a var in it is mutable.

However, as you have seen in [Chapter 16](#), even an object which does not have vars in it can be “stateful”, because it might reference a mutable object which has vars. Stateful objects also should not be shared between actors, because two different actors can follow the same references and arrive at the same mutable object at the same time. Verifying that an object is stateless means that you have to consider the object itself, plus every object it references, and so on. To help in this effort, many people note in the class comments if they think that instances of a class are stateless.

All of the examples in this chapter pass only immutable data between the actors. Thus, they avoid the need to assign data to actors. In larger programs you will not always be able to make all data immutable. In such cases, you will need to assign an actor to each data structure. All other actors that access a mutable object must send messages to the object’s owner and wait for a message to come back with a reply.

Make messages self-contained

When you return a value from a method, the caller is in a good position to remember what it was doing before it called the method. It can take the response value and then continue whatever it was doing.

With actors, things are trickier. When one actor makes a request of another, the response might come not come for a long time. The calling actor should not block, but should continue to do any other work it can while it waits for the response. A difficulty, then, is interpreting the response when it finally comes. Can the actor remember what it was doing when it made the request?

One way to simplify the logic of an actors program is to include redundant information in the messages. If the request is an immutable object, you can even cheaply include a reference to the request in the return value! For example, the IP-lookup actor would be better if it returned the host name in addition to the IP address found for it. It would make this actor slightly longer, but it should simplify the logic of any actor making requests on it.

```
def act() {
    Actor.loop {
        react {
            case (name: String, actor: Actor) =>
                actor ! (name, getip(name))
        }
    }
}
```

Another way to increase redundancy in the messages is to make a case class for each kind of message. While such a wrapper is not strictly necessary in many cases, it makes an actors program much easier to understand. Imagine a programmer looking at a send of a string, for example:

```
lookerUpper ! ("www.scala-lang.org", this)
```

It can be difficult to figure out which actors in the code might respond. It is much easier if the code looks like this:

```
case class LookupIP(hostname: String, requester: Actor)
lookerUpper ! LookupIP("www.scala-lang.org", this)
```

Now, the programmer can search for `LookupIP` in the source code, probably finding very few possible responders. Here is the full code for an IP-lookup actor that uses case classes for the messages:

```
import java.net.InetAddress
case class LookupIP(name: String, respondto: Actor)
case class LookupResult(name: String, address: Option[InetAddress])

val nameResolver2 = actor {
    loop {
        react {
            case LookupIP(name, actor) =>
                actor ! LookupResult(name, getip(name))
        }
    }
}
```

Chapter 24

Extractors

By now you have probably grown accustomed to the concise way data can be decomposed and analyzed using pattern matching. This chapter shows you how to generalize this concept further. Until now, constructor patterns were linked to case classes. For instance, `Some(x)` is a valid pattern because `x` is a case class. Sometimes you might wish that you can write patterns like this without creating an associated case class. In fact, you might wish to be able to create your own kinds of patterns. Extractors give you a way to do this.

This chapter explains what extractors are and how you can use them to define patterns that are decoupled from an object's representation.

24.1 An Example

Say you want to analyze strings that represent e-mail addresses. Given a string, you want to decide whether it is an e-mail address or not, and, if it is one, you want to access the user and domain parts of the address. The traditional way to do this uses three helper functions:

```
def isEmail(s: String): Boolean  
def domain(s: String): String  
def user(s: String): String
```

With these functions, you could parse a given string `s` as follows:

```
if (isEmail(s)) println(user(s)+" AT "+domain(s))  
else println("not an e-mail address")
```

This works, but it is kind of clumsy. What's more, things would become more complicated when you combine several such tests. For instance you might want to find two successive strings in a list that are both e-mail addresses with the same user. You can try this yourself with the access functions defined previously to see what would be involved.

You have seen already in [Chapter 12](#) that pattern matching is ideal for attacking problems like this. Let's assume for the moment that you could match a string with a pattern

```
EMail(user, domain)
```

The pattern would match if the string contained an embedded '@'-sign. In that case it would bind variable `user` to the part of the string before the '@' and variable `domain` to the part after it. Postulating a pattern like this, the previous expression could be written more clearly like this:

```
s match {
    case EMail(user, domain) => println(user+" AT "+domain)
    case _ => println("not an e-mail address")
}
```

The more complicated problem of finding two successive e-mail addresses with the same user part would translate to the following pattern:

```
ss match {
    case EMail(u1, d1) :: Email(u2, d2) :: _ if (u1 == u2) => ...
    ...
}
```

This is much more legible than anything that could be written with access functions. However, the problem with all this is that strings are not case classes; they do not have a representation that conforms to `EMail(user, domain)`. This is where Scala's extractors come in: They let you define new patterns for pre-existing types, where the pattern need not follow the internal representation of the type.

24.2 Extractors

An extractor in Scala is an object that has a method called `unapply` as one of its members. The purpose of that `unapply` method is to match a value and take it apart. Often, the extractor object also defines a dual method `apply` for building values, but this is not required. As an example, here is an extractor object for e-mail addresses:

```
/** An extractor object */
object EMail {

    /** The injection method (optional) */
    def apply(user: String, domain: String) = user+"@"+domain

    /** The extraction method (mandatory) */
    def unapply(email: String): Option[(String, String)] = {
        val parts = email split "@"
        if (parts.length == 2) Some(parts(0), parts(1)) else None
    }
}
```

This object defines both `apply` and `unapply` methods. The `apply` method has the same meaning as always: It turns `EMail` into a function object. So you can write `EMail("John", "epfl.ch")` to construct the string `"John@epfl.ch@"`. To make this more explicit, you could also let `EMail` inherit from Scala's function type, like this:

```
object EMail extends (String, String) => String { ... }
```

The `unapply` method is what turns `EMail` into an extractor. In a sense, it reverses the construction process of `apply`. Where `apply` takes two strings and forms an e-mail address string out of them, `unapply` takes an e-mail address and returns potentially two strings: the user and the domain of the address. But `unapply` must also handle the case where the given string is not an e-mail address. That's why `unapply` returns an `Option`-type over pairs of strings. Its result is either `Some(user, domain)` if the string `s` is an e-mail address with the given user and domain parts, or `None`, if `s` is not an e-mail address. Here are some examples:

```
unapply("John@epfl.ch") = Some("John", "epfl.ch")
unapply("John Doe") = None
```

Now, whenever pattern matching encounters a pattern referring to a extractor object, it invokes the extractor's unapply method on the selector expression. For instance, executing the code

```
s match { case EMail(user, domain) => ... }
```

would lead to the call

```
EMail.unapply(s)
```

As you have seen previously, this call returns either None or Some(u, d), for some values u for the user part of the address and d for the domain part. In the first case, the pattern does not match, and the system tries another pattern or fails with a MatchError exception. In the second case, if Some(u, d) is returned from the unapply, the pattern matches and its variables are bound to the fields of the returned value. In the previous match, user would be bound to u and domain would be bound to d.

In the EMail pattern matching example, the type String of the selector expression s conformed to unapply's argument type (which in the example was also String). This is quite common, but not necessary. It would also be possible to use the Email extractor to match selector expressions for more general types. For instance, to find out whether an arbitrary value x was a e-mail address string, you could write:

```
val x: Any = ...
obj match { case EMail(user, domain) => ... }
```

Given this code, the pattern matcher will first check whether the given value x conforms to String, the parameter type of Email's unapply method. If it does conform, the value is cast to String and pattern matching proceeds as before. If it does not conform, the pattern fails immediately.

In object EMail, the apply method is called an *injection*, because it takes some arguments and yields an element of a given set (in our case: the set of strings that are e-mail addresses). The unapply method is called an *extraction*, because it takes an element of the same set and extracts some of its parts (in our case: the user and domain substrings). Injections and extractions are often grouped together in one object, because then one can use the object's name for both a constructor and a pattern, which simulates the convention for pattern matching with case classes. However, it is also possible to define

an extraction in an object without a corresponding injection. The object itself is called an *extractor*, independently of the fact whether it has an `apply` method or not.

If an injection method is included, it should be the dual to the extraction method. For instance, a call of

```
EMail.unapply(EMail.apply(user, domain))
```

should return

```
Some(user, domain)
```

i.e. the same sequence of arguments wrapped in a `Some`. Going in the other direction means running first the `unapply` and then the `apply`, as shown in the following code:

```
EMail.unapply(obj) match {
    case Some(u, d) => EMail.apply(u, d)
}
```

In that code, if the match on `obj` succeeds, you'd expect to get back that same object from the `apply`. These two conditions for the duality of `apply` and `unapply` are good design principles. They are not enforced by Scala, but it's recommended to keep to them when designing your extractors.

24.3 Patterns with zero or one variables

In `unapply` method of the previous example returned a pair of element values in the success case. This is easily generalized to patterns of more than two variables. To bind N variables, an `unapply` would return a N -element tuple, wrapped in a `Some`.

The case where a pattern binds just one variable is treated differently, however. There is no one-tuple in Scala. So to return just one pattern element, the `unapply` method simply wraps the element itself in a `Some`.

Here is an example: The following extractor object defines `apply` and `unapply` methods for strings that consist of two times the same substring in a row.

```
object Twice {  
    def apply(s: String) = s + s  
    def unapply(s: String) = {  
        val l = s.length / 2  
        val half = s.substring(0, l)  
        if (half == s.substring(l)) Some(half) else None  
    }  
}
```

It's also possible that an extractor pattern does not bind any variables. In that case the corresponding `unapply` method returns a boolean—true for success and false for failure. For instance, the following extractor object characterizes strings consisting of all uppercase characters.

```
object UpperCase {  
    def unapply(s: String) = s.toUpperCase == s  
}
```

This time, the extractor only defines an `unapply`, but not an `apply`. It would make no sense to define an `apply`, as there's nothing to construct.

The following `test` method applies all previously defined extractors together in its pattern matching code:

```
def test2(s: String) = s match {  
    case EMail(Twice(x @ UpperCase()), domain) =>  
        "match: "+x+" in domain "+domain  
    case _ =>  
        "no match"  
}
```

The first pattern of this method matches strings that are e-mail addresses consisting of two times the same string in uppercase letters. For instance:

```
scala> test2("DIDI@hotmail.com")  
res2: java.lang.String = match: DI in domain hotmail.com  
  
scala> test2("DIDO@hotmail.com")  
res3: java.lang.String = no match  
  
scala> test2("didi@hotmail.com")  
res4: java.lang.String = no match
```

Note that `UpperCase` in method `test2` takes an empty parameter list `()`. This cannot be omitted as otherwise the match would test for equality with the object `UpperCase!` Note also that, even though `UpperCase()` itself does not bind any variables, it is still possible to associate a variable with the whole pattern matched by it. To do this, you use the standard scheme of variable binding explained in [Section 12.2](#): The form `x @UpperCase()` associates the variable `x` with the pattern matched by `UpperCase()`.

24.4 Variable argument extractors

The previous extraction methods for e-mail addresses all returned a fixed number of element values. Sometimes, this is not flexible enough. For example, you might want to match on a string representing a domain name, so that every part of the domain is kept in a different sub-pattern. This would let you express patterns such as the following:

```
dom match {
    case Domain("org", "acm") => println("acm.org")
    case Domain("com", "sun", "java") => println("java.sun.com")
    case Domain("net", _) => println("a .net domain")
}
```

In this example things were arranged so that domains are expanded in reverse order—from the top-level domain down to the sub-domains. This was done so that one can better profit from sequence patterns. You have seen in [Section 12.2](#) that a sequence wildcard pattern `_*` at the end of an argument list matches any remaining fields in a sequence. This feature is more useful if the top-level domain comes first, because then one can use sequence wildcards to match sub-domains of arbitrary depth.

But the question remains how an extractor can support *vararg matching* like in the previous example, where patterns can have a varying number of sub-patterns. The `unapply` methods encountered so far are not sufficient, because they each return a fixed number of sub-elements in the success case. Instead, Scala lets you define a different extraction method specifically for vararg matching. This method is called `unapplySeq`. To see how it is written, have a look at the `Domain` extractor

```
object Domain {
```

```
def apply(parts: String*): String =
    parts.reverse.mkString(".")
def unapplySeq(whole: String): Option[Seq[String]] =
    Some(whole.split("\\.").reverse)
}
```

The `Domain` object defines an `unapplySeq` method that first splits the string into parts separated by periods. This is done using Java's `split` method on strings, which takes a regular expression as its second argument. The result of `split` is an array of substrings. The result of `unapplySeq` is then that array with all elements reversed and wrapped in a `Some`.

The most general result type `unapplySeq` needs to have is `Option[Seq[T]]`, where the element type `T` is arbitrary. Here, `Seq` is a class in Scala's collection hierarchy. It's a common superclass of several classes describing different kinds of sequences: `Lists`, `Arrays`, and several others. All methods that are common to lists and arrays are inherited from class `Seq`.

For symmetry, `Domain` also has an `apply` method that builds a domain string from a variable argument parameter of domain parts starting with the top-level domain. As always, the `apply` method is optional.

You can use the `Domain` extractor to get more detailed information out of e-mail strings. For instance, to search for an e-mail address named "tom" in some `.com` domain, you could write the following test method:

```
scala> def isTomInDotCom(s: String) = s match {
|   case EMail("tom", Domain("com", _*)) => true
|   case _ => false
| }
```

isTomInDotCom: (String)Boolean

This gives the expected results:

```
scala> isTomInDotCom("tom@sun.com")
res3: Boolean = true

scala> isTomInDotCom("peter@sun.com")
res4: Boolean = false

scala> isTomInDotCom("tom@acm.org")
res5: Boolean = false
```

It's also possible to return some fixed elements from an `unapplySeq` together with the variable part. This is expressed by returning all elements in a tuple, where the variable part comes last, as usual. As an example, here is a new extractor for e-mails where the domain part is already expanded into a sequence:

```
object ExpandedEMail {  
    def unapplySeq(email: String)  
        : Option[(String, Seq[String])] = {  
        val parts = email split "@"  
        if (parts.length == 2)  
            Some(parts(0), parts(1).split("\\\\.").reverse)  
        else  
            None  
    }  
}
```

The `unapplySeq` method in `ExpandedEMail` returns an optional value of a pair. The first element of the pair is the user part. The second element is a sequence of names representing the domain. You can match on this as usual:

```
scala> val s = "tom@support.epfl.ch"  
s: java.lang.String = tom@support.epfl.ch  
  
scala> val ExpandedEMail(name, topdomain, subdomains @ _) = s  
name: String = tom  
topdomain: String = ch  
subdomains: Seq[String] = List(epfl, support)
```

24.5 Extractors and sequence patterns

You have seen in [Chapter 8](#) that you can access the elements of a list or an array using sequence patterns such as

```
List()  
List(x, y, _*)  
Array(x, 0, 0, _)
```

In fact, these sequence patterns are all implemented using extractors in the standard Scala library. For instance, patterns of the form `List(...)` are possible because the `scala.List` object is an extractor that defines an `unapplySeq` method. Here are the relevant definitions:

```
package scala
object List {
    def apply[T](elems: T*) = elems.toList
    def unapplySeq[T](x: List[T]): Option[Seq[T]] = Some(x)
    ...
}
```

The `List` object contains an `apply` method which takes a variable number of arguments. That's what lets you write expressions such as

```
List()
List(1, 2, 3)
```

It also contains an `unapplySeq` method that returns all elements of the list as a sequence. That's what supports `List(...)` patterns. Very similar definitions exist in the object `scala.Array`. These support analogous injections and extractions for arrays.

24.6 Extractors vs Case Classes

Even though they are very useful, case classes have one shortcoming: they expose the concrete representation of data. This means that the name of the class in a constructor pattern corresponds to the concrete representation type of the selector object. If a match against

```
case C(...)
```

succeeds, you know that the selector expression is an instance of class `C`.

Extractors break this link between data representations and patterns. You have seen in the examples in this section that they enable patterns that have nothing to do with the data type of the object that's selected on. This property is called *representation independence*. In open systems of large size, representation independence is very important because it allows you to change an

implementation type used in a set of components without affecting clients of these components.

If your component had defined and exported a set of case classes, you'd be stuck with them, because client code could already contain pattern matches against these case classes. Renaming some case classes or changing the class hierarchy would affect client code. Extractors do not share this problem, because they represent a layer of indirection between a data representation and the way it is viewed by clients. You could still change a concrete representations of a type, as long as you update all your extractors with it.

Representation independence is an important advantage of extractors over case classes. On the other hand, case classes also have some advantages of their own over extractors. First, they are much easier to set up and to define, and they require less code. Second, they usually lead to more efficient pattern matches than extractors, because the Scala compiler can optimize patterns over case classes much better than patterns over extractors. This is because the mechanisms of case classes are fixed, whereas an unapply or unapplySeq method in an extractor could do almost anything. Third, if your case classes inherit from a sealed base class, the Scala compiler will check your pattern matches for exhaustiveness and will complain if some combination of possible values is not covered by a pattern. No such exhaustiveness checks are available for extractors.

So which of the two methods should you prefer for your pattern matches? It depends. If you write code for a closed application, case classes are usually preferable because of their advantages in conciseness, speed and static checking. If you decide to change your class hierarchy later, the application needs to be refactored, but this is usually not a problem. On the other hand, if you need to expose a type to unknown clients, extractors might be preferable because they maintain representation independence.

Fortunately, you need not decide right away. You could always start with case classes and then, if the need arises, change to extractors. Because patterns over extractors and patterns over case classes look exactly the same in Scala, pattern matches in your clients will continue to work.

Of course, there are also situations where it's clear from the start that the structure of your patterns does not match the representation type of your data. The e-mail addresses discussed in this chapter were one such example. In that case, extractors are the only possible choice.

24.7 Conclusion

In this chapter you have seen out how to generalize pattern matching with extractors. Extractors let you define your own kinds of patterns, which need not correspond to the type of the expressions you select on. This gives you more flexibility for the kinds of patterns you want to use for matching. In effect it's like having different possible views on the same data. It also gives you a layer between a type's representation and the way clients view it. This lets you do pattern matching while maintaining representation independence, a property which is very useful in large software systems.

Chapter 25

Objects As Modules

You saw in [Chapter 13](#) how to divide programs into packages and thus get more modular code. While this kind of division is already quite helpful, it is limited because it provides no way to abstract. You cannot reconfigure a package two different ways within the same program, and you cannot inherit between packages. A package always includes one precise list of contents, and that list is fixed until you change the code.

A more powerful approach is to make modules out of plain old objects. In Scala, there is no need for objects to be “small” things, no need to use some other kind of construct for “big” things like modules. One of the ways Scala is a *scalable* language is that the same constructs are used for structures both small and large.

This chapter walks through using objects as modules, starting with a basic example and then showing how to take advantage of various Scala features to improve on them.

25.1 A basic database

We'll start by building a persistent database of recipes. The database will be in one module, and a database browser will be in another. The database will hold all of the recipes that a person has collected. The browser will help search and browse that database, for example to find every recipe that includes an ingredient you have on hand.

The first thing to do is to model foods and recipes. To keep things simple, a food will simply have a name, and a recipe will simply have a name, a list

Figure 25.1: A simple Food class and some example foods.

```
abstract class Food(val name: String) {  
    override def toString() = name  
}  
object Apple extends Food("Apple")  
object Orange extends Food("Orange")  
object Cream extends Food("Cream")  
object Sugar extends Food("Sugar")
```

Figure 25.2: A simple Recipe class and an example recipe.

```
abstract class Recipe(val name: String,  
                     val ingredients: List[Food],  
                     val instructions: String) {  
    override def toString() = name  
}  
object FruitSalad extends Recipe(  
    "fruit salad",  
    List(Apple, Orange, Cream, Sugar),  
    "Stir it all together.")
```

of ingredients, and some instructions. The necessary classes are shown in [Figure 25.1](#) and [Figure 25.2](#).

Scala uses objects for modules, so start by making two singleton objects as follows. For now, the database module is backed by a simple in-memory list.

```
object SimpleDatabase {  
    def allFoods = List(Apple, Orange, Cream, Sugar)  
  
    def foodNamed(name: String) =  
        allFoods.find(f => f.name == name)  
  
    def allRecipes: List[Recipe] = List(FruitSalad)  
}  
  
object SimpleBrowser {  
    def recipesUsing(food: Food) =  
        SimpleDatabase.allRecipes.filter(recipe =>  
            recipe.ingredients.contains(food))  
}
```

You can use this database as follows:

```
scala> val apple = SimpleDatabase.foodNamed("Apple")  
apple: Option[Food] = Some(Apple)  
  
scala> SimpleBrowser.recipesUsing(apple.get)  
res0: List[Recipe] = List(fruit salad)
```

To make things a little more interesting, suppose the database sorts foods into categories. To implement this, add a `FoodCategory` class and a list of all categories in the database.

```
object SimpleDatabase {  
    ...  
    case class FoodCategory(name: String, foods: List[Food])  
  
    private var categories = List(  
        FoodCategory("fruits", List(Apple, Orange)),  
        FoodCategory("misc", List(Cream, Sugar)))
```

```
def allCategories = categories
}

object SimpleBrowser {
    ...
    def displayCategory(category: SimpleDatabase.FoodCategory) {
        // show info about the specified category
    }
}
```

Notice in this last example that the `private` keyword, so useful for implementing classes, is also useful for implementing modules. Items marked `private` are part of the implementation of a module, and thus are particularly easy to change without affecting other modules.

At this point, many more facilities could be added, but you get the idea. Programs can be divided into singleton objects, which we can think of as modules. This viewpoint is not very useful by itself, but it becomes very useful when you consider abstraction.

25.2 Abstraction

Suppose you want the same code base to support multiple recipe databases, and you want to be able to create a separate browser for each of these databases. You would like to reuse the browser code for each of the instances, because the only thing different about the browsers is which database they refer to. Except for the database implementation, the rest of the code can be reused character for character. How can the program be arranged to minimize repetitive code? How can the code be made reconfigurable, so that you can configure it using either database implementation?

The answer is a familiar one: if a module is an object, then a template for module is a class. Just like a class describes the common parts of all its instances, a class can describe the parts of a module that are common to all of its possible configurations.

The browser definition therefore becomes a class, instead of an object, and the database to use is specified as an abstract member of the class.

```
abstract class Browser {
```

```
val database: Database
def recipesUsing(food: Food) =
  database.allRecipes.filter(recipe =>
    recipe.ingredients.contains(food))
def displayCategory(category: database.FoodCategory) {
  //...
}
}
```

Database also becomes a class, including as much as possible that is common between all databases, and declaring the missing parts that a database must define. In this case, all database modules must define methods for `allFoods` and `allRecipes`, but since they can use an arbitrary definition, the methods must be left abstract in the `Database` class. The `foodNamed` method, by contrast, can be defined in the abstract `Database` class.

```
abstract class Database {
  def allFoods: List[Food]
  def allRecipes: List[Recipe]
  def foodNamed(name: String) =
    allFoods.find(f => f.name == name)
  case class FoodCategory(name: String, foods: List[Food])
  def allCategories: List[FoodCategory]
}
```

The simple database must be updated to inherit from this `Database`:

```
object SimpleDatabase extends Database {
  //...
}
```

Then, a specific browser module is made by instantiating the `Browser` class and specifying which database to use.

```
object SimpleBrowser extends Browser {
  val database = SimpleDatabase
}
```

```
scala> val apple = SimpleDatabase.foodNamed("Apple")
apple: Option[Food] = Some(Apple)
scala> SimpleBrowser.recipesUsing(apple.get)
res1: List[Recipe] = List(fruit salad)
```

Now you can create a second database, and use the same browser with it.

```
object StudentDatabase extends Database {
    object FrozenFood extends Food("FrozenFood")

    object HeatItUp extends Recipe(
        "heat it up",
        List(FrozenFood),
        "Microwave the 'food' for 10 minutes.")

    def allFoods = List(FrozenFood)
    def allRecipes = List(HeatItUp)
    def allCategories = List(
        FoodCategory("edible", List(FrozenFood)))
}

object SillyBrowser extends Browser {
    val database = StudentDatabase
}
```

25.3 Splitting modules into traits

Often a module is too large to fit comfortably into a single file. When that happens, you can use traits to split a module into separate files.

For example, suppose you wanted to move categorization code out of the main `Database` file and into its own. You can create a trait for the code like this:

```
trait FoodCategories {
    case class FoodCategory(name: String, foods: List[Food])
}
```

Now the `Database` class can mix in this trait instead of defining `FoodCategory` itself:

```
abstract class Database extends FoodCategories {  
    //...  
}
```

A trick is necessary if you want to use this approach and have the traits refer to the contents of each other. For example, suppose you try to divide SimpleDatabase into a two traits, one for foods and one for recipes:

```
trait SimpleFoods {  
    object Pear extends Food("Pear")  
    def allFoods = List(Apple, Pear)  
    def allCategories = Nil  
}  
  
trait SimpleRecipes {  
    object FruitSalad extends Recipe(  
        "fruit salad",  
        List(Apple, Pear), // uh oh  
        "Mix it all together.")  
    def allRecipes = List(FruitSalad)  
}  
  
object SimpleDatabase extends Database  
with SimpleFoods with SimpleRecipes
```

The problem here is that Pear is located in a different trait from the one that uses it, and so it is out of scope. The compiler has no idea that SimpleRecipes is only ever mixed together with SimpleFoods.

So tell it. Scala provides the *self type* for precisely this situation. Technically, a self type is an assumed type for this whenever this is mentioned within the class. Pragmatically, a self type adds extra requirements to any class the trait is mixed into. If you have a trait that is only ever used when mixed in with another trait or traits, then you can specify that those other traits should be assumed. In the present case, the self type can be SimpleDatabase.

```
trait SimpleRecipes {
```

```
self: SimpleDatabase.type =>
object FruitSalad extends Recipe(
    "fruit salad",
    List(Apple, Pear),    // now Pear is in scope
    "Mix it all together.")

def allRecipes = List(FruitSalad)
}
```

Now Pear is in scope as `this.Pear`, because `this` is assumed to be a `SimpleDatabase`.

25.4 Runtime linking

One final feature of Scala modules is worth emphasizing: they can be linked together at runtime, and you can decide which modules will link to which depending on runtime computations. For example, here is a small program that chooses a database at runtime and then prints out all the apple recipes in it:

```
object GotApples {
  def main(args: Array[String]) {
    val db: Database =
      if(args(0) == "student")
        StudentDatabase
      else
        SimpleDatabase
    object browser extends Browser {
      val database = db
    }
    val apple = SimpleDatabase.foodNamed("Apple").get
    for(recipe <- browser.recipesUsing(apple))
      println(recipe)
  }
}
```

Now, if you use the standard database, you will find a recipe for fruit salad. If you use the student database, then you will find no recipes at all using apples.

```
$ scala GotApples simple  
fruit salad  
$ scala GotApples student  
$
```

25.5 Tracking module instances

Despite using the same code, the different browser and database modules created above really are separate modules. This means that each module has its own contents, including any nested classes. `FoodCategory` in `SimpleDatabase` is a different class from `FoodCategory` in `StudentDatabase`!

```
scala> val category = StudentDatabase.allCategories.head  
category: StudentDatabase.FoodCategory =  
FoodCategory(edible,List(FrozenFood))  
  
scala> SimpleBrowser.displayCategory(category)  
<console>:12: error: type mismatch;  
      found   : StudentDatabase.FoodCategory  
      required: SimpleBrowser.database.FoodCategory  
                           SimpleBrowser.displayCategory(category)
```

If instead you prefer all `FoodCategory`s to be the same, you can accomplish this by moving the definition of `FoodCategory` outside of any class or trait. The choice is yours, but as it is written, each Database gets its own, unique `FoodCategory` class.

The above two classes really are different, so the compiler is correct to complain. Sometimes, though, you will encounter a case where two types are the same but the compiler cannot verify it. You will see the compiler complaining that two types are not the same, even though you as the programmer know they perfectly well are.

In such cases you can often fix the problem using *singleton types*. For example, in the GotApples program, the type checker does not know that db and browser.database are the same. This will cause type errors if you try to pass categories between the two objects:

```
object GotApples {  
    // same definitions...  
  
    for (category <- db.allCategories)  
        browser.displayCategory(category)  
  
    // ...  
}  
  
GotApples2.scala:14: error: type mismatch;  
  found   : db.FoodCategory  
  required: browser.database.FoodCategory  
          browser.displayCategory(category)  
                                ^  
  
one error found
```

To avoid this error, you need to inform the type checker that they are the same object. You can do this by changing the definition of browser.database as follows:

```
object browser extends Browser {  
    val database: db.type = db  
}
```

This definition is the same as before except that database has the funny-looking type db.type. The “.type” on the end means that this is a singleton type. A singleton type is extremely specific and holds only one object, in this case whichever object is referred to by db. Usually such types are too specific to be useful, which is why the compiler is reluctant to insert them automatically. In this case, though, the singleton type allows the compiler to know that db and browser.database are the same object, enough information to eliminate the above type error.

25.6 Conclusion

This chapter has shown how to use Scala's objects as modules. In addition to simple static modules, this approach gives you a variety of ways to create abstract, reconfigurable modules. There are actually even more abstraction techniques than shown, because anything that works on a class, also works on a class used to implement a module. As always, how much of this power you use should be a matter of taste.

Modules are part of programming in the large, and thus are hard to experiment with. You need a large program before it really makes a difference. Nonetheless, after reading this chapter you know which Scala features to think about when you want to program in a modular style. Think about these techniques when you write your own large programs, and recognize these coding patterns when you see them in other people's code.

Chapter 26

Annotations

Annotations are structured information added to program source code. Like comments, they can be sprinkled throughout a program and attached to any variable, method, expression, or other program element. Unlike comments, they have structure, thus making them easier to machine process.

This chapter shows how to use annotations in Scala. It shows the syntax of them in general, and then it shows how to use several standard annotations.

This chapter does not show how to write new annotation processing tools. In general that topic is beyond the scope of this book, but you can see one technique in [Chapter 27](#).

26.1 Why have annotations?

There is no perfect programming language. We have done our best with Scala to include today's best proven ideas. Many more ideas are under current research, though, and inevitably at least some of those ideas will prove their worth in the future. The state of the art language in ten years will be better than the state of the art language of today.

On top of that, there are a variety of special circumstances where language support can help, but there is no way for a single language to support all of them. Many organizations have their own testing framework, and it would be useful to mark which parts of a code base correspond to which parts of the organization's testing framework, but no one language can support every organization's testing framework. Many programs work in a specialized domain such as scientific computing or symbolic logic, and while a

general purpose language often supports one or two of these domains, there is no way to support all of them.

Because no single language can include specialized support for every organization and every programming domain, Scala includes a system of [annotations](#). While Scala's functions ([Chapter 8](#)), pattern matching ([Chapter 12](#)), and implicits ([Chapter 19](#)) go a long way, even these powerful features can only help you support special domains to a certain extent. While Scala should need language extensions less often than other general-purpose languages, a single language can fundamentally only go so far.

The way annotations work is that programmers put annotations in their code that are only meaningful for some special circumstance. They mark methods as tests, or they mark matrices as sparse, or they append a proof to a logical proposition. The core Scala compiler ignores these annotations except to check that they are well-formed. It is up to separate tools—[metaprogramming](#) tools—to pay attention to these annotations and do something useful.

26.2 Syntax of annotations

A typical use of an annotation looks like this:

```
@deprecated def bigMistake() = //...
```

The annotation is the `@deprecated` part, and it applies to the entirety of the `bigMistake` method (not shown—it's too embarrassing). In this case, the method is being marked as something the author of `bigMistake` wishes you not to use. Maybe `bigMistake` will be removed entirely from a future version of the code.

In the previous example, it is a method that is annotated as `@deprecated`. There are other places annotations are allowed, too. Annotations are allowed on any other kind of declaration or definition, including `vals`, `vars`, `defs`, `classes`, `objects`, `traits`, and `types`. The annotation applies to the entirety of the declaration or definition which follows it.

```
@deprecated class QuickAndDirty {  
    //...  
}
```

Annotations can also be applied to an expression, as with the `@unchecked` annotation for pattern matching (see [Chapter 12](#)). To do so, place a colon (“`:`”) after the expression and then write the annotation. Syntactically, it looks like the annotation is being used as a type:

```
(e: @unchecked) match {  
    // non-exhaustive cases...  
}
```

Finally, annotations can be placed on types. Annotated types are described later in this chapter.

So far the annotations shown have been simply an at sign followed by an annotation class. Such simple annotations are common and useful, but annotations have a richer general form:

```
@annot(exp, exp, ...) {val name=const, ..., val name=const}
```

The `annot` specifies the class of annotation. All annotations must include that much. The `exp` parts are arguments to the annotation. For annotations like `@deprecated` that do not need any arguments, you would normally leave off the parentheses, but you can write `@deprecated()` if you like. For annotations that do have arguments, place the arguments in parentheses: `@serial(1234)`.

The precise form the arguments you may give to an annotation depends on the particular annotation class. Most annotation processors only let you supply immediate constants such as `123` or `"hello"`. The compiler itself supports arbitrary expressions, however, so long as they type check. Some annotation classes can make use of this, for example to let you refer to other variables that are in scope:

```
@cool val normal = "Hello"  
@coolerThan(normal) val fonzy = "Heeyyy"
```

The `name=const` pairs in the general syntax are available for more complicated annotation that have optional arguments. Since the arguments are optional, the names of any arguments you give must include the name of the arguments you are including. To keep things simple, the arguments themselves are restricted to be immediate constants.

26.3 Standard annotations

Scala includes several standard annotations. They are for features that are used widely enough to merit putting in the language specification, but that are not fundamental enough to merit their own syntax. Over time, there should be a trickle of new annotations that are added to the standard in just the same way.

Deprecation

Sometimes you write a class or method that you later wish you had not. Once it is available, though, code written by other people might call the method. Thus, you cannot simply delete the method instantly, because you would cause other people's code to stop compiling.

Deprecation lets you gracefully remove a method or class that turns out to be a mistake. You mark the method or class as deprecated, and then anyone who calls that method or class will get a deprecation warning. They had better head this warning and update their code! The idea is that after a suitable amount of time has passed, you feel safe in assuming that all reasonable clients will have stopped accessing the deprecated class or method and thus that you can safely remove it.

You mark a method as deprecated simply by writing `@deprecated` before it. For example:

```
@deprecated def bigMistake() = //...
```

Such an annotation will cause Scala to emit deprecation warnings whenever Scala code accesses the method.

Volatile fields

Concurrent programming does not mix well with shared mutable state. For this reason, the focus of Scala's concurrency support is message passing and a minimum of shared mutable state. See [Chapter 23](#) for the details.

Nonetheless, sometimes programmers want to use mutable state in their concurrent programs. The `@volatile` annotation helps in such cases. It informs the compiler that the variable in question will be used by multiple threads. Such variables are implemented in a way that reads and writes to

the variable are slower, but accesses from multiple threads behave more predictably.

The `@volatile` keyword gives different guarantees on different platforms, so this book cannot document exactly what you get. In general, though, if a mutable variable is going to be accessed from multiple threads, mark it `@volatile`.

Binary serialization

Many languages include a framework for binary *serialization*. A serialization framework helps you convert objects into a stream of bytes and *vice versa*. This is useful if you want to save the objects to disk or send them over the network. XML can help with the same goals (see [Chapter 22](#)), but it has different trade offs regarding speed, space usage, flexibility, and portability.

Scala does not have its own serialization framework. Instead, you should use a framework from your underlying platform. What Scala does is provide three annotations that are useful for a variety of frameworks. Also, the Scala compiler for the Java platform interprets these annotations in the Java way (see [Chapter 27](#)).

The first annotation indicates whether a class is serializable at all. Most classes are serializable, but for example a handle to a socket or to a GUI window cannot be serialized. By default, a class is not considered serializable. You should add a `@Serializable` annotation to any class you would like to be deemed serializable.

The second annotation helps deal with serializable classes changing as time goes by. You can attach a serial number to the current version of a class by adding an annotation like `@SerialVersionUID(1234)`, where 1234 should be replaced by your serial number of choice. The framework should store this number in the generated byte stream. When you later reload that byte stream and try to convert it to an object, the framework can check that the current version of the class has the same version number as the version in the byte stream. If you want to make a serialization-incompatible change to your class, then you can change the version number. The framework will then automatically refuse to load old instances of the class.

Finally, Scala provides a `@transient` annotation for fields that should not be serialized at all. If you mark a field as `@transient`, then the framework should not save the field even when the surrounding object is serialized.

When the object is loaded, the field will be restored to the default value for the class.

Automatic getters and setters

Scala code normally does not need explicit getters and setters for fields, because Scala blends the syntax for field access and method invocation. Some platform-specific frameworks do expect getters and setters, however. For that purpose, Scala provides the `@scala.reflect.BeanProperty` annotation. If you annotate a variable with this annotation, the compiler will automatically generate getters and setters for you. If you annotate a variable named `crazy`, the getter will be named `getCrazy` and the setter will be named `setCrazy`.

The generated getter and setter are only available after a compilation pass completes. Thus, you cannot call these getters and setters from code you compile at the same time as the annotated fields. This should not be a problem in practice you can use the fields directly in Scala. This feature is intended to support frameworks that expect getters and setters to be in a regular form.

Unchecked

The `@unchecked` annotation is interpreted by the compiler during pattern matches. It tells the compiler not to worry if the match expression seems to leave out some cases. See [Chapter 12](#) for details.

26.4 Conclusion

This chapter has described how to use annotations, and how to use several standard annotations.

Chapter 27

Combining Scala and Java

Scala code is often used in tandem with large Java programs and frameworks. Since Scala is highly compatible with Java, most of the time you can combine the languages without worrying very much. For example, standard frameworks such as Swing, Servlets, and JUnit are known to work just fine with Scala. Nonetheless, from time to time you will run into some issue with combining Java and Scala. Further, you might just have an engineer's motives and want to know more about how Scala works under the hood.

This chapter describes two aspects of combining Java and Scala. First, it discusses how Scala is translated to Java, which is especially important if you call Scala code from Java. Second, it discusses the use of Java annotations in Scala, an important feature if you want to use Scala with an existing Java framework.

27.1 Translation details

Most of the time you can just think of Scala at the source code level. However, you will have a richer understanding of how the system works if you know something about its translation. Further, if you call Scala code from Java, you will need to know what Scala code looks like from a Java point of view.

General rules

Scala is implemented as a translation to standard Java bytecodes. As much as possible, Scala features map directly onto the equivalent Java features. Classes, methods, primitive types, strings, and exceptions all appear exactly the same in Java bytecode as they did in Scala source code.

To make this happen required an occasional hard choice in the design of Scala. For example, it might have been nice to resolve overloaded methods at runtime, using run-time types, rather than at compile time. Such a design would break with Java's, however, making it much trickier to mesh Java and Scala. In this case, Scala stays with Java's overloading resolution, and thus Scala methods and method calls can map directly to Java methods and method calls.

For other features Scala has its own design. For example, traits have no equivalent in Java. Similarly, while both Scala and Java have generic types, the details of the two systems clash. For language features like these, Scala code cannot be mapped directly to a Java construct, so it must be encoded using some combination of the structures Java does have.

For these features that are mapped indirectly, the encoding is not fixed. There is an ongoing effort to make the translations as simple as possible, so by the time you read this, some details may be different than at the time of writing. You can find out what translation your current Scala compiler uses by examining the `.class` files with tools like `javap`.

Those are the general rules. Consider now some special cases.

Value types

A value type like `Int` can be translated in two different ways to Java. Whenever possible, the compiler translates a Scala `Int` to a Java `int` for performance. Sometimes this is not possible, though, because the compiler is not sure whether it is translating an `Int` or some other data type. For example, a particular `List[Any]` might hold only `Ints`, but the compiler has no way to be sure.

In cases like this, where the compiler is unsure whether an object is a value type or not, the compiler uses objects and relies on wrapper classes. Wrapper classes, for example `java.lang.Integer`, allow a value type to be wrapped inside a Java object and thereby manipulated by code that needs

objects.

The implementation of value types is discussed further in [Section 10.16](#).

Singleton objects

Java has no exact equivalent to a singleton object, but it does have static methods. The Scala translation uses static methods as much as possible, but sometimes it must fall back on a more general approach.

In every case, the compiler creates a class for the object with a dollar sign added to the end. For a singleton object named App, the compiler produces a Java class named App\$. This class has instance methods, not static methods, for each method of the Scala App object. The Java class also has a single static field named MODULE\$ to hold the one instance of the class that is created. For example, suppose you compile the following singleton object:

```
object App {  
    def main(args: Array[String]) {  
        println("Hello, world!")  
    }  
}
```

Scala will generate a Java App\$ class with the following fields and methods:

```
$ javap App$  
public final class App$ extends java.lang.Object  
implements scala.ScalaObject{  
    public static final App$ MODULE$;  
    public static {};  
    public App$();  
    public void main(java.lang.String[]);  
    public int $tag();  
}
```

An important special case is if you have a standalone singleton object, which does not come with a class of the same name. For example, you might have a singleton object named App, and not have any class named App. In that case, the compiler creates a Java class named App that has a static forwarder method for each method of the Scala singleton object:

```
$ javap App
Compiled from "App.scala"
public final class App extends java.lang.Object{
    public static final int $tag();
    public static final void main(java.lang.String[]);
}
```

To contrast, if you did have a class named `App`, Scala would create a corresponding Java `App` class to hold the members of your true class. In that case it would not add any forwarding methods for the same-named singleton object, and Java code would have to access the singleton via the `MODULE$` field.

This difference in behavior is due to a restriction in Java's class file format: The JVM does not let you define a static method with the same name and signature as an instance method in the same class. On the other hand, Scala imposes no such restriction: A class can well define a method with the same name and signature as its companion object. Therefore, it would not be safe to install forwarders for all static methods, because they might clash with an instance method of the companion class.

Generics

As of Scala 2.6.1, the generic types of Java and the generic types of Scala are incompatible. As a result, generic types in one language are not visible in the other. Generic classes and methods in each language appear as their non-generic versions in the other language. A `List[Int]` in Scala looks like a plain `List` in Java. This *type erasure* is discussed more in [Chapter 12](#).

We expect that this will change in Scala version 2.6.2 though. Hopefully, generics in Scala are then visible as generics in Java and *vice versa*.

Traits as interfaces

Compiling any trait creates a Java interface of the same name. This interface is usable as a Java type, and it lets you call methods on Scala objects through variables of that type.

Implementing a trait in Java is another story. One special case is important, however. If you make a Scala trait that includes only abstract methods, then that trait will be translated directly to a Java interface, with no other code

to worry about. Essentially this means that you can write a Java interface in Scala syntax if you like.

27.2 Annotations

Scala's general annotations system is discussed in [Chapter 26](#). This section discusses Java-specific aspects of annotations.

Additional effects from standard annotations

Several annotations cause the compiler to emit extra information when running on Java. When the compiler sees such an annotation, it processes it according to the general Scala rules, and then it does something extra for Java.

Deprecation For any method or class marked `@deprecated`, the compiler will add Java's own deprecation annotation to the emitted code. Because of this, Java compilers can issue deprecation warnings when Java code accesses the method.

Volatile fields Likewise, any field marked `@volatile` in Scala is given the `@volatile` Java annotation in the emitted code. Thus, volatile fields in Scala behave exactly according to Java's semantics, and accesses to volatile fields are sequenced precisely according to the rules of the Java Memory Model.

Serialization Scala's three standard serialization annotations discussed are all translated to Java equivalents. A `@serializable` class has the Java `Serializable` interface added to it. A `@SerialVersionUID(1234L)` annotation is converted to the following Java field definition:

```
// Java serial version marker  
private final static long serialVersionUID = 1234L
```

Any variable marked `@transient` is given the Java `transient` modifier.

Exceptions thrown

Scala does not check that thrown exceptions are caught. That is, Scala has no equivalent to Java's `throws` declarations on methods. All Scala methods are translated to Java methods that declare no thrown exceptions.¹

The reason this feature is omitted from Scala is that the Java experience with it has not been purely positive. Because annotating methods with `throws` clauses is a heavy burden, too many developers write code that swallows and drops exceptions, just to get the code to compile without adding all these `throws` clauses. They intend to improve the exception handling later, but experience shows that all too often time-pressed programmers will never come back and add proper exception handling. The twisted result is that this well-intentioned feature often ends up making code *less* reliable. A large amount of production Java code swallows and hides run-time exceptions, and the reason it does so is to satisfy the compiler.

Sometimes when interfacing to Java you will need Scala code to have Java-friendly annotations about which exceptions your methods throw. All you have to do is mark your methods with `@throws` annotations. For example, the following Scala method has a method marked as throwing `IOException`:

```
import java.io._  
class Reader(fname: String) {  
    private val in =  
        new BufferedReader(new FileReader(fname))  
    @throws(classOf[IOException])  
    def read() = in.read()  
}
```

Here is how it looks from Java:

```
$ javap Reader  
Compiled from "Reader.scala"  
public class Reader extends java.lang.Object implements scala.ScalaObject{  
    public Reader(java.lang.String);  
    public int read() throws java.io.IOException;
```

¹The reason it all works is that Java bytecode verifier does not check the declarations, anyway! The compiler checks, but not the verifier.

```
    public int $tag();  
}  
$
```

Note that the `read()` method is annotated as throwing an `IOException`.

Java annotations

Existing annotations from Java frameworks can be used in Scala code. Any Java framework will see the annotations you write just as if you were writing Java code instead of Scala code.

A wide variety of Java packages use annotations. As an example, consider JUnit 4. JUnit is a framework for writing automated tests and for running those tests. The latest version, JUnit 4, uses annotations to indicate which parts of your code are tests. The idea is that you write a lot of tests for your code, and then you run those tests whenever you change the source code. That way, if your changes add a new bug, one of the tests will fail and you will find out immediately.

Writing a test is easy. You simply write a method in a top-level class that exercises your code, and you use an annotation to mark the method as a test. It looks like this:

```
import org.junit.Test  
import org.junit.Assert.assertTrue  
  
class SetTest {  
  
    @Test  
    def testMultiAdd {  
        val set = Set() + 1 + 2 + 3 + 1 + 2 + 3  
        assertTrue(set.size == 3)  
    }  
}
```

The meat of this test is in the `testMultiAdd` method. This test adds multiple items to a set and makes sure that each one is only added one time. The `assertTrue` method comes from the Java code of JUnit and checks that the expression evaluates to `true` when it is executed. If it does not evaluate to `true`, then the test fails.

The test is marked using the annotation `org.junit.Test`. Note that this annotation has been imported, so it can be referred to as simply `@Test` instead of the more cumbersome `@org.junit.Test`.

That is all. Run a test using any JUnit test runner, for example:

```
$ scala -cp junit-4.3.1.jar:. org.junit.runner.JUnitCore SetTest
JUnit version 4.3.1

.
Time: 0.023

OK (1 test)
```

Writing your own annotations

To make an annotation that is visible to Java reflection, you must use Java notation and compile it with `javac`. Here is an example annotation:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Ignore { }
```

After compiling the above with `javac`, you can use the annotation as follows:

```
object Tests {
    @Ignore
    def testData = List(0, 1, -1, 5, -5)

    def test1 {
        assert(testData == (testData.head :: testData.tail))
    }

    def test2 {
        assert(testData.contains(testData.head))
    }
}
```

In this example, `test1` and `test2` are supposed to be test methods, but `testData` should be ignored even though its name starts with “test.”

To see when these annotations are present, you can use the Java reflection API's. Here is sample code to show how it works:

```
for {
    method <- Tests.getClass.getMethods
    if method.getName.startsWith("test")
    if method.getAnnotation(classOf[Ignore]) == null
} {
    println("found a test method: " + method)
}
```

The reflective methods `getClass` and `getMethods` are used here to go through all the fields of the input object's class. These are normal reflection methods. The annotation-specific part is the use of method `getAnnotation`. As of Java 1.5, many reflection objects have a `getAnnotation` method for searching for annotations of a specific type. In this case, the code looks for an annotation of our new `Ignore` type. Since this is a Java API, success is indicated by whether the result is `null` or is an actual annotation object.

Here is the code in action:

```
$ javac Ignore.java
$ scalac Tests.scala
$ scalac FindTests.scala
$ scala FindTests
found a test method: public void Tests$.test2()
found a test method: public void Tests$.test1()
```

Be aware that when you use Java annotations you have to work within their limitations. For example, you can only use constants, not expressions, in the arguments to annotations. You can support `@serial(1234)` but not `@serial(x*2)`, because `x*2` is not a constant.

27.3 Existential types

All Java types have a Scala equivalent. This is necessary so that Scala code can access any legal Java class. Most of the time the translation is straightforward. `Pattern` in Java is `Pattern` in Scala, and `Iterator<Component>` in Java is `Iterator[Component]` in Scala. For

some cases, though, the Scala types you have seen so far are not enough. What should be done with Java wildcard types such as `Iterator<?>` and `Iterator<? extends Component>`? What should be about raw types like `Iterator`, where the type parameter is omitted? For wildcard types and raw types, Scala uses an extra kind of type called an *existential type*.

Existential types are a fully supported part of the language, but in practice they are mainly used when viewing Java types from Scala. This section briefly overviews how existential types work, but mostly this is only useful so that you can understand compiler error messages that appear if there is a type error when accessing Java code.

The general form of an existential type is as follows:

```
type forSome { declarations }
```

The `type` part is an arbitrary Scala type, and the `declarations` part is a list of abstract vals and types. The interpretation is that the declarations variables and types exist but are unknown, just like abstract members of a class. The `type` is then allowed to refer to the declared variables and types even though it is unknown what they refer to.

Take a look at some concrete examples. A Java `Iterator<?>` would be written in Scala as:

```
Iterator[T] forSome { type T }
```

Read this from left to right. This is a `Iterator` of T's for some type T. The type T is unknown, and could be anything, but it is known to be fixed for this particular `Iterator`. Similarly, a Java `Iterator<? extends Component>` would be viewed in Scala as:

```
Iterator[T] forSome { type T <: Component }
```

This is an `Iterator` of T, for some type T that is a subtype of `Component`. In this case T is completely unknown, but it is sure to be subtype of `Component`.

In simple cases, you use an existential type just as if the `forSome` were not there. Scala will check that the program is sound even though the types and values in the `forSome` clause are unknown. For example, suppose you had the following Java class:

```
/* This is a Java class with wildcards */
```

```
class Wild {  
    Collection<?> contents() {  
        Collection<String> stuff = new Vector<String>();  
        stuff.add("a");  
        stuff.add("b");  
        stuff.add("see");  
        return stuff;  
    }  
}
```

If you access this in Scala code you will see that it has an existential type:

```
scala> val contents = (new Wild).contents  
contents: java.util.Collection[T] forSome { type T } = [a,  
b, see]
```

If you want to find out how many elements are in this collection, you can simply ignore the existential part and call the `size` method as normal:

```
scala> contents.size()  
res0: Int = 3
```

In more complicated cases, existential types can be more awkward, because there is no way to name the existential type. For example, suppose you wanted to create a mutable Scala set and initialize it with the elements of `contents`.

```
import scala.collection.mutable.Set  
val iter = (new Wild).contents.iterator  
val set = Set.empty[????]      // what type goes here?  
while (iter.hasMore)  
    set += iter.next()
```

A problem strikes immediately on line 2. There is no way to name the type of elements in the Java collection, so you cannot write down the code type of `set`. To work around this, there are two tricks you should consider:

1. When passing an existential type into a method, move type parameters from the `forSome` clause to type parameters of the method. Inside the

body of the method, you can use the type parameters to refer to the types that were in the `forSome` clause.

2. Instead of returning an existential type from a method, return an object that has abstract members for each of the types in the `forSome` clause. (See [Chapter 18](#) for information on abstract members.)

Using these two tricks together, the previous code can be written as follows:

```
abstract class SetAndType {  
    type Elem  
    val set: Set[Elem]  
}  
  
def javaSet2ScalaSet[T](jset: Collection[T]): SetAndType = {  
    val sset = Set.empty[T] // now T can be named!  
    val iter = jset.iterator  
    while (iter.hasNext)  
        sset += iter.next()  
  
    return new SetAndType {  
        type Elem = T  
        val set = sset  
    }  
}
```

You can see why Scala code normally does not use existential types. To do anything sophisticated with them, you tend to convert them to use abstract members. You may as well use abstract members to begin with.

27.4 Conclusion

Most of the time, you can ignore how Scala is implemented and simply write and run your code. Sometimes it is nice to “look under the hood,” however, and so this chapter has gone into two aspects of Scala’s implementation on Java: what the translation looks like, and how you access Java-compatible annotations. These topics become important when you need interoperation between Scala and Java.

Chapter 28

Combinator Parsing

Occasionally it can be useful to whip up a processor for a small language. However, you must solve the problem of *parsing* sentences in the language you want to process. Essentially, you have only a few choices.

One choice is to roll your own parser (and lexical analyzer). If you are not an expert this is hard. If you are an expert, it is still time-consuming to do this.

An alternative choice is to use a parser generator. There exist quite a few of these generators. Some of the better known are Yacc or Bison for parsers written in C and ANTLR for parsers written in Java. You'll probably also need a scanner generator such as Lex, Flex, or JFlex to go with it. This might be the best solution, except for a couple of inconveniences: You need to learn new tools, including their—sometimes obscure—error messages. You also need to figure out how to connect the output of these tools to your program. This might limit the choice of your programming language, and complicate your tool chain.

This chapter presents a third alternative: Instead of using the stand-alone domain specific language of a parser generator you will use an *embedded domain specific language* (or: embedded DSL for short). The embedded DSL consists of a library of *parser combinators*. These are functions and operators defined in Scala that are building blocks for parsers. The building blocks follow one by one the constructions of a context-free grammar, so they are very easy to understand.

This chapter introduces only a single language feature that was not explained before: this-aliasing in [Section 28.5](#). It does however heavily use

several other features that were explained in previous chapters. Among others, parameterized types, abstract types, functions as objects, operator overloading, by-name parameters, and implicit conversions all play important roles. The chapter shows how these language elements can be combined in the design of a very high-level library.

The concepts explained in this chapter tend to be a bit more advanced than previous chapters. If you have a good grounding in compiler construction, you'll profit from it reading this chapter, because it will help you put things better in perspective. However, the only prerequisite for understanding this chapter is that you know about regular and context-free grammars.

28.1 Example: Arithmetic Expressions

Let's start with an example: Say you want to construct a parser for arithmetic expressions consisting of integer numbers, parentheses, and the binary operators `+`, `-`, `*`, and `/`. The first step is always to write down a grammar for the language to be parsed. For arithmetic expressions, this grammar reads as follows:

```
expr      ::=  term { '+' term | '-' term } .
term     ::=  factor { '*' factor | '/' factor } .
factor   ::=  numericLit | '(' expr ')' .
```

Here, “`|`” denotes alternative productions. `{ ... }` denotes repetition (zero or more times) whereas `[...]` denotes an optional occurrence.

This context-free grammar defines formally a language of arithmetic expressions: Every expression (represented by `expr`) is a `term`, which can be followed by a sequence of `‘+’` or `‘-’` operators and further terms. A `term` is a `factor`, possibly followed by a sequence of `‘*’` or `‘/’` operators and further factors. A `factor` is either a numeric literal or an expression in parentheses. Note that the grammar already encodes the relative precedence of operators. For instance, `‘*’` binds more tightly than `‘+’`, because a `‘*’` operation gives a `term`, whereas a `‘+’` operation gives an `expr`, and `exprs` can contain `terms` but a `term` can contain an `expr` only when the latter is enclosed in parentheses.

Now that you have defined the grammar, what's next? If you use Scala's combinator parsers, you are basically done! You only need to perform some systematic text replacements and wrap the parser in a class as follows:

```
import scala.util.parsing.combinator.syntactical._  
class Arith extends StandardTokenParsers {  
    lexical.delimiters +== List("(", ")", "+", "-", "*", "/")  
    def expr : Parser[Any] =  
        term ~ rep(("+" ~ term) | ("-" ~ term))  
    def term : Parser[Any] =  
        factor ~ rep("*" ~ factor | "/" ~ factor)  
    def factor: Parser[Any] =  
        "(" ~ expr ~ ")" | numericLit  
}
```

The parser for arithmetic expressions is a class that inherits from the class `StandardTokenParsers`. This class provides the basic machinery to write a standard parser. The class builds on a standard lexer that recognizes Java-like tokens consisting of strings, integers, and identifiers. The lexer skips over whitespace and Java-like comments. Both multi-line comments `/* ... */` and single line comments `// ...` are supported. The lexer still needs to be configured with a set of *delimiters*. These are tokens consisting of special symbols that the lexer should recognize. In the case of arithmetic expressions, the delimiters are “(”, “)”, “+”, “-”, “*”, and “/”. They are declared to the parsing combinator system by the line:

```
lexical.delimiters +== List("(", ")", "+", "-", "*", "/")
```

Here, `lexical` refers to the lexer component inherited from class `StandardTokenParsers` and `delimiters` is a mutable set in this component. The “`+==`” operation enters the desired delimiters into the set.

The next three lines represent the productions for arithmetic expressions. As you can see, they follow very closely the productions of the context-free grammar. In fact, you could generate this part automatically from the context-free grammar, by performing a number of simple text replacements:

1. Every production becomes a method, so you need to prefix it with `def`.

2. The result type of each method is `Parser[Any]`, so you need to change the “produces” sign `:=` to `: Parser[Any] =`. You’ll find out below what the type `Parser[Any]` signifies, and also how to make it more precise.
3. In the grammar, sequential composition was implicit, but in the program it is expressed by an explicit operator “`~`”. So you need to insert a “`~`” between every two consecutive symbols of a production.
4. Repetition is expressed `rep(...)` instead of `{...}`. Analogously (not shown in the example), option is expressed `opt(...)` instead of `[...]`.
5. The point “`.`” at the end of each production is omitted—you can however write a semicolon “`;`” if you prefer.

That’s it! You have a parser for arithmetic expressions. As you can see, the combinator parsing framework gives you a fast path to construct your own parsers. In part this is due to the fact that a lot of functionality is “pre-canned” in the `StandardTokenParsers` class. But the parsing framework as a whole is also easy to adapt to other scenarios. For instance, it is quite possible to configure the framework to use a different lexer (including one you write). In fact, the lexer itself can be written with the same combinator parsers that also underlie the parser for arithmetic expressions.

28.2 Running Your Parser

You can test your parser with the following a small program:

```
object ArithTest extends Arith {  
    def main(args: Array[String]) {  
        val tokens = new lexical.Scanner(args(0))  
        println("input: "+args(0))  
        println(phrase(expr)(tokens))  
    }  
}
```

The `ArithTest` object defines a `main` method which parses the first command line argument that’s passed to it. It first creates a `Scanner` object that

reads the first input argument and converts it to a token sequence named `tokens`. It then prints the original input argument, and finally prints its parsed version. Parsing is done by the expression

```
phrase(expr)(tokens)
```

This expression applies the parser `phrase(expr)` to the token sequence `tokens`. The `phrase` method is a special parser. It takes another parser as argument, applies this parser to an input sequence, and at the same time makes sure that after parsing the input sequence is completely read. So `phrase(expr)` is like `expr`, except that `expr` can parse parts of input sequences, whereas `phrase(expr)` succeeds only if the input sequence is parsed from beginning to end.

You can run the arithmetic parser with the following command:

```
scala ArithTest "2 * (3 + 7)"  
input: 2 * (3 + 7)  
[1.12] parsed: ((2 ~ List((* ~ ((( ~ ((3 ~ List()) ~ List((+  
~ (7 ~ List())))))) ~ )))) ~ List())
```

The output tells you that the parser successfully analyzed the input string up to position [1.12]. That means the first line and the twelfth column, or, otherwise put, all the input string was parsed. Disregard for the moment the result after “parsed:”—it is not very useful, and you will find out later how to get more specific parser results.

You can also try to introduce some input string that is not a legal expression. For instance, you could write one closing parenthesis too many:

```
scala ArithTest "2 * (3 + 7))"  
input: 2 * (3 + 7))  
[1.12] failure: end of input expected
```

```
2 * (3 + 7))  
^
```

Here, the `expr` parser parsed everything until the final closing parenthesis, which does not form part of the arithmetic expression. The `phrase` parser then issued an error message which said that it expected the input to end at the point of the closing parenthesis.

28.3 Another Example: JSON

Let's try another example. JSON, the JavaScript Object Notation, is a popular data interchange format. You'll now find out how to write a parser for it. Here is the syntax of JSON:

```
value = obj | arr | stringLit | numericLit |
        "null" | "true" | "false"
obj = "{" [members] "}"
arr = "[" [values] "]"
members = member {"," member}
member = stringLit ":" value
values = value {"," value}
```

A JSON value is an object, or an array, or a string, or a number, or one of the three reserved words `null`, `true`, or `false`. A JSON object is a (possibly empty) sequence of members separated by commas and enclosed in braces. Each member is a string/value pair where the string and the value are separated by a colon. Finally, a JSON array is a sequence of values separated by commas and enclosed in square brackets.

Here is an example JSON object:

```
{ "address book": {
    "name": "John Smith",
    "address": {
        "street": "10 Market Street",
        "city" : "San Francisco, CA",
        "zip" : 94111
    },
    "phone numbers": [
        "408 338-4238",
        "408 111-6892"
    ]
}}
```

Parsing JSON data is straightforward when using Scala's parser combinators. Here is the complete parser:

```

import scala.util.parsing.combinator.syntactical._
class JSON extends StandardTokenParsers {
    lexical.delimiters += ("{", "}", "[", "]", ":", ",")
    lexical.reserved += ("null", "true", "false")

    def value : Parser[Any] = obj | arr | stringLit | numericLit |
        "null" | "true" | "false"
    def obj   : Parser[Any] = "{" ~ repsep(member, ",") ~ "}"
    def arr   : Parser[Any] = "[" ~ repsep(value, ",") ~ "]"
    def member: Parser[Any] = stringLit ~ ":" ~ value
}

```

This parser follows the same structure as the arithmetic expression parser. The delimiters of JSON are "{", "}", "[", "]", ":", ",". There are also some *reserved words*: null, true, false. Reserved words are tokens that follow the syntax of identifiers, but that are reserved. Reserved words are communicated to the lexer by entering them into its reserved table:

```
lexical.reserved += ("null", "true", "false")
```

The rest of the parser is made up of the productions of the JSON grammar. The productions use one shortcut which simplifies the grammar: The repsep combinator parses a (possibly empty) sequence of terms that are separated by a given separator string. For instance, in the example above, repsep(member, ",") parses a comma-separated sequence of member terms. Otherwise, the productions in the parser correspond exactly to the productions in the grammar, just like it was the case for the arithmetic expression parsers.

To test the JSON parsers, let's change the framework a bit, so that the parser operates on a file instead of on the command line:

```

import scala.util.parsing.input.StreamReader
object JSONTest extends JSON {
    def main(args: Array[String]) {
        val reader = StreamReader(new java.io.FileReader(args(0)))
        val tokens = new lexical.Scanner(reader)
        println(phrase(value)(tokens))
    }
}

```

The `main` method in this program first creates a `StreamReader` object. This object represents an input stream of characters with positions; for every character that's read one can query its line and column numbers (both lines and columns start at 1). It then creates a `Scanner` over this stream reader. Finally the tokens returned from the scanner are parsed; they need to conform to the value production of the JSON grammar. If you store the “address book” object above into a file named `address-book.json` and run the test program on it you should get:

```
scala JSONTest address-book.json
[17.1] parsed: (({ ~ List(((address book ~ :) ~ ({ ~
List((name ~ :) ~ John Smith), ((address ~ :) ~ ({ ~
List((street ~ :) ~ 10 Market Street), ((city ~ :) ~ San
Francisco, CA), ((zip ~ :) ~ 94111))) ~ })), ((phone numbers
~ :) ~ ([ ~ List(408 338-4238, 408 111-6892)) ~ ])))) ~
}))) ~ }
```

28.4 Parser Output

The test run above succeeded; the JSON address book was successfully parsed. However, the parser output looks strange—it seems to be a sequence composed of bits and pieces of the input glued together with lists and “`~`” combinations. This parser output is not very useful. It is certainly less readable for humans than the input, but it is also too disorganized to be easily analyzable by a computer. It’s time to do something about this.

To figure out what to do, you need to know first what the individual parsers in the combinator frameworks return as a result (provided they succeed in parsing the input). Here are the rules:

1. Each parser written as a string (such as: `"{"` or ":" or `"null"`) returns the parsed string itself.
2. Each of the single-token parsers—`stringLit`, `numericLit`, and `ident`—also returns the parsed string itself.
3. A sequential composition `P ~ Q` returns the results of both `P` and of `Q`. These results are returned in an instance of a case class which is

also written “ \sim ”. So if P returns “true” and Q returns “?”, then the sequential composition $P \sim Q$ returns $\sim(\text{“true”}, \text{“?”})$, which prints as `(true ~ ?)`.

4. An alternative composition $P \mid Q$ returns the result of either P and Q (whichever one succeeds).
5. A repetition `rep(P)` or `repsep(P, separator)` returns the results of all runs of P as elements of a list.
6. An option `opt(P)` returns an instance of Scala’s `Option` type. It returns the `Some(R)` if P succeeds with result R and `None` if P fails.

With these rules you can now deduce *why* the parser output was as shown in the example above. However, the output is still not very convenient. It would be much better to map a JSON object into an internal Scala representation that represents the “meaning” of the JSON value. A representation which is natural would be as follows:

- A JSON object is represented as a Scala map of type `Map[String, Any]`. Every member is represented as a key/value binding in the map.
- A JSON array is represented as a Scala list of type `List[Any]`.
- A JSON string is represented as a Scala `String`.
- A JSON numeric literal is represented as a Scala `Int`.
- The values `true`, `false` and `null` are represented in as the Scala values with the same names.

To produce to this representation, you need to make use of two more combination forms for parsers, “ $\wedge\wedge$ ” and “ $\wedge\wedge\wedge$ ”.

The “ $\wedge\wedge$ ” operator *transforms* the result of a parser. Expressions using this operator have the form $P \wedge\wedge f$ where P is a parser and f is a function. $P \wedge\wedge f$ parses the same sentences as just P. Whenever P returns with some result R, the result of $P \wedge\wedge f$ is $f(R)$.

The “ $\wedge\wedge\wedge$ ” operator *replaces* the result of a parser. Expressions using this operator have the form $P \wedge\wedge\wedge v$ where P is a parser and v is a value. But

whenever P returns with some result R, the result of $P \wedge\wedge v$ is v instead of R. So “ $\wedge\wedge$ ” is related to “ \wedge ” by the equality

$$P \wedge\wedge v = P \wedge (r \Rightarrow v) .$$

As an example, here is the JSON parser that parses a numeric literal and converts it to a Scala integer:

```
numericLit  $\wedge$  (_.toInt)
```

And here is the JSON parser that parses the string "true" and returns Scala's true value:

```
"true"  $\wedge\wedge$  true
```

Now for more advanced transformations. Here's a new version of a parser for JSON objects that returns a Scala Map:

```
def obj: Parser[Map[String, Any]] =
  "{" ~ repsep(member, ",") ~ "}"  $\wedge\wedge$ 
  { case "{" ~ ms ~ "}" => Map() ++ ms }
```

Remember that the “ \sim ” operator produces as result an instance of a case class with the same name, “ \sim ”. This is no coincidence. It is designed that way so that you can match parser results with patterns that follow the same structure as the parsers themselves. For instance, the pattern “ ${"\sim ms \sim "}$ ” matches a result string “ ${$ ” followed by a result variable ms , which is followed in turn by a result string “ $}$ ”. This pattern corresponds exactly to what is returned by the parser on the left of the “ $\wedge\wedge$ ”. In its desugared versions where the “ \sim ” operator comes first, the same pattern reads $\sim(\sim("{", ms), ")")$ but this is much less legible.

The purpose of the pattern in the code above is to “strip off” the braces so you can get at the list of members resulting from the `repsep(member, ",")` parser. In cases like these there is also an alternative, which avoids producing the unnecessary parser results that are then discarded by the pattern match. The alternative makes use of the “ $\sim>$ ” and “ $<\sim$ ” parser combinators. Both express sequential composition just like “ \sim ”, but “ $\sim>$ ” keeps only the result of its right operand, whereas “ $<\sim$ ” keeps only the result of its left operand. So a shorter way to express the JSON object parser would be this:

```
def obj: Parser[Map[String, Any]] =
  "{" ~> repsep(member, ",") <~ "}" ^~ (Map() ++ _)
```

Here is a full JSON parser that returns meaningful results:

```
class JSON1 extends StandardTokenParsers {
  lexical.delimiters += ("{", "}", "[", "]", ":", ",")
  lexical.reserved += ("null", "true", "false")

  def obj: Parser[Map[String, Any]] =
    "{" ~> repsep(member, ",") <~ "}" ^~ (Map() ++ _)

  def arr: Parser[List[Any]] =
    "[" ~> repsep(value, ",") <~ "]"

  def member: Parser[(String, Any)] =
    stringLit ~ ":" ~ value ^^
      { case name ~ ":" ~ value => (name, value) }

  def value: Parser[Any] =
    obj | arr | stringLit | numericLit ^^ (_.toInt) |
    "null" ^^^ null | "true" ^^^ true | "false" ^^^ false
}
```

If you run this parser on the `address-book.json` file, you get the following result (after adding some newlines and indentation):

```
scala JSON1Test address-book.json
[14.1] parsed: Map(
  address book -> Map(
    name -> John Smith,
    address -> Map(
      street -> 10 Market Street,
      city -> San Francisco, CA,
      zip -> 94111),
    phone numbers -> List(408 338-4238, 408 111-6892)
  )
)
```

ident	identifier
keyword(...)	keyword or special symbol (implicit)
numericLit	integer number
stringLit	string literal
$P \sim Q$	sequential composition
$P \sim\sim Q, P \sim> Q$	sequential composition; keep left/right only
$P \mid Q$	alternative
opt(P)	option
rep(P)	repetition
repsep(P, Q)	interleaved repetition
$P \wedge\wedge f$	result conversion
$P \wedge\wedge\wedge v$	constant result

Table 28.1: Summary of parser combinators

Summary: Using Combinator Parsers

This is all you need to know in order to get started writing your own parsers. As an aide to memory, Table 28.1 lists the parser combinators that were discussed so far.

28.5 Implementing Combinator Parsers

The previous sections have shown that Scala’s combinator parsers provide a convenient means for constructing your own parsers. Since they are nothing more than a Scala library, they fit seamlessly into your Scala programs. So it’s very easy to combine a parser with some code that processes the results it delivers, or to rig a parser so that it takes its input from some specific source (say, a file, a string, or a character array).

How is this achieved? In the rest of this chapter you’ll take a look “under the hood” of the combinator parser library. You’ll see what a parser is, and how the primitive parsers and parser combinators encountered in previous sections are implemented. You can safely skip these parts if all you want is write some simple combinator parsers. On the other hand, reading the rest of this chapter should give you a deeper understanding of combinator parsers in particular, and of the design principles of a combinator DSL in general.

The core of Scala's combinator parsing framework is contained in a class `scala.util.parsing.combinator.Parsers`. This class defines the `Parser` type as well as all fundamental combinators. Except where stated explicitly otherwise, the definitions explained in the following two sub-sections all reside in this class. That is they are assumed to be contained in a class definition that starts as follows.

```
package scala.util.parsing.combinator
class Parsers {
    ... // code below goes here unless otherwise stated
}
```

The `StandardTokenParsers` class from which all previous example parsers inherited is itself a subclass of `Parsers`. `StandardTokenParsers` fixes some of the things that are left open in `Parsers`.

A `Parser` is in essence just a function from some input type to a parse result. As a first approximation, the type could be written as follows:

```
type Parser[T] = Input => ParseResult[T]
```

Parser Input

Here, the type of parser inputs is fixed by the definition:

```
type Input = Reader[Elem]
```

The class `Reader` comes from the package `scala.util.parsing.input`. It is similar to a `Stream`, but as mentioned previously, also keeps track of the positions of all the elements it reads. The type `Elem` represents individual input elements. It is an abstract type member of the `Parsers` class.

```
type Elem
```

This means that subclasses of `Parsers` need to instantiate class `Elem` to the type of input elements that are being parsed. For instance, the class `StandardTokenParsers` fixes `Elem` to be the Java-like word-tokens that you have encountered so far.

Parser Results

A parser might either succeed or fail on some given input. Consequently class ParseResult has two subclasses for representing success and failure:

```
abstract class ParseResult[+T]
case class Success[T](result: T, in: Input)
  extends ParseResult[T]
case class Failure(msg: String, in: Input)
  extends ParseResult[Nothing]
```

The Success case carries the result returned from the parser in its `result` parameter. The type of parser results is arbitrary; that's why Success, ParseResult, and Parser are all parameterized with a type parameter `T`, which represents the kinds of results returned by a given parser. Success also takes a second parameter, `in`, which refers to the input immediately following the part that the parser consumed. This field is needed for chaining parsers, so that one parser can operate after another one. Note that this is a purely functional approach to parsing. Input is not read as a side effect, but it is kept in a stream. A parser analyzes some part of the input streams, and then return the remaining part in its result.

The other subclass of ParseResult is Failure. This class takes as parameter a message that describes why the parser has failed. Like Success, Failure also takes the remaining input stream as a second parameter. This is needed not for chaining (the parser won't continue after a failure), but to position the error message at the correct place in the input stream.

Note that parse results are defined to be covariant in the type parameter `T`. That is, a parser returning `Strings` as result, say, is compatible with a parser returning `Objects`.

The Parser class

In fact, the previous characterization of parsers as functions from inputs to parse result was oversimplified a bit. The examples above have shown that parsers also implement *methods* such as “`~`” for sequential composition of two parsers and “`|`” for their alternative composition. So Parser is in reality a class that inherits from the function type `Input => ParseResult[T]` and that additionally defines these methods:

```
abstract class Parser[+T] extends (Input => ParseResult[T])
{ p =>
  /** An unspecified method that defines
   * the behavior of this parser: */
  def apply(in: Input): ParseResult[T]

  def ~ ...
  def | ...
  ...
}
```

Since parsers are (*i.e.* inherit from) functions, they need to define an `apply` method. You see an abstract `apply` method in class `Parser`, but this is just for documentation, as the same method is in any case inherited from the parent type `Input => ParseResult[T]` (recall that this type is an abbreviation for `scala.Function1[Input, ParseResult[T]]`). The `apply` method still needs to be implemented in the individual parsers that inherit from the abstract `Parser` class. These parsers will be discussed after the following section.

Aliasing this

The body of the `Parser` class above starts with a curious expression:

```
abstract class Parser[+T] extends ... { p =>
```

A clause such as `id =>` immediately after the opening brace of a class template defines the identifier `id` as an alias for `this` in the class. It's as if you had written

```
val id = this
```

in the class body, except that the Scala compiler knows that `id` is an alias for `this`. For instance, you could access an object-private member `m` of the class using either `id.m` or `this.m`; the two are completely equivalent. The first expression would not compile if `id` was just defined as a `val` with `this` as its right hand side, because in that case the Scala compiler would treat `id` as a normal identifier.

Aliasing can be a good abbreviation when you need to access the `this` of an outer class. Here's an example:

```
class Outer { outer =>
    class Inner {
        println(Outer.this eq outer) // prints: true
    }
}
```

The example defines two nested classes, Outer and Inner. Inside Inner the this value of the Outer class is referred to two times, using different expressions. The first expression shows the Java way of doing things: You can prefix the reserved word this with the name of an outer class and a period; such an expression then refers to the this of the outer class. The second expression shows the alternative that Scala gives you: By introducing an alias named outer for this in class Outer, you can refer to this alias directly also in inner classes. The Scala way is more concise, and can also improve clarity, if you choose the name of the alias well. You'll see examples of this in [pages 518](#) and [519](#).

Single-Token Parsers

Class `Parsers` defines a generic parser `elem` that can be used to parse any single token:

```
def elem(kind: String, p: Elem => Boolean) =
  new Parser[Elem] {
    def apply(in: Input) =
      if (p(in.first)) Success(in.first, in.rest)
      else Failure(kind+" expected", in)
  }
```

This parser takes two parameters: A kind string describing what kind of token should be parsed, and a predicate `p` on `Elements` which indicates whether an element fits the class of tokens to be parsed.

When applying the parser `elem(kind, p)` to some input `in`, the first element of the input stream is tested with predicate `p`. If `p` returns `true`, the parser succeeds. Its result is the element itself, and its remaining input is the input stream starting just after the element that was parsed. On the other hand, if `p` returns `false`, the parser fails with an error message that indicates what kind of token was expected.

Sequential Composition

The `elem` parser only consumes a single element. To parse more interesting phrases, you can string parsers together with the sequential composition operator “ \sim ”. As you have seen before, $P \sim Q$ is a parser that applies first the P parser to a given input string. Then, if P succeeds, the Q parser is applied to the input that’s left after P has done its job.

The “ \sim ” combinator is implemented as a method in class `Parser`. Here is its definition:

```
abstract class Parser[+T] ... { p =>
  ...
  def ~ [U](q: => Parser[U]) = new Parser[T ~ U] {
    def apply(in: Input) = p(in) match {
      case Success(x, in1) =>
        q(in1) match {
          case Success(y, in2) => Success(new ~(x, y), in2)
          case failure => failure
        }
      case failure => failure
    }
  }
}
```

Let’s analyze this method in detail. It is a member of the `Parser` class. Inside this class, `p` is specified by the `p =>` part as an alias of `this`, so `p` designates the left-hand argument (or: receiver) of “ \sim ”. Its right-hand argument is represented by parameter `q`. Now, if $p \sim q$ is run on some input `in`, first `p` is run on `in` and the result is analyzed in a pattern match. If `p` succeeds, `q` is run on the remaining input `in1`. If `q` also succeeds, the parser as a whole succeeds. Its result is a “ \sim ”-object containing both `x`, the result of `p`, and `y`, the result of `y`. On the other hand, if either `p` or `q` fails the result of $p \sim q$ is the `Failure` object returned by `p` or `q`.

The other two sequential composition operators “ $<\sim$ ” and “ $\sim>$ ” can be defined just like “ \sim ”, with some small adjustment how the result is computed. But a more elegant technique is to define them *in terms* of “ \sim ” as follows:

```
def <~[U](q: => Parser[U]): Parser[T] =
```

```
(p ~ q) ^~ { case x ~ y => x }
def ~>[U](q: => Parser[U]): Parser[U] =
  (p ~ q) ^~ { case x ~ y => y }
```

Alternative Composition

An alternative composition $P \mid Q$ applies either P or Q to a given input. It first tries P . If P succeeds, the whole parser succeeds with the result of P . Otherwise, if P fails, then Q is tried *on the same input* as P . The result of Q is then the result of the whole parser.

Here is a definition of “ \mid ” as a method of class `Parser`.

```
def | (q: => Parser[T]) = new Parser[T] {
  def apply(in: Input) = p(in) match {
    case s1 @ Success(_, _) => s1
    case failure => q(in)
  }
}
```

Dealing with Recursion

Note that the q parameter in methods “ \sim ” and “ \mid ” is specified to be call-by-name. This means that the actual parser argument will be evaluated only when q is needed; and that is the case only after p has run. This makes it possible to write recursive parsers like the following one which parses a number enclosed by arbitrarily many parentheses:

```
def parens = numericLit | "(" ~ parens ~ ")"
```

If “ \mid ” and “ \sim ” had taken call-by-value parameters, this definition would immediately cause a stack overflow without reading anything, because the value of `parens` occurs in the middle of its right-hand side.

Result Conversion

The last two methods of class `Parser` convert a parser’s result. The parser forms $P \wedge f$ and $P \wedge\wedge v$ both succeed exactly when P succeeds but they

change its result. $P \wedge\wedge f$ transforms P's result by applying the function f to it. By contrast, $P \wedge\wedge\wedge v$ replaces P's result with the value v.

```
def ^~ [U](f: T => U): Parser[U] = new Parser[U] {
    def apply(in: Input) = p(in) match {
        case Success(x, in1) => Success(f(x), in1)
        case failure => failure
    }
}
def ^~~~ [U](v: U): Parser[U] = f ^~ (x => v)
} // end Parser
```

Parsers that don't read any input

There are also two parsers that do not consume any input. The parser `success(result)` always succeeds with the given `result`. The parser `failure(msg)` always fails with error message `msg`. Both are implemented as methods in class `Parsers`, the outer class that also contains class `Parser`:

```
def success[T](v: T) = new Parser[T] {
    def apply(in: Input) = Success(v, in)
}
def failure(msg: String) = new Parser[Nothing] {
    def apply(in: Input) = Failure(msg, in)
}
```

Option and repetition

Also defined in class `Parsers` are the option and repetition combinators `opt`, `rep`, and `repsep`. They are all implemented in terms of sequential composition, alternative, and result conversion:

```
def opt[T](p: => Parser[T]): Parser[Option[T]] = (
    p ^~ Some(_)
    | success(None)
)
def rep[T](p: Parser[T]): Parser[List[T]] = (
```

```

    p ~ rep(p) ^^ { case x ~ xs => x :: xs }
| success(List())
)

def repsep[T, U](p: Parser[T], q: Parser[U]): Parser[List[T]] = (
    p ~ rep(q ~> p) ^^ { case r ~ rs => r :: rs }
| success(List())
)

} // end Parsers

```

Aside: Note that the body of each of the three parsers above is enclosed in parentheses. This is a little trick to disable semicolon inference in parser expressions. You have seen in [Section 4.7](#) that Scala assumes there's a semi-colon between any two lines that can be separate statements syntactically, unless the first line ends in an infix operator, or the two lines are enclosed in parentheses or square brackets. Now, you could have written the ‘|’ operator at the end of the first alternative instead of at the beginning of the second, like this:

```

def opt[T](p: => Parser[T]): Parser[Option[T]] =
    p ^^ Some(_) |
    success(None)

```

In that case, no parentheses around the body of function `opt` are required. However, some people prefer to see the ‘|’ operator at the beginning of the second alternative rather than at the end of the first. Normally, this would lead to an unwanted semicolon between the two lines, like this:

```

p ^^ Some(_); // semicolon implicitly inserted
| Success

```

The semicolon changes the structure of the code, causing it to fail compilation. Putting the whole expression in parentheses avoids the semicolon and makes the code compile correctly.

This concludes the description of the general combinator parser framework. In the next sections, you'll find out how this framework is adapted to yield the kind of standard token parsers used by the examples at the beginning of this chapter.

28.6 Lexing and Parsing

The task of syntax analysis is usually split into two phases. The *lexer* phase recognizes individual words in the input and classifies them into some *token* classes. This phase is also called *lexical analysis*. This is followed by a *syntactical analysis* phase that analyzes sequences of tokens. Syntactical analysis is also sometimes just called parsing, even though this is slightly imprecise, as lexical analysis can also be regarded as a parsing problem.

The `Parsers` class as described above can be used for either phase, because its input elements are of the abstract type `Elem`. For lexical analysis, `Elem` would be instantiated to `Char`, meaning that what's parsed are the individual characters that make up a word. The syntactical analyzer would in turn instantiate `Elem` to the type of `Tokens` returned by the lexer.

Scala's parsing combinators provide several utility classes for lexing and syntactic analysis. These are contained in two sub-packages, one each for lexical and syntactical analysis.

```
scala.util.parsing.combinator.lexical  
scala.util.parsing.combinator.syntactical
```

In the following, you will learn of some of the abstractions in the syntactical analysis package, just enough to understand how standard token parsers work. These parsers use a standard lexical analysis that distinguishes a subset of the tokens of Java and Scala. If you need to write a lexical analyzer that follows different rules, you should consult the scaladoc API documentation for the `lexical` sub-package.

The tokens that are supported by the standard lexer are described by the following class:

```
abstract class Token { def chars: String }
```

Here, the `chars` method returns the characters making up the token as a `String`. Class `Token` has four standard subclasses:

```
case class Keyword (override val chars: String) ...  
case class NumericLit(override val chars: String) ...  
case class StringLit (override val chars: String) ...
```

```
case class Identifier(override val chars: String)...
```

These represent keywords, numbers, strings, and identifiers, respectively.

28.7 Standard Token Parsers

The `StandardTokenParsers` class is found in the syntactical analysis package. It extends class `Parsers`, fixing the kind `Elem` of input elements to be instances of class `Token`. A slightly simplified account of the class is given below (in reality the contents of the class are spread over several parent classes, to enable more flexible re-use):

```
package scala.util.parsing.combinator.syntactical
class StandardTokenParsers extends Parsers {
    type Elem = Token
    ...
}
```

The `StandardTokenParsers` class also defines four single-token parsers for the four kinds of tokens that are supported. Each of these is defined in terms of `elem`:

```
...
/** A parser that matches a numeric literal */
def numericLit: Parser[String] =
    elem("number", _.asInstanceOf[NumericLit]) ^^ (_.chars)

/** A parser that matches a string literal */
def stringLit: Parser[String] =
    elem("string", _.asInstanceOf[StringLit]) ^^ (_.chars)

/** A parser that matches an identifier */
def ident: Parser[String] =
    elem("identifier", _.asInstanceOf[Identifier]) ^^ (_.chars)

/** A parser matching a given reserved word or delimiter */
implicit def keyword(chars: String): Parser[String] =
    elem("'" + chars + "'", _ == Keyword(chars))

} // end StandardTokenParsers
```

The numericLit parser succeeds if the first input token is a numeric literal of type NumericLit; in that case it returns the characters making up the literal as a string. Analogously, the ident and stringLit parsers accept identifiers and string literals.

The last of the four parsers is keyword. This parser accepts a given reserved word or delimiter. For instance keyword("+) succeeds if the first token is a "+" and fails otherwise. Or keyword("true") succeeds if the first token is the reserved word true and fails otherwise.

One peculiarity of the keyword method is that it carries an implicit modifier. This means that the keyword method is applied implicitly to an expression e whenever e is a string and the expected type of the expression is a Parser. In that case, the Scala compiler expands e to keyword(e). That's why you could simply write strings in place of parsers in the examples at the beginning of this chapter. For instance, the JSON parser term for an object member stringLit ~ ":" ~ value really means stringLit ~ keyword(":") ~ value.

28.8 Error reporting

There's one final topic that was not covered yet: How does the parser issue an error message? Error reporting for parsers is somewhat of a black art. One problem is that a parser that rejects some input contains many different failures. After all each alternative parse must have failed, and so on recursively for each choice point. Which of the usually numerous failures should be emitted as error message to the user?

Scala's parsing library implements a simple heuristic: Among all failures, the one that occurred at the latest position in the input is chosen. In other words, the parser picks the longest prefix that is still valid and then issues an error message that describes why parsing the prefix could not be continued further. If there are several failure points at that latest position, the one that was visited last is chosen.

For instance, consider running the JSON parser on a faulty address book which starts with the line

```
{ "name": John,
```

The longest legal prefix of this phrase is { "name": . So the JSON parser

will flag the word John as an error. The JSON parser expects a value at this point but John is an identifier, which does not count as a value (presumably, the author of the document had forgotten to enclose the name in quotation marks). The error message issued by the parser for this document is:

```
[1.13] failure: "false" expected but identifier John found  
{ "name": John,  
  ^
```

The part that “false” was expected comes from the fact that "false" is the last alternative of the production for value in the JSON grammar. So this was the last failure at this point. If one knows the JSON grammar in detail, one can reconstruct the error message, but for a non-expert this error message is probably surprising and can also be quite misleading.

A better error message can be engineered by adding a “catch all” failure point as last alternative of a value production:

```
def value : Parser[Any] = obj | arr | stringLit | numericLit |  
  "null" | "true" | "false" |  
  failure("illegal start of value")
```

This addition does not change the set of inputs that are accepted as valid documents. But it does improve the error messages, because now it will be the explicitly added failure that comes as last alternative and therefore gets reported:

```
[1.13] failure: illegal start of value  
{ "name": John,  
  ^
```

The implementation of the “latest possible” scheme of error reporting uses a field

```
var lastFailure: Option[Failure] = None
```

in class `Parsers` to mark the failure that occurred at the latest position in the input. The field is initialized to `None`. It is updated in the constructor of class `Failure`:

```

case class Failure(msg: String, in: Input)
extends ParseResult[Nothing] {
    if (lastFailure.isDefined &&
        lastFailure.get.in.pos <= in.pos)
        lastFailure = Some(this)
}

```

The field is read by the `phrase` method, which emits the final error message if the parser failed. Here is the implementation of `phrase` in class `Parsers`:

```

def phrase[T](p: Parser[T]) = new Parser[T] {
    lastFailure = None
    def apply(in: Input) = p(in) match {
        case s @ Success(out, in1) =>
            if (in1.atEnd) s
            else Failure("end of input expected", in1)
        case f : Failure =>
            lastFailure
    }
}

```

The method runs its argument parser `p`. If `p` succeeds with a completely consumed input, the success result of `p` is returned. If `p` succeeds but the input is not read completely, a failure with message “end of input expected” is returned. If `p` fails, the failure or error stored in `lastFailure` is returned. Note that the treatment of `lastFailure` is non-functional; it is updated as a side-effect by the constructor of `Failure` and the `phrase` method itself. A functional version of the same scheme would be possible, but it would require “threading” the `lastFailure` value through every parser result, no matter whether this result is a `Success` or a `Failure`.

28.9 Backtracking vs LL(1)

The parser combinators employ *backtracking* to choose between different parsers in an alternative. In an expression `P | Q`, if `P` fails, then `Q` is run on the same input as `P`. This happens even if `P` has parsed some tokens before failing. In this case the same tokens will be parsed again by `Q`.

Backtracking imposes only few restrictions on how to formulate a grammar so that it can be parsed. Essentially, you just need to avoid left-recursive productions. A production such as

```
expr ::= expr "+" term | term
```

will always fail because `expr` immediately calls itself and thus never progresses any further.¹ On the other hand, backtracking is potentially costly because the same input can be parsed several times. Consider for instance the production

```
expr ::= term "+" expr | term
```

What happens if `expr` parser is applied to an input such as $(1 + 2) * 3$ which constitutes a legal term? The first alternative would be tried, and would fail when matching the “+” sign. Then the second alternative would be tried on the same term and this would succeed. In the end the term ended up being parsed twice.

It is often possible to modify the grammar so that backtracking is avoided. For instance, in the case of arithmetic expressions, either one of the following productions would work:

```
expr ::= term ["+" expr]
```

```
expr ::= term {"+" term}
```

Many languages admit so-called “LL(1)” grammars. When a combinator parser is formed from such a grammar, it will never do backtracking (*i.e.* the input position is never reset to an earlier value, provided the parser input is correct).

For instance, the grammars for arithmetic expressions and JSON terms earlier in this chapter are both LL(1), so the backtracking capabilities of the parser combinator framework are never exercised for inputs from these languages.

The combinator parsing framework allows you to express the expectation that a grammar is LL(1) explicitly, using a new operator $\sim!$. This operator is

¹There are ways to avoid stack overflows even in the presence of left-recursion, but this requires a more refined parsing combinator framework, which to date has not been implemented.

like sequential composition \sim but it will never backtrack to “un-read” input elements that have already been parsed. Using this operator, the productions in the arithmetic expression parser could alternatively be written as follows:

```
def expr  : Parser[Any] =
  term ~! rep("+" ~! term | "-" ~! term)
def term  : Parser[Any] =
  factor ~! rep("*" ~! factor | "/" ~! factor)
def factor: Parser[Any] =
  "(" ~! expr ~! ")" | numericLit
```

One advantage of an LL(1) parser is that it can use a simpler input technique. Input can be read sequentially, and input elements can be discarded once they are read. That’s another reason why LL(1) parsers are usually more efficient than backtracking parsers.

28.10 Conclusion

You have now seen all the essential elements of Scala’s combinator parsing framework. It’s surprisingly little code for something that’s genuinely useful. With the framework you can construct parsers for a large class of context-free grammars. The framework lets you get started quickly but it is also customizable to new kinds of grammars and input methods. Being a Scala library, it integrates seamlessly with the rest of the language. So it’s easy to integrate a combinator parser in a larger Scala program.

One downside of combinator parsers is that they are not very efficient, at least not when compared with parsers generated from special purpose tools such as Yacc or Bison. This has to do with two effects: First, the backtracking method used by combinator parsing is itself not very efficient. Depending on the grammar and the parse input, it might yield an exponential slow-down due to repeated backtracking. This can be fixed by making the grammar LL(1) and by the committed sequential composition operator “ $\sim!$ ”.

The second problem affecting the performance of combinator parsers is that they mix parser construction and input analysis in the same set of operations. In effect, a parser is generated anew for each input that’s parsed.

This problem can be overcome, but it requires a different implementation of the parser combinator framework. In an optimizing framework, a parser

would no longer be represented as a function from inputs to parse results. Instead, it would be represented as a tree, where every construction step was represented as a case class. For instance, sequential composition could be represented by a case class Seq, alternative by Alt and so on. The “outermost” parser method phrase could then take this symbolic representation of a parser and convert it to highly efficient parsing tables, using standard parser generator algorithms.

What’s nice about all this is that from a user perspective nothing changes compared to plain combinator parsers. Users still write parsers in terms of ident, numericLit, “~”, “|” and so on. They need not be aware of the fact that these methods generate a symbolic representation of a parser instead of a parser function. Since the phrase combinator converts these representations into real parsers, everything works as before.

The advantage of this scheme with respect to performance are two-fold. First, you can now factor out parser construction from input analysis. If you write

```
val jsonParser = phrase(value)
```

and then apply jsonParser to several different inputs, the jsonParser is constructed only once, not every time an input is read.

Second, the parser generation can use efficient parsing algorithms such as LALR(1). These algorithms usually lead to much faster parsers than parsers that operate with backtracking.

At present, such an optimizing parser generator has not yet been written for Scala. But it would be perfectly possible to do so. If someone contributes such a generator, it will be easy to integrate into the standard Scala library.

Even postulating that such a generator will exist at some point in the future, there remain still reasons for also keeping the current parser combinator framework around because it is much easier to understand and to adapt than a parser generator. Furthermore, the difference in speed would often not matter in practice, unless you want to parse very large inputs.

Glossary

algebraic data type A type defined by giving several alternatives, each of which comes with its own constructor. It usually comes with a way to decompose the type through pattern matching. The concept is found in specification languages and functional programming languages. Algebraic data types can be emulated in Scala with case classes.

alternative An *alternative* is a branch of a match expression. It has the form “*case pattern => expression*.” Another name for alternative is, simply, *case*.

annotation An annotation appears in source code and is attached to some part of the syntax. Annotations are computer processable, so you can use them to effectively add an extension to Scala.

anonymous function Another name for function literal.

apply You can *apply* a method, function, or closure *to* arguments, which means you invoke it on those arguments.

argument When a function is invoked, an *argument* is passed for each parameter of that function. The parameter is the variable that refers to the argument. The argument is the object passed at invocation time. In addition, applications can take (command line) arguments that show up in the `Array[String]` passed to `main` methods of singleton objects.

assign You can *assign* an object *to* a variable. Afterwards, the variable will refer to the object.

auxiliary constructor Extra constructors defined inside the curly braces of the class definition, which look like method definitions named `this`, but with no result type.

block A *block* is one or more statements in Scala source code, usually surrounded by curly braces. Blocks are commonly used as the bodies of functions, for expressions, while loops, and any other places where you want to group a number of statements together. More formally, a block is an encapsulation construct for which you can only see side effects and a result value. The curly braces in which you define a class or object do not, therefore, form a block, because fields and methods (which are defined inside those curly braces) are visible from the outside. Such curly braces form a *template*.

bound variable A *bound variable* of an expression is a variable that's both used and defined inside the expression. For instance, in the function literal expression `(x: Int) => (x, y)`, both variables `x` and `y` are used, but only `x` is bound, because it is defined in the expression as an `Int` and the sole argument to the function described by the expression.

by-name parameter A parameter that is marked with a '`=>`' in front of the parameter type, *e.g.* `(x: => Int)`. The argument corresponding to a by-name parameter is evaluated not before the method is invoked, but each time the parameter is referenced *by name* inside the method.

class A *class* is defined with the `class` keyword. A class may either be abstract or concrete. A class may be parameterized with types and values when instantiated. In “`new Array[String](2)`,” the class being instantiated is `Array` and the type of the value that results is `Array[String]`. A class that takes type parameters is called a *type constructor*. A type can be said to have a class as well, as in: the class of type `Array[String]` is `Array`.

closure A *closure* is a function object that captures free variables, and is said to be “closed” over the variables visible at the time it is created.

companion class A class that shares the same name with a singleton object defined in the same source file. The class is the singleton object’s companion class.

companion object A singleton object that shares the same name with a class defined in the same source file. Companion objects and classes have access to each other's private members. In addition, any implicit conversions defined in the companion object will be in scope anywhere the class is used.

currying *Currying* is a way to write functions with multiple parameter lists.

For instance `def f(x: Int)(y: Int)` is a curried function with two parameter lists. A curried function is applied by passing several arguments lists, as in: `f(3)(4)`. However, it is also possible to write a *partial application* of a curried function, such as `f(3)`.

declare You can *declare* an abstract field, method, or type, which gives an entity a name but not an implementation. The key difference between declarations and definitions is that definitions establish an implementation for the named entity, declarations do not.

define To *define* something in a Scala program is to give it a name and an implementation. You can define classes, traits, singleton objects, fields, methods, local functions, local variables, etc. Because definitions always involve some kind of implementation, abstract members are *declared* not defined.

direct subclass A class is a *direct subclass* of its direct superclass.

direct superclass A class's *direct superclass* is the class from which it is immediately derived, the nearest class above it in its inheritance hierarchy. If a class `Parent` is mentioned in a class `Child`'s optional extends clause, then `Parent` is the direct superclass of `Child`. If a trait is mentioned in a `Child`'s extends clause, or `Child` has no extends clause, then `AnyRef` is the direct superclass of `Child`. If a class's direct superclass takes type parameters, for example `class Child extends Parent[String]`, the direct superclass of `Child` is still `Parent`, not `Parent[String]`. On the other hand, `Parent[String]` would be a direct *supertype* of `Child`. See supertype for more discussion of the distinction between class and type.)

equality When used without qualification, *equality* is the relation between values expressed by '`==`'. See also *reference equality*.

existential type todo

expression An *expression* is any bit of Scala code that yields a result. You can also say that an expression *evaluates to* or *results in* a result.

filter A *filter* is an `if` followed by a boolean expression in a `for` expression. For example, in `for(i <- 1 to 10; if i % 2 == 0)`, the filter is “`if i % 2 == 0`. ” The value to the right of the `if` is the *filter expression*.

filter expression A *filter expression* is the boolean expression following an `if` in a `for` expression. For example, in `for(i <- 1 to 10; if i % 2 == 0)`, the filter expression is “`i % 2 == 0`. ”

first-class function Scala supports *first-class functions*, which means you can express functions in *function literal* syntax, such as `(x: Int) => x + 1`, and that functions can be represented by objects, called *function values*.

free variable A *free variable* of an expression is a variable that’s used inside the expression but that is not defined inside the expression. For instance, in the function literal expression `(x: Int) => (x, y)`, both variables `x` and `y` are used, but only `y` is free, because it is not defined in the expression.

function A *function* can be *invoked* with a list of arguments to produce a result. A function has a parameter list, a body, and a result type. Functions that are members of a class, trait, or singleton object are called *methods*. Functions defined inside other functions are called *local functions*. Methods with the result type of `Unit` are called *procedures*. A function created with function literal syntax is called a *function value*.

function literal A function with no name in Scala source code, specified with function literal syntax. For example, `(x: Int, y: Int) => x + y`.

function value A *function value* is a function object and can be invoked just like any other function. A function value’s class extends one of the `FunctionN` traits (*e.g.*, `Function0`, `Function1`) from package `scala`,

and is usually expressed in source code via *function literal* syntax. A function value is “invoked” when its `apply` method is called. A function value that captures free variables is a *closure*.

functional style The *functional style* of programming is characterized by passing function values into looping methods, immutable data, methods with no side effects. It is the dominant paradigm of languages such as Haskell and Erlang, and contrasts with the *imperative style*.

generator A *generator* defines a named `val` and assigns to it a series of values in a `for` expression. For example, in `for(i <- 1 to 10)`, the generator is “`i <- 1 to 10`.`”` The value to the right of the `<-` is the *generator expression*.

generator expression A *generator expression* generates a series of values in a `for` expression. For example, in `for(i <- 1 to 10)`, the generator expression is “`1 to 10`.`”`

generic class A *generic class* is a class that takes type parameters. For example, because `scala.List` takes a type parameter, `scala.List` is a generic class.

generic trait A *generic trait* is a trait that takes type parameters. For example, because `scala.collection.Set` takes a type parameter, `scala.collection.Set` is a generic trait.

helper method A *helper method* is a method whose purpose is to provide a service to one or more other methods nearby. Helper methods are often implemented as local functions.

immutable An object is *immutable* if its value cannot be changed after it is created in any way visible to clients. Objects may or may not be immutable.

imperative style The *imperative style* of programming is characterized by iteration with loops, mutating data in place, and methods with side effects. It is the dominant paradigm of languages such as C, C++, C# and Java, and contrasts with the *functional style*.

initialize When a variable is defined in Scala source code, you must *initialize* it with an object.

instance An *instance*, or class instance, is an object, a concept that exists only at runtime.

instantiate To *instantiate* a class is to make a new object from a class blueprint, an action that happens only at runtime.

invoke You can *invoke* a method, function, or closure *on* arguments, meaning its body will be executed with the specified arguments.

JVM The *JVM* is the Java Virtual Machine, or *runtime*, that hosts a running Scala program.

literal 1, "One", and `(x: Int) => x + 1` are examples of *literals*. A literal is a shorthand way to describe an object, where the shorthand exactly mirrors the structure of the created object.

local variable A *local variable* is a `val` or `var` defined inside a block. Although similar to local variables, parameters to functions are not referred to as local variables, but simply as parameters or “variables” without the “local.”

member A *member* is any named element of the template of a class, trait, or singleton object. A member may be accessed with the name of its owner, a dot, and its simple name. For example, top-level fields and methods defined in a class are members of that class. A trait defined inside a class is a member of its enclosing class. A type defined with the `type` keyword in a class is a member of that class. A class is a member of the package in which it is defined. By contrast, a local variable or local function is not a member of its surrounding block.

meta-programming Meta-programming software is software whose input is itself software. Compilers are meta-programs, as are tools like ScalaDoc. Meta-programming software is required in order to do anything with an [annotation](#).

method A *method* is a function that is a member of some class, trait, or singleton object.

Mixin *Mixin* is what a trait is called when it is being used in a mixin composition. In other words, in “trait Hat,” Hat is just a trait, but in “new Cat extends AnyRef with Hat,” Hat can be called a mixin. When used as a verb, “mix in” is two words. For example, you can mix traits into classes or other traits.

Mixin composition *Mixin composition* is the process of mixing traits into classes or other traits. Mixin composition differs from multiple inheritance in that the type of the super reference is not known at the point the trait is defined, but rather is determined anew each time the trait is mixed into a class or other trait.

modifier A *modifier* is a keyword that qualifies a class, trait, field, or method definition in some way. For example, the `private` modifier indicates that a class, trait, field, or method being defined is private.

multiple definitions The same expression can be assigned in multiple definitions if you use the syntax `val v1, v2, v3 = exp.`

operation In Scala, every *operation* is a method call. Methods may be invoked in *operator notation*, such as `b + 2`, and when in that notation, `+` is an *operator*.

parameter Functions may take zero to many *parameters*. Each parameter has a name and a type. The difference between parameters and arguments is that arguments refer to the actual objects passed when a function is invoked. Parameters are the variables that refer to those passed arguments.

parameterless function A *parameterless function* is a function that takes no parameters, and is defined without any empty parentheses. Invocations of parameterless functions may not supply parentheses. This supports the *uniform access principle*, which enables the `def` to be changed into a `val` without requiring a change to client code.

parametric field A *parametric field* is a field defined as a class parameter.

partially applied function A *partially applied function* is a function that's used in an expression and that misses some of its arguments. For instance, if function `f` has type `Int => Int => Int`, then `f` as well as `f(1)` are partially applied functions.

pattern In a match expression alternative, a *pattern* follows each case keyword and precedes either a *pattern guard* or the `=>` symbol.

pattern guard In a match expression alternative, a *pattern guard* can follow a *pattern*. For example, in “`case x if x % 2 == 0 => x + 1,`” the pattern guard is “`if x % 2 == 0.`” The case with a pattern guard will only be selected if the pattern matches and the pattern guard yields true.

predicate A *predicate* is a first-class function with a result type of Boolean.

primary constructor The main constructor of a class, which invokes a superclass constructor, if necessary, initializes fields for any value parameters not passed to the superclass constructor to passed values, except any that are not used in the body of the class and can therefore be optimized away, and executes any top-level code defined in between the curly braces of the class, but outside any field and method definitions.

procedure A *procedure* is a method with result type `Unit`, which would therefore be executed for its side effects.

reassignable A variable may or may not be *reassignable*. A `var` is reassignable while a `val` is not.

recursive A function is *recursive* if it calls itself. If the only place the function calls itself is the last expression of the function, then the function is *tail recursive*.

reference A *reference* is the Java abstraction of a pointer, which uniquely identifies an object that resides on the JVM's heap. Reference type variables hold references to objects, because reference types (instances of `AnyRef`) are implemented as Java objects that reside on the JVM's heap. Value type variables, by contrast, may sometimes hold a reference (to a boxed wrapper type) and sometimes not (when the object

is being represented as a primitive value). Speaking generally, a Scala variable *refers* to an object. The term “refers” is more abstract than “holds a reference.” If a variable of type `scala.Int` is currently represented as a primitive Java `int` value, then that variable still refers to the `Int` object, but no reference is involved.

reference equality *Reference equality* means that two references identify the very same Java object. Reference equality can be determined, for reference types only, by calling `eq` in `AnyRef`. (In Java programs, reference equality can be determined using `==` on Java reference types.)

reference type A *reference type* is a subclass of `AnyRef`. Instances of reference types always reside on the JVM’s heap at runtime.

refers A variable in a running Scala program always *refers* to some object. Even if that variable is assigned to `null`, it conceptually refers to the `Null` object. At runtime, an object may be implemented by a Java object or a value of a primitive type, but Scala allows programmers to think at a higher level of abstraction about their code as they imagine it running. See also *reference*.

result An expression in a Scala program yields a *result*. The result of every expression in Scala is an object.

result type A method’s *result type* is the type of the value that results from calling the method. (In Java, this concept is called the return type.)

return A function in a Scala program *returns* a value. You can call this value the *result* of the function. You can also say the function *results in* the value. The result of every function in Scala is an object.

runtime The *runtime* is the Java Virtual Machine, or JVM, that hosts a running Scala program. Runtime encompasses both the virtual machine, as defined by the Java Virtual Machine Specification, and the runtime libraries of the Java API and the standard Scala API. The phrase *at runtime* means when the program is running, and contrasts with compile time.

runtime type A *runtime type* is the type of an object at runtime. To contrast, a *static type* is the type of an expression at compile time. Most

runtime types are simply bare classes with no type parameters. For example, the runtime type of "Hi" is `String`, and the runtime type of `(x: Int) => x+1` is `Function1`. Runtime types can be tested with the `isInstanceOf` method.

script A file containing top level statements and definitions, which can be run directly with `scala` without explicitly compiling. A script must end in an expression, not a definition.

selector A *selector* is the value being matched on in a match expression. For example, in "`s match { case _ => }`", the selector is `s`.

self type A self type of a trait is the assumed type of `this`, the receiver, to be used within the trait. Any concrete class that mixes in the trait must ensure that its type conforms to the trait's self type. The most common use of self types is for dividing a large class into several traits as described in [Chapter 25](#).

semi-structured data XML data is semi-structured. It is more structured than a flat binary file or text file, but it does not have the full structure of a programming language's data structures. text

serialization You can serialize an object into a byte stream which can then be saved to files or transmitted over the network. You can later deserialize the byte stream, even on different computer, and obtain an object that is the same as the original serialized object.

shadow A new declaration of a local variable *shadows* one of the same name in an enclosing scope.

signature A function's *signature* comprises its name, the number, order, and types of its parameters, if any, and its result type.

singleton object A *singleton object* is an object defined with the `object` keyword. A singleton object has one and only one instance. A singleton object that shares its name with a class, and defined in the same source file as that class, is that class's *companion object*. The class is its *companion class*. A singleton object that doesn't have a companion class is a *standalone object*.

standalone object A *standalone object* is a singleton object that has no companion class.

statement A *statement* is a bit of Scala code that is executed for its side-effects.

static type See *type*.

subclass A class is a *subclass* of all of its superclasses and supertraits.

subtrait A trait is a *subtrait* of all of its supertraits.

subtype The Scala compiler will allow any of a type's *subtypes* to be used as a substitute wherever that type is required. For classes and traits that take no type parameters, the subtype relationship mirrors the subclass relationship. For example, if class Cat is a subclass of abstract class Animal, and neither takes type parameters, type Cat is a subtype of type Animal. Likewise, if trait Apple is a subtrait of trait Fruit, and neither takes type parameters, type Apple is a subtype of type Fruit. For classes and traits that take type parameters, however, variance comes into play. For example, because abstract class List is declared to be covariant in its lone type parameter (*i.e.*, List is declared List[+A]), List[Cat] is a subtype of List[Animal], and List[Apple] a subtype of List[Fruit]. These subtype relationships exist even though the class of each of these types is List. By contrast, because Set is not declared to be covariant in its type parameter (*i.e.*, Set is declared Set[A] with no plus sign), Set[Cat] is *not* a subtype of Set[Animal]. The term "subtype" does not imply that a class or trait correctly implements the contracts of its supertypes, which is required for the Liskov substitution principle to work, just that the compiler will allow such substitution of the type where a supertype is required. If a class does not correctly implement the contract of its direct superclass, for example, it can be said that the class is not a *valid subclass* of its superclass. It is still a subclass, just not a "valid" one, of its superclass.

superclass A class's *superclasses* includes its direct superclass, its direct superclass's direct superclass, and so on, all the way up to Any.

supertrait A class's or trait's *supertraits*, if any, include all traits directly mixed into the class or trait or any of its superclasses, plus any supertraits of those traits.

supertype A type is a *supertype* of all of its subtypes.

synthetic class A *synthetic class* is generated automatically by the compiler rather than being written by hand by the programmer.

tail recursive A function is *tail recursive* if the only place the function calls itself is the last operation of the function.

target typing *Target typing* is a form of type inference that takes into account the type that's expected. For example, in `nums.filter((x) => x > 0)`, the Scala compiler infers type of x to be the element type of nums, because the filter method invokes the function on each element of nums.

template A *template* is the body of a class, trait, or singleton object definition. It defines the interface, behavior and initial state of the class, trait, or object.

trait A *trait*, which is defined with the trait keyword, is like an abstract class that cannot take any value parameters and can be "mixed into" classes or other traits via the process known as *mixin composition*. When a trait is being mixed into a class or trait, it is called a *mixin*. A trait may be parameterized with one or more types. When parameterized with types, the trait constructs a type. For example, Set is a trait that takes a single type parameter, whereas Set[Int] is a type. Also, Set is said to be "the trait of" type Set[Int].

type Every variable and expression in a Scala program has a *type* that is known at compile time. A type restricts the possible values to which a variable can refer, or an expression can produce, at runtime. A variable or expression's type can also be referred to as a *static type* if necessary to differentiate it from an object's *runtime type*. In other words, "type" by itself means static type. Type is distinct from class because a class that takes type parameters can construct many types. For example, List is a class, but not a type. List[T] is a type with a free

type parameter. `List[Int]` and `List[String]` are also types (called *ground types* because they have no free type parameters). A type can have a “class” or “trait.” For example, the class of type `List[Int]` is `List`. The trait of type `Set[String]` is `Set`.

type constraint Some annotations are type constraints, meaning that they add additional limits, or constraints, on what values the type includes. A typical example is that `@positive` could be a type constraint on the type `Int`, limiting the type of 32-bit integers down to those that are positive. Type constraints are not checked by the standard Scala compiler, but must instead be checked by an extra tool or by a compiler plugin.

uniform access principle The *uniform access principle* states that variables and parameterless functions should be accessed using the same syntax. Scala supports this principle by not allowing parentheses to be placed at call sites of parameterless functions. As a result, a parameterless function definition can be changed to a `val`, or vice versa, without affecting client code.

unreachable At the Scala level, objects can become *unreachable*, after which the memory they occupy may be reclaimed by the runtime. Unreachable does not necessarily mean unreferenced. Reference types (instances of `AnyRef`) are implemented as objects that reside on the JVM’s heap. When an instance of a reference type becomes unreachable, it indeed becomes unreferenced, and is available for garbage collection. Value types (instances of `AnyVal`) are implemented as both primitive type values and Java wrapper types (such as `java.lang.Integer`), which reside on the heap. Value type instances can be boxed (converted from a primitive value to a wrapper object) and unboxed (converted from a wrapper object to a primitive value) throughout the lifetime of the variables that refer to them. If a value type instance currently represented as a wrapper object on the JVM’s heap becomes unreachable, it indeed becomes unreferenced, and is available for garbage collection. But if a value type currently represented as a primitive value becomes unreachable, then it does not become unreferenced, because it does not exist as an object on the

JVM’s heap at that point in time. The runtime may reclaim memory occupied by unreachable objects, but if an `Int`, for example, is implemented at runtime by a primitive Java `int` that occupies some memory in the stack frame of an executing method, then the memory for that object is “reclaimed” when the stack frame is popped when the method completes. Memory for reference types, such as `Strings`, may be reclaimed by the JVM’s garbage collector after they become unreachable.

unreferenced See *unreachable*.

value The result of any computation or expression in Scala is a *value*, and in Scala, every value is an object. The term *value* essentially means the image of an object in memory (on the JVM’s heap or stack).

value type A *value type* is any subclass of `AnyVal`, such as `Int`, `Double`, or `Unit`. This term has meaning at the level of Scala source code. At runtime, instances of value types that correspond to Java primitive types may be implemented in terms of primitive type values or instances of wrapper types, such as `java.lang.Integer`. Over the lifetime of a value type instance, the runtime may transform it back and forth between primitive and wrapper types (*i.e.*, to box and unbox it) many times.

variable A *variable* is a named entity that refers to an object. A variable is either a `val` or a `var`. Both `vals` and `vars` must be initialized when defined, but only `vars` can be later reassigned to refer to a different object.

yield An expression can *yield* a result. The `yield` keyword designates the result of a `for` expression.

Bibliography

- [Abe96] Abelson, Harold and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, second edition, 1996.
- [Emi07] Emir, Burak, Martin Odersky, and John Williams. “Matching Objects With Patterns.” In *Proc. ECOOP*, Springer LNCS, pages 273–295. July 2007.
- [Gam94] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [Kay96] Kay, Alan C. “The Early History of Smalltalk.” In *History of programming languages—II*, pages 511–598. ACM, New York, NY, USA, 1996. ISBN 0-201-89502-1. doi:<http://doi.acm.org/10.1145/234286.1057828>.
- [Lan66] Landin, Peter J. “The Next 700 Programming Languages.” *Communications of the ACM*, 9(3):157–166, 1966.
- [Ode03] Odersky, Martin, Vincent Cremet, Christine Röckl, and Matthias Zenger. “A Nominal Theory of Objects with Dependent Types.” In *Proc. ECOOP’03*, Springer LNCS, pages 201–225. July 2003.
- [Ode05] Odersky, Martin and Matthias Zenger. “Scalable Component Abstractions.” In *Proceedings of OOPSLA*, pages 41–58. October 2005.
- [Ray99] Raymond, Eric. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 1999.

- [Ste99] Steele, Jr., Guy L. “Growing a Language.” *Higher-Order and Symbolic Computation*, 12:221–223, 1999. Transcript of a talk given at OOPSLA 1998.

About the Authors

Martin Odersky

Martin Odersky is the creator of the Scala language. As a professor at EPFL in Lausanne, Switzerland he is working on programming languages, more specifically languages for object-oriented and functional programming. His research thesis is that the two paradigms are two sides of the same coin, to be identified as much as possible. To prove this, he has experimented with a number of language designs, from Pizza to GJ to Functional Nets. He has also influenced the development of Java as a co-designer of Java generics and as the original author of the current javac reference compiler. Since 2001 he has concentrated on designing, implementing, and refining the Scala programming language.

Lex Spoon

Lex Spoon worked on Scala for two years as a post-doc at EPFL. He has a Ph.D. in computer science from Georgia Tech. His research is on programming environments and on better support for distributed development. In addition to Scala, he has worked on a wide variety of languages, ranging from the dynamic language Smalltalk to the scientific language X10. He and his wife live in Atlanta with two cats, a chihuahua, and a turtle.

Bill Venners

Bill Venners is president of Artima, Inc., publisher of Artima Developer (www.artima.com). He is author of the book, Inside the Java Virtual Ma-

chine, a programmer-oriented survey of the Java platform's architecture and internals. His popular columns in JavaWorld magazine covered Java internals, object-oriented design, and Jini. Active in the Jini Community since its inception, Bill led the Jini Community's ServiceUI project, whose ServiceUI API became the de facto standard way to associate user interfaces to Jini services. Bill is also the lead developer and designer of ScalaTest, an open source testing tool for Scala and Java developers.