

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 to 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

Foreword by Neal Gafter

artima
ARTIMA PRESS
MOUNTAIN VIEW, CALIFORNIA

Programming in Scala
PrePrint™ Edition Version 3

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 3 published May 4, 2008
Produced in the United States of America

12 11 10 09 08 3 4 5 6 7

ISBN-10: 0-9815316-0-1
ISBN-13: 978-0-9815316-0-1

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
Foreword	xvii
Preface	xix
Acknowledgments	xx
Introduction	xxii
1. A Scalable Language	29
2. First Steps in Scala	48
3. Next Steps in Scala	62
4. Classes and Objects	83
5. Basic Types and Operations	100
6. Functional Objects	121
7. Built-in Control Structures	140
8. Functions and Closures	157
9. Control Abstraction	178
10. Composition and Inheritance	193
11. Scala's Hierarchy	218
12. Traits	226
13. Packages and Imports	243
14. Assertions and Unit Testing	257
15. Case Classes and Pattern Matching	267
16. Working with Lists	298
17. Collections	328
18. Stateful Objects	349
19. Type Parameterization	369
20. Abstract Members	387
21. Implicit Conversions and Parameters	404
22. Implementing Lists	422
23. For-Expressions Revisited	433
24. Extractors	448
25. Annotations	463
26. Working with XML	469
27. Objects As Modules	482
28. Object Equality	493
29. Combining Scala and Java	507
30. Actors and Concurrency	519
31. Combinator Parsing	550
32. GUI Programming	579
33. The SCells Spreadsheet	590
Glossary	615
Bibliography	630
About the Authors	632
Index	634

Contents

Contents	viii
Foreword	xvii
Preface	xix
Acknowledgments	xx
Introduction	xxii
1 A Scalable Language	29
1.1 A language that grows on you	30
1.2 What makes Scala scalable?	35
1.3 Why Scala?	38
1.4 Scala's roots	45
1.5 Conclusion	47
2 First Steps in Scala	48
Step 1. Learn to use the Scala interpreter	48
Step 2. Define some variables	50
Step 3. Define some functions	52
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	61
3 Next Steps in Scala	62
Step 7. Parameterize Arrays with types	62
Step 8. Use Lists	66

Step 9. Use Tuples	71
Step 10. Use Sets and Maps	72
Step 11. Learn to recognize the functional style	76
Step 12. Read lines from a file	79
Conclusion	82
4 Classes and Objects	83
4.1 Classes, fields, and methods	83
4.2 Semicolon inference	89
4.3 Variable scope	90
4.4 Singleton objects	94
4.5 A Scala application	97
4.6 Conclusion	99
5 Basic Types and Operations	100
5.1 Some basic types	101
5.2 Literals	102
5.3 Operators are methods	107
5.4 Arithmetic operations	110
5.5 Relational and logical operations	112
5.6 Object equality	114
5.7 Bitwise operations	115
5.8 Operator precedence and associativity	117
5.9 Rich wrappers	119
5.10 Conclusion	119
6 Functional Objects	121
6.1 A specification for class Rational	121
6.2 Constructing a Rational	122
6.3 Reimplementing the <code>toString</code> method	123
6.4 Checking preconditions	124
6.5 Adding fields	125
6.6 Self references	127
6.7 Auxiliary constructors	128
6.8 Private fields and methods	129
6.9 Defining operators	131
6.10 Identifiers in Scala	133
6.11 Method overloading	135

6.12 Implicit conversions	137
6.13 A word of caution	138
6.14 Conclusion	139
7 Built-in Control Structures	140
7.1 If expressions	141
7.2 While loops	143
7.3 For expressions	145
7.4 Try expressions	150
7.5 Match expressions	153
7.6 Living without break and continue	154
7.7 Conclusion	156
8 Functions and Closures	157
8.1 Methods	157
8.2 Local functions	159
8.3 First-class functions	160
8.4 Short forms of function literals	162
8.5 Placeholder syntax	163
8.6 Partially applied functions	164
8.7 Closures	168
8.8 Repeated parameters	171
8.9 Tail recursion	173
8.10 Conclusion	177
9 Control Abstraction	178
9.1 Reducing code duplication	178
9.2 Simplifying client code	182
9.3 Currying	184
9.4 Writing new control structures	186
9.5 By-name parameters	189
9.6 Conclusion	191
10 Composition and Inheritance	193
10.1 A two-dimensional layout library	193
10.2 Abstract classes	194
10.3 Defining parameterless methods	195
10.4 Extending classes	198

10.5 Overriding methods and fields	200
10.6 Defining parametric fields	201
10.7 Invoking superclass constructors	202
10.8 Using Override modifiers	203
10.9 Polymorphism and dynamic binding	205
10.10 Declaring final members	208
10.11 Using composition and inheritance	209
10.12 Implementing above, beside, and <code>toString</code>	209
10.13 Defining a factory object	213
10.14 Putting it all together	215
10.15 Conclusion	217
11 Scala's Hierarchy	218
11.1 Scala's class hierarchy	218
11.2 How primitives are implemented	222
11.3 Bottom types	224
11.4 Conclusion	225
12 Traits	226
12.1 How traits work	226
12.2 Thin versus rich interfaces	228
12.3 Example: Rectangular objects	229
12.4 The standard Ordered trait	231
12.5 Traits as stackable modifications	234
12.6 Why not multiple inheritance?	238
12.7 To trait, or not to trait?	241
12.8 Conclusion	242
13 Packages and Imports	243
13.1 Packages	244
13.2 Imports	246
13.3 Implicit imports	250
13.4 Access modifiers	250
13.5 Conclusion	256
14 Assertions and Unit Testing	257
14.1 Assertions	257
14.2 Unit testing in Scala	259

14.3 Using JUnit and TestNG	260
14.4 Using fixtures	263
14.5 Tests as specifications	265
14.6 Property-based testing	265
14.7 Conclusion	266
15 Case Classes and Pattern Matching	267
15.1 A simple example	267
15.2 Kinds of patterns	271
15.3 Pattern guards	280
15.4 Pattern overlaps	281
15.5 Sealed classes	282
15.6 The Option type	284
15.7 Patterns everywhere	286
15.8 A larger example	290
15.9 Conclusion	297
16 Working with Lists	298
16.1 List literals	298
16.2 The List type	298
16.3 Constructing lists	299
16.4 Basic operations on lists	300
16.5 List patterns	301
16.6 Operations on lists, Part I: First-order methods	303
16.7 Operations on lists, Part II: Higher-order methods	313
16.8 Operations on lists, Part III: Methods of the List object	321
16.9 Understanding Scala's type inference algorithm	324
16.10 Conclusion	327
17 Collections	328
17.1 Overview of the library	328
17.2 Sequences	330
17.3 Tuples	334
17.4 Sets and maps	336
17.5 Initializing collections	340
17.6 Immutable collections	341
17.7 Conclusion	347

18 Stateful Objects	349
18.1 What makes an object stateful?	349
18.2 Reassignable variables and properties	352
18.3 Case study: discrete event simulation	355
18.4 A language for digital circuits	356
18.5 The Simulation API	358
18.6 Circuit Simulation	363
18.7 Conclusion	368
19 Type Parameterization	369
19.1 Functional queues	369
19.2 Information hiding	372
19.3 Variance annotations	376
19.4 Checking variance	379
19.5 Lower bounds	382
19.6 Contravariance	383
19.7 Object private data	384
19.8 Conclusion	386
20 Abstract Members	387
20.1 Abstract vals	388
20.2 Abstract vars	389
20.3 Abstract types	390
20.4 Case study: Currencies	393
20.5 Conclusion	403
21 Implicit Conversions and Parameters	404
21.1 Implicit conversions	404
21.2 The fine print	407
21.3 Implicit conversion to an expected type	410
21.4 Converting the receiver	411
21.5 Implicit parameters	414
21.6 View bounds	417
21.7 Debugging implicits	418
21.8 Conclusion	421
22 Implementing Lists	422
22.1 The List class in principle	422

22.2 The ListBuffer class	427
22.3 The List class in practice	428
22.4 Functional at the outside	431
22.5 Conclusion	432
23 For-Expressions Revisited	433
23.1 For-Expressions	434
23.2 The N-Queens Problem	436
23.3 Querying with For-Expressions	438
23.4 Translation of For-Expressions	440
23.5 Going the other way	445
23.6 Generalizing For	445
23.7 Conclusion	447
24 Extractors	448
24.1 An Example	448
24.2 Extractors	449
24.3 Patterns with zero or one variables	452
24.4 Variable argument extractors	454
24.5 Extractors and sequence patterns	456
24.6 Extractors vs Case Classes	457
24.7 Regular Expressions	458
24.8 Conclusion	462
25 Annotations	463
25.1 Why have annotations?	463
25.2 Syntax of annotations	464
25.3 Standard annotations	465
25.4 Conclusion	468
26 Working with XML	469
26.1 Semi-structured data	469
26.2 XML overview	470
26.3 XML Literals	471
26.4 Serialization	473
26.5 Taking XML apart	475
26.6 Deserialization	476
26.7 Loading and saving	477

26.8 Pattern matching on XML	478
26.9 Conclusion	481
27 Objects As Modules	482
27.1 A basic database	482
27.2 Abstraction	485
27.3 Splitting modules into traits	487
27.4 Runtime linking	489
27.5 Tracking module instances	490
27.6 Conclusion	492
28 Object Equality	493
28.1 Equality in Scala	493
28.2 Writing an equality method	494
28.3 Conclusion	506
29 Combining Scala and Java	507
29.1 Using Scala from Java	507
29.2 Annotations	510
29.3 Existential types	515
29.4 Conclusion	518
30 Actors and Concurrency	519
30.1 Overview	519
30.2 Locks considered harmful	519
30.3 Actors and message passing	520
30.4 Better performance through thread reuse	523
30.5 Treating native threads as actors	526
30.6 Good actors style	526
30.7 A longer example: Parallel discrete event simulation	531
30.8 Conclusion	549
31 Combinator Parsing	550
31.1 Example: Arithmetic expressions	551
31.2 Running your parser	553
31.3 Basic Regular Expression Parsers	554
31.4 Another example: JSON	556
31.5 Parser output	557

31.6 Implementing combinator parsers	563
31.7 String literals and regular expressions	571
31.8 Lexing and parsing	572
31.9 Error reporting	573
31.10 Backtracking versus LL(1)	576
31.11 Conclusion	577
32 GUI Programming	579
32.1 Introduction	579
32.2 A first Swing Application	579
32.3 Panels and layouts	581
32.4 Handling events	583
32.5 Example: Celsius/Fahrenheit converter	586
32.6 Conclusion	589
33 The SCells Spreadsheet	590
33.1 Introduction	590
33.2 The visual framework	590
33.3 Disconnecting data entry and display	593
33.4 Formulas	596
33.5 Parsing formulas	597
33.6 Evaluation	603
33.7 Operation Libraries	606
33.8 Change Propagation	609
33.9 Conclusion	611
Glossary	615
Bibliography	630
About the Authors	632
Index	634

Foreword

Martin Odersky made a huge impact on the Java world with his design of the Pizza language. Although Pizza itself never became popular, it demonstrated that object-oriented and functional language features, when combined with skill and taste, form a natural and powerful combination. Pizza's design became the basis for generics in Java, and Martin's GJ (Generic Java) compiler was Sun Microsystem's standard compiler starting in 1.3 (though with generics disabled). I had the pleasure of maintaining this compiler for a number of years, so I can report from first-hand experience that Martin's skill in language design extends to language implementation. While we at Sun tried to simplify program development with piecemeal solutions like the for-each loop, enums, and autoboxing, Martin continued his work on more powerful orthogonal language primitives.

Meanwhile, there has been a backlash against statically typed languages. Experience with Java has shown that programming in a static language can result in an abundance of boilerplate. The common wisdom is that one must abandon static typing to eliminate the boilerplate, and there is a rising interest in dynamic languages such as Python, Ruby, and Groovy. This common wisdom is debunked by the existence of Martin's latest brainchild, Scala.

Scala is a *tastefully typed* language: it is statically typed, but explicit types appear in just the right places. Scala takes powerful features from object-oriented and functional languages, and combines them with a few novel ideas in a beautifully coherent whole. The syntax is so lightweight, and its primitives so expressive, that APIs can be used with virtually no syntactic overhead at all. Examples can be found in standard libraries such as parser combinators and actors. In this sense Scala supports *embedded domain-specific languages*.

Will Scala be the *next great language*? Only time will tell. Martin Odersky and his team certainly have the taste and skill for the job. One thing

is sure: Scala sets a new standard against which future languages will be measured.

Neal Gafter

Preface

I'd like to thank you for purchasing the PrePrint™ Edition of *Programming in Scala*. While mostly complete, this book is still in its final stages of review, editing, layout, and indexing. 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™ 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. Please forgive us for this for the time being, and don't report such problems.

By purchasing Version 3 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
May 4, 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. Thanks also to the Silicon Valley Patterns group for their very helpful review: Dave Astels, Tracy Bialik, John Brewer, Andrew Chase, Bradford Cross, Raoul Duke, John P. Eurich, Steven Ganz, Phil Goodwin, Ralph Jocham, Yan-Fa Li, Tao Ma, Jeffery Miller, Suresh Pai, Russ Rufer, Dave W. Smith, Scott Turnquest, Walter Vannini,

Darlene Wallach, and Jonathan Andrew Wolter. We'd also like to thank De-wayne Johnson and Kim Leedy for their help with the cover art.

We'd also like to extend a special thanks to all of our readers who contributed comments. Your comments were very helpful to us in shaping this into an even better book.

Lastly, Bill would also like to thank Gary Cornell, Greg Doench, Andy Hunt, Mike Leonard, Tyler Ortman, Dave Thomas, and Adam Wright for providing insight and advice on book publishing.

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. All examples in this book compile with Scala version 2.6.1-final.

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

Because the main purpose of this book is to serve as a tutorial, 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 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 and Closures,” discusses in depth functions, the basic building block of functional languages.
- [Chapter 9](#), “Control Abstraction,” shows how to augment Scala’s basic control structures by defining your own control abstractions.
- [Chapter 10](#), “Composition and Inheritance,” discusses more of Scala’s support for object-oriented programming. The topics are not as fundamental as those in [Chapter 4](#), but they frequently arise in practice.
- [Chapter 11](#), “Scala’s Hierarchy,” explains Scala’s inheritance hierarchy.
- [Chapter 12](#), “Traits,” shows Scala’s *trait* mechanism for mixin composition.

- [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](#), “Assertions and Unit Testing,” shows Scala’s assertion mechanism and gives a tour of the various tools available for writing tests in Scala.
- [Chapter 15](#), “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 16](#), “Working with Lists,” explains lists in detail. Lists are probably the most commonly used data structure in Scala programs.
- [Chapter 17](#), “Collections,” shows you how to use the basic Scala collections, such as lists, arrays, tuples, sets, and maps.
- [Chapter 18](#), “Stateful Objects,” explains what stateful objects are, and what Scala provides in terms 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 19](#), “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 20](#), “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 21](#), “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 22](#), “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 23](#), “For-Expressions Revisited,” explains how for-expressions are translated to invocations of `map`, `flatMap`, `filter`, and `foreach`.
- [Chapter 24](#), “Extractors,” shows how to pattern match against arbitrary classes, not just case classes.
- [Chapter 25](#), “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 26](#), “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 27](#), “Objects As Modules,” shows how Scala’s objects are rich enough to remove the need for a separate modules system.
- [Chapter 28](#), “Object Equality,” points out several issues to consider when writing an `equals` method. There are several pitfalls to avoid.
- [Chapter 29](#), “Combining Scala and Java,” discusses several issues that arise when programming in Scala on the Java platform.
- [Chapter 30](#), “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 the traditional concurrency approach.
- [Chapter 31](#), “Combinator Parsing,” shows how to build parsers using Scala’s library of parser combinators.
- [Chapter 32](#), “GUI Programming,” gives a quick tour of a Scala library that simplifies GUI programming with Swing.
- [Chapter 33](#), “The SCells Spreadsheet,” ties everything together by showing a complete spreadsheet implementation.

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 inter-operates 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 chapter you should have a basic feel for what Scala is and what kinds of

¹ Scala is pronounced *skah-la*.

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 not cluttered 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 method of the map you create. In each case, you can use the same easy access syntax for maps as in the example above.

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

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

`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

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.

happens to be defined in Scala’s standard library.⁶ 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 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 any language that attempted to implement all of these abstractions at the same time 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.

⁶Scala comes with a standard library, some of which will be covered in this book. For more information, you can also consult the library’s scaladoc documentation, which is available in the distribution and online at <http://www.scala-lang.org>.

As multicore processors proliferate in the coming years, achieving acceptable performance will demand you use more parallelism in your applications. 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. Even though they are defined in a library, actors feel like an integral part of the Scala language.

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 operations as well as data, 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.⁷ So objects have a lot to do with

⁷Kay, "The Early History of Smalltalk" [Kay96]

language scalability: the same techniques apply to the construction of small as well as large programs.

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 definitions 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 some new 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 for 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 this section: compatibility, brevity, high-level abstractions, and advanced static typing.

Scala is compatible

Scala 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.⁸ 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 descriptions, or glue code. In fact, almost all Scala code makes heavy use of Java libraries, often without programmers being aware of this fact.

⁸There is also a Scala variant that runs on the .NET platform, but the JVM variant currently has better support.

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 29](#) 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 possibilities 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.⁹ 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 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

⁹The 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.

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.¹⁰ It describes a function that takes a character argument (represented by the underscore character), and tests whether it is an upper case letter.¹¹

¹⁰A function literal can be called a *predicate* if its result type is Boolean.

¹¹This use of the underscore as a placeholder for arguments is described in [Section 8.5](#) on page 163.

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 write out the loops and 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.

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.¹² 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 arguments against static types are that they make programs too verbose, prevent programmers from expressing themselves as they wish, and make impossible certain patterns of dynamic modifications of software systems. However, often these 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 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

¹²Generics are discussed in [Chapter 19](#), intersections in [Chapter 12](#), and abstract types in [Chapter 20](#).

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. Although a static type system certainly cannot *replace* unit testing, it can reduce the number of unit tests needed by taking care of some properties that would otherwise need to be tested. Likewise, unit testing can not replace static typing. 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 a 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.¹³ 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 application areas is Peter Landin's 1966 paper "The Next 700 Programming Languages."¹⁴ (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-

¹³ 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.

¹⁴Landin, "The Next 700 Programming Languages" [Lan66]

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.¹⁵

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 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.

¹⁵For more information, see [Ode03], [Ode05], and [Emi07] in the bibliography.

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.¹

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. To get you up to speed quickly, we had to leave out some details. Everything will be explained in a less "firehose" fashion in later chapters. In addition, we inserted in these next two chapters quite a few footnotes to supply more information and point you to later sections of the book where you'll find more detailed explanations.

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.

¹We tested the examples in this book with Scala version 2.6.1-final.

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:²

```
$ scala  
Welcome to Scala version 2.7.0-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.³ 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`. `scala.Float` corresponds to Java's `float`.

²If you're using Windows, you'll need to type the `scala` command into the "Command Prompt" DOS box.

³If 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](#).

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`. A `val` is similar to a final variable in Java. Once initialized, a `val` can never be reassigned. A `var`, by contrast, is similar to a non-final variable 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` nor `String` appear anywhere in the `val` definition. This example illustrates *type inference*, Scala's ability to figure out types you leave out. In this case, because you initialized `msg` with a string literal, Scala inferred the type of `msg` to be `String`. When the Scala interpreter (or compiler) can infer a type, it is often 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 names⁴ 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.⁵ For example, see how the interpreter complains when you attempt the following:

```
scala> msg = "Goodbye cruel world!"  
<console>:5: error: reassignment to val  
      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:

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

⁴The simple name of `java.lang.String` is `String`.

⁵In the interpreter, however, you can *define* a new `val` with a name that was already used before. This mechanism is explained in [Section 4.3](#).

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 `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.

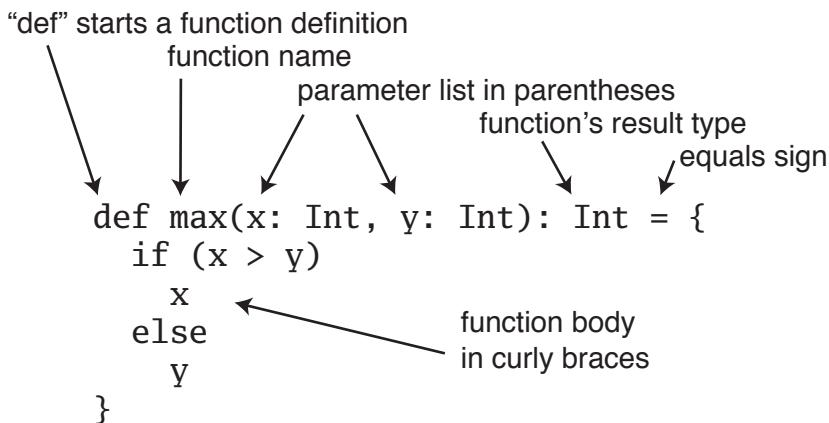


Figure 2.1: The basic form of a function definition in Scala.

This one defines the *result type* of the `max` function itself.⁶ 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. As 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. 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 of just one statement, you can optionally leave off the curly braces. Thus, you could alternatively write the `max` function like this:

⁶In Java, the type of the value returned from a method is its return type. In Scala, that same concept is called *result type*.

⁷A function is recursive if it calls itself.

⁸Nevertheless, it is often a good idea to indicate function result types explicitly, even when the compiler doesn't require it. 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.

```
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 property—some attribute or characteristic that can be expressed as a value—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 property (the length of a `String`), you should invoke it without parentheses, as in `msg.length`.

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.”

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 naming the file helloarg.bat and placing this at the top of your script:

```
::#!@echo offcall 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 named `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 with the name `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 semicolons to separate statements as in Java, except that in Scala the semicolons 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();
```

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:

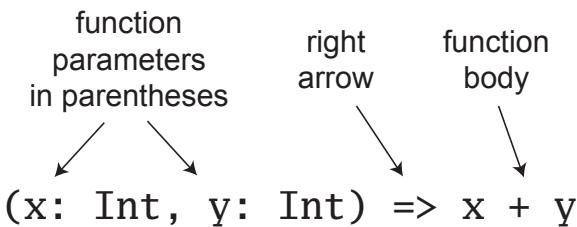


Figure 2.2: The syntax of a function literal in Scala.

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

You'll get:

```
Explicit  
can  
be  
nice  
too
```

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.¹⁰ 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

¹⁰This shorthand is called a *partially applied function*, which is described in [Section 8.6](#) on [page 164](#).

peek ahead to) [Section 7.3 on page 145](#), we'll give you a glimpse here. In a new file named `forargs.scala`, type the following:

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

The parentheses after “`for`” in this `for` expression contain `arg <- args`.¹¹ 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`). (Because it is always a `val`, you just write `arg` by itself, not `val arg`.) 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 `forargs.scala` script with the command:

```
$ scala forargs.scala
```

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.

¹¹ You 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. Parameterize Arrays with types

In Scala, as in Java, you define a blueprint for objects with classes. From a class blueprint, you can instantiate objects, or class instances, using new. When you instantiate an object in Scala, you can *parameterize* it with values and types. Parameterization means configuring an instance when you create it. You parameterize an instance with values by passing objects to a constructor of the instance in parentheses, as in Java. For example, the following Scala code instantiates a new `java.lang.BigInteger` and parameterizes it with the value "12345":

```
val big = new java.math.BigInteger("12345")
```

Parameterization with types is akin to specifying one or more types 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 parameterizing it with the value 3 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 changed. 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)
    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. The `to` in this example 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 different kind of sequence, containing the values 0, 1, and 2, which the `for` expression iterates over. Sequences and other collections will be described in [Chapter 17](#).) Note that this syntax only works if you explicitly specify the receiver of the method call. You cannot write `println 10`, but you can write `Console println 10`.

Scala doesn't technically have operator overloading, because it doesn't actually have operators in the traditional sense. Instead, characters such as `+`, `-`, `*`, and `/` 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 one or more variables and pass in arguments, 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.

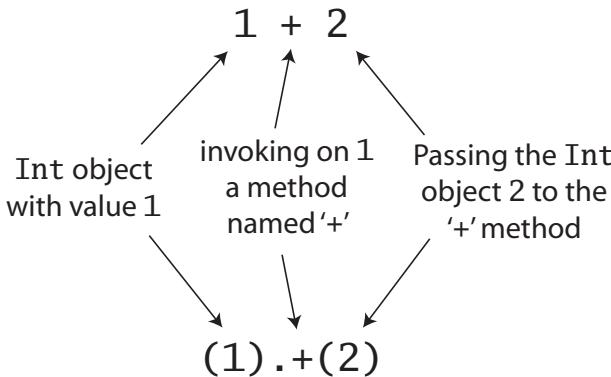


Figure 3.1: All operations are method calls in Scala.

Similarly, when an assignment is made to a variable to which parentheses and one or more arguments have been applied, the compiler will transform that into an invocation of an update method that takes the arguments in parentheses as well as the object to the right of the equals sign. 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.

Although the examples you've seen so far in this step compile and run just fine, Scala provides a more concise way to create and initialize arrays that you would normally use in real code. It looks like this:

```
val numNames = Array("zero", "one", "two")
```

This code creates a new array of length three, initialized to the passed strings, "one", "two", and "three". The compiler infers the type of the array to be `Array[String]`, because you passed strings to it. What you're actually doing here is calling a factory method, named `apply`, on the `Array companion object`. You'll learn more about companion objects in [Step 11](#). If you're a Java programmer, you can think of this as calling a static method named `apply` on class `Array`. A more verbose way to call the same `apply` method, therefore, is:

```
val numNames2 = Array.apply("zero", "one", "two")
```

Step 8. Use Lists

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. For an immutable sequence of objects that share the same type there's is Scala's `List` class. 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). More generally, 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 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

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.

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

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)`

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

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 `b`, passing in `a`, like this: `b :: (a)`. Thus, `b` prepends `a` to itself and returns the result. This is described in more detail in [Section 5.8](#).

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 22.2](#) on [page 427](#).

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

Nil	An empty List
List("Cool", "tools", "rule")	Creates a new List[String] with the three values "Cool", "tools", and "rule"
val thrill = "Will" :: "fill" :: "until" :: Nil	Creates a new List[String] with the three values "Will", "fill", and "until"
List("a", "b") :: List("c", "d")	Concatenates two lists (returns a new List[String] with values "a", "b", "c", and "d")
thrill(2)	Returns the element at index 2 (zero based) of the thrill List (returns "until")
thrill.count(s => s.length == 4)	Counts the number of String elements in thrill that have length 4 (returns 2)
thrill.drop(2)	Returns the thrill List without its first 2 elements (returns List("until"))
thrill.dropRight(2)	Returns the thrill List without its rightmost 2 elements (returns List("Will"))
thrill.exists(s => s == "until")	Determines whether a String element exists in thrill that has the value "until" (returns true)
thrill.filter(s => s.length == 4)	Returns a List of all elements, in order, of the thrill List that have length 4 (returns List("Will", "fill"))
thrill.forall(s => s.endsWith("l"))	Indicates whether all elements in the thrill List end with the letter "l" (returns true)

Table 3.1: continued

<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 List("Will", "fill"))
<code>thrill.isEmpty</code>	Indicates whether the <code>thrill</code> List is empty (returns <code>false</code>)
<code>thrill.last</code>	Returns the last element in the <code>thrill</code> List (returns "until")
<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 List("Willy", "filly", "untily"))
<code>thrill.mkString(", ")</code>	Makes a string with the elements of the list (returns "will, fill, until")
<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 List("until"))
<code>thrill.reverse</code>	Returns a List containing all elements of the <code>thrill</code> List in reverse order (returns List("until", "fill", "Will"))

Table 3.1: continued

thrill.sort((s, t) => s.charAt(0).toLowerCase < t.charAt(0).toLowerCase)	Returns a List containing all elements of the thrill List in alphabetical order of the first character lowercased (returns List("fill", "until", "Will"))
thrill.tail	Returns the thrill List minus its first element (returns List("fill", "until"))

Step 9. Use Tuples

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.

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
```

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`.

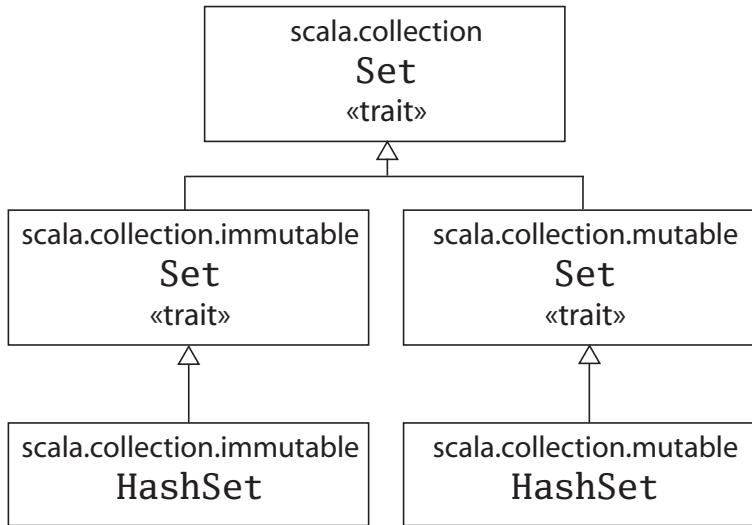


Figure 3.2: Class hierarchy for Scala Sets.

```
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 `String`s. 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 an `apply` factory method defined in `scala.collection.immutable.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`:

```
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

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 want to pass a tuple into a function that takes repeated parameters (*i.e.*, “`varargs`”), you'll need two sets of parentheses, as in `function(("a", "b"))`. The outer parentheses enclose the parameter list; the inner ones define the tuple. Repeated parameters are described in [Section 8.8](#).

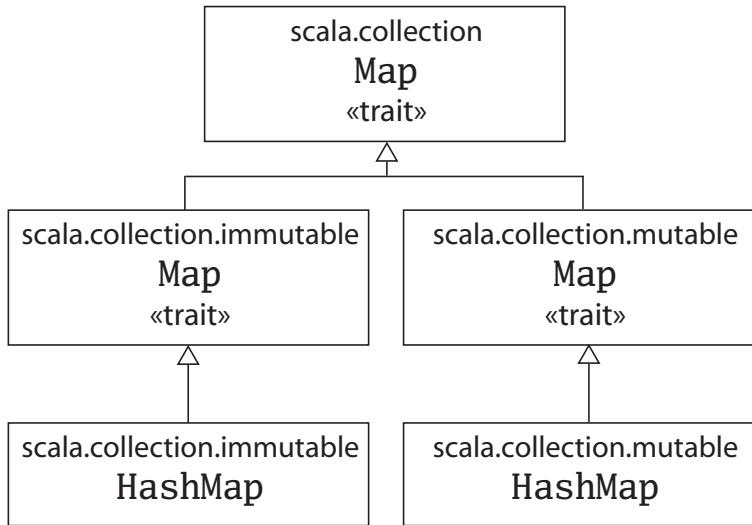


Figure 3.3: Class hierarchy for Scala Maps.

(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(
```

⁸The Scala mechanism that allows you to invoke `->` on any object is called *implicit conversion*, which will be covered in Chapter 21.

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

When you say `Map` in the first line of `numerals.scala`, 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. 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.

Step 11. Learn to recognize the functional style

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 telltale sign is that if code contains any `vars`, variables that can be reassigned, it is probably in an imperative style. If the code contains no `vars` at all—*i.e.*, it contains *only* `vals`—it is probably in a more functional style. Thus one way to move towards a functional style is to try to program without any `vars`.

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. 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. Scala encourages you to lean towards `vals`, but ultimately reach, without guilt, for the best tool given the job at hand. Even if you agree with this balanced philosophy, however, you may still find it challenging at first to figure out how to get rid of `vars` in your code.

Consider the following while loop example, taken from [Chapter 2](#), which uses a `var` and is therefore in the imperative style. Here we’ve placed it inside a function named `printArgs`:

```
def printArgs(args: Array[String]): Unit = {  
    var i = 0  
    while (i < args.length) {  
        println(args(i))  
        i += 1  
    }  
}
```

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

```
def printArgs(args: Array[String]): Unit = {  
    for (arg <- args)  
        println(arg)  
}
```

or this:

```
def printArgs(args: Array[String]): Unit = {  
    args.foreach(println)  
}
```

This example illustrates an important benefit of programming with fewer `vars`. The refactored (more functional) code is more concise, clearer, and less error-prone than the original (more 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.

You can go even farther, though. The refactored `printArgs` methods is not *purely* functional, because it has side effects—in this case, its side effect is printing to the standard output stream. The telltale sign of a function with side effects is that its result type is `Unit`. If a function isn't returning any useful value, which is what a result type of `Unit` means, the only way that function can make a difference in the world is through some kind of side effect. A more functional approach would be to define a method that formats the passed args for printing, but just returns the formatted string:

```
def formatArgs(args: Array[String]): String =  
    args.mkString("\n")
```

Now you're really functional: no side effects or vars in sight. The `mkString` method, which you can call on Arrays, returns a string consisting of the result of calling `toString` on each array element, separated by the passed string. Thus if `args` contains three elements "zero", "one", and "two", `formatArgs` will return "zero\none\ntwo". Of course, this function doesn't actually print anything out like the `printArgs` methods did, but you can easily pass its result to `println` to accomplish that:

```
println(formatArgs(args))
```

Every useful program is likely to have side effects of some form, because otherwise it wouldn't be able to provide value to the outside world. Preferring methods without side effects encourages you to design programs where side effecting code is minimized. One benefit of this approach is that it can help make your programs easier to test, because code without side effects is easier to test. For example, to test any of the three `printArgs` methods shown earlier in this section, you'd need to redefine `println`, capture the output passed to it, and make sure it is what you expect. By contrast, you could test `formatArgs` simply by checking its result:

```
val res = formatArgs(Array("zero", "one", "two"))  
assert(res == "zero\none\ntwo")
```

Scala's `assert` method checks the passed Boolean and if it is false, throws `AssertionError`. If the passed Boolean is true, `assert` just returns quietly. You'll learn more about assertions and testing in [Chapter 14](#).

That said, bear in mind that neither vars nor side effects are inherently 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.

The attitude we suggest you adopt is to be suspicious of vars and side effects 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. Likewise, try and minimize code with side effects. To help you learn how to program without vars, we'll show you many specific examples of code with vars and how to transform those vars to vals in [Chapter 7](#).

Step 12. Read lines from a file

Scripts that perform small, everyday tasks often need to process lines in files. In this section, you'll build a script that reads lines from a file, and prints them out prepended with the number of characters in each line. Here's the first version:

```
import scala.io.Source  
  
if (args.length > 0) {  
    for (line <- Source.fromFile(args(0)).getLines)  
        print(line.length + " " + line)  
}  
else  
    Console.err.println("Please enter filename")
```

This script starts with an import of a class named `Source` from package `scala.io`. It then checks to see if at least one argument was specified on the command line. If so, the first argument is interpreted as a filename to open and process. The expression, `Source.fromFile(args(0))`, attempts to open the specified file and returns a `Source` object, on which you call `getLines`. The `getLines` method returns an `Iterator[String]`, which provides one line on each iteration, including the end of line character. The for-expression iterates through these lines and prints for each the length of the line, a space, and the line itself. If there were no arguments supplied on the command line, the final else clause will print a message to the standard error stream. If you place this code in a file named `countchars1.scala`, and run it on itself with:

```
$ scala countchars1.scala countchars1.scala
```

You should see:

```
23 import scala.io.Source
1
23 if (args.length > 0) {
1
50   for (line <- Source.fromFile(args(0)).getLines)
36     print(line.length + " " + line)
2 }
5 else
47   Console.err.println("Please enter filename")
```

Although the script in its current form prints out the needed information, you may wish to line up the numbers, right adjusted, so that the output looks instead like:

```
23 import scala.io.Source
1
23 if (args.length > 0) {
1
50   for (line <- Source.fromFile(args(0)).getLines)
36     print(line.length + " " + line)
2 }
5 else
47   Console.err.println("Please enter filename")
```

To accomplish this, you can iterate through the lines twice. The first time through you'll determine the maximum width required by any line's character count. The second time through you'll print the output, using the maximum width calculated previously. Because you'll be iterating through the lines twice, you may as well assign them to a variable:

```
val lines = Source.fromFile(args(0)).getLines.toList
```

The final `toList` is required because the `getLines` method returns an iterator. Once you've iterated through an iterator, it is spent. By transforming it into a `List` via the `toList` call, you gain the ability to iterate as many times as you wish, at the cost of storing all lines from the file in memory at once.

The `lines` variable, therefore, references an array of strings that contains the contents of the file specified on the command line.

Next, because you'll be calculating the width of each line's character count twice, once per iteration, you might factor that expression out into a small function, which calculates the character width of the passed string's length:

```
def widthOfLength(s: String) = s.length.toString.length
```

With this function, you could calculate the maximum width like this:

```
var maxWidth = 0
for (line <- lines)
    maxWidth = maxWidth.max(widthOfLength(line))
```

Here you iterate through each line with a `for`-expression, calculate the character width of that line's length, and, if it is larger than the current maximum, assign it to `maxWidth`, a var that was initialized to 0. (The `max` method, which you can invoke on any `Int`, returns the greater of the value on which it was invoked and the value passed to it.) Alternatively, if you prefer to find the maximum without vars, you could first transform the list of lines into a list of character widths, like this:

```
val widths = lines.map(widthOfLength)
```

The `map` method applies the passed function to each element of the `List[String]`, and returns the results as a `List[Int]`. Thus `widths` will reference a list of integer widths, one per line. Given this list, you can calculate the maximum width by calling `reduceLeft`, like this:

```
val maxWidth = widths.reduceLeft((a, b) => a.max(b))
```

The `reduceLeft` method applies the passed function to the first two elements in `widths`, then applies it to the result of the first application and the next element in `widths`, and so on, all the way through the list. On each such application, the result will be the maximum integer encountered so far, because the passed function, `(a, b) => a.max(b)`, returns the maximum of two passed integers. `reduceLeft` will return the result of the last application of the function, which in this case will be the maximum integer element contained in `widths`.

All that remains is to print out the lines with proper formatting. You can do that like this:

```
for (line <- lines) {  
    val numSpaces = maxWidth - widthOfLength(line)  
    val padding = " " * numSpaces  
    print(padding + line.length + " " + line)  
}
```

In this for-expression, you once again iterate through the lines. For each line, you first calculate the number of spaces required before the line length and assign it to numSpaces. Then you create a string containing numSpaces spaces with the expression: " " * numSpaces. Finally, you print out the information with the desired formatting. The entire script looks like:

```
import scala.io.Source  
  
def widthOfLength(s: String) = s.length.toString.length  
  
if (args.length > 0) {  
    val lines = Source.fromFile(args(0)).getLines.toList  
    val widths = lines.map(widthOfLength)  
    val maxWidth = widths.reduceLeft((a, b) => a.max(b))  
  
    for (line <- lines) {  
        val numSpaces = maxWidth - widthOfLength(line)  
        val padding = " " * numSpaces  
        print(padding + line.length + " " + line)  
    }  
}  
else  
    Console.err.println("Please enter filename")
```

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

To write a program with complex requirements, you must decompose those requirements into smaller, simpler pieces. In Scala, the most fundamental unit of program decomposition is the class. To manage complexity, you can define classes whose requirements are manageable. You can instantiate those classes into objects and orchestrate the objects in ways that solve the larger problem. In the process of designing your classes, you can also define interfaces that abstract away the details of the implementation. As you and other client programmers work with the objects, you can think primarily in terms of their abstract interfaces, and to a great extent 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 Classes, fields, and methods

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

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

```
new ChecksumAccumulator
```

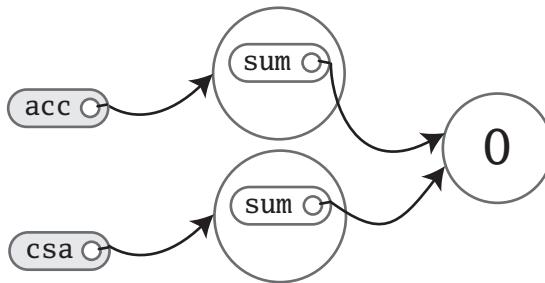
Inside a class definition, you place fields and methods, which are collectively called *members*. Fields, which you define with either `val` or `var`, are variables that refer to objects. Methods, which you define with `def`, contain executable code. The fields hold the state, or data, of an object, 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 `ChecksumAccumulator` class and gave it a `var` field named `sum`:

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

and you instantiated it twice with:

```
val acc = new ChecksumAccumulator  
val csa = new ChecksumAccumulator
```

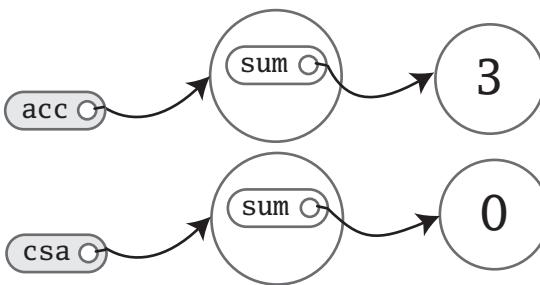
The image of the objects in memory might look like:



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

```
acc.sum = 3
```

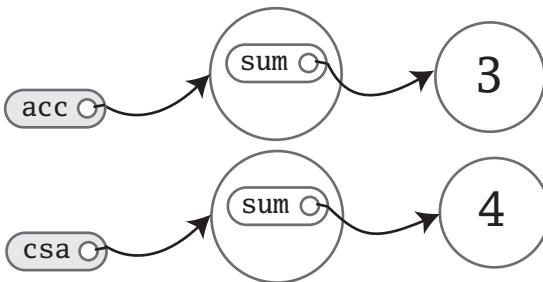
Now the picture would look like:



The first thing to notice about this picture is that there are two `sum` variables, one inside the `ChecksumAccumulator` object referred to by `acc` and the other inside the `ChecksumAccumulator` object referred to by `csa`. 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 `acc.sum = 3`, for example, the `sum` inside the `ChecksumAccumulator` referred to by `csa` remained at `0`. Similarly, if you reassigned the `sum` instance variable inside the `ChecksumAccumulator` object referred to by `csa`:

```
csa.sum = 4
```

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



Another thing to note in this example is that you were able to mutate the objects `acc` and `csa` referred to, even though both `acc` and `csa` are `vals`. What you can't do with `acc` or `csa` 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 acc is a val
acc = new ChecksumAccumulator
```

What you can count on, then, is that `cc` will always refer to the same `ChecksumAccumulator` 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 ChecksumAccumulator {
    private var sum = 0
}
```

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

```
val acc = new ChecksumAccumulator
acc.sum = 5 // 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, `ChecksumAccumulator` won't be of much use to anyone unless we define some methods in it:

```
class ChecksumAccumulator {
    private var sum = 0
    def add(b: Byte): Unit = {
        sum += b
    }
}
```

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

The `ChecksumAccumulator` 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 53](#).

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 `ChecksumAccumulator` 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. On the other hand, design choices depend on the design context, and Scala makes it easy to write methods that have multiple, explicit `returns` 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 ChecksumAccumulator {  
    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 `ChecksumAccumulator`'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 `ChecksumAccumulator`:

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

One puzzler 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 puzzler 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
res0: java.lang.String = this String gets returned!
```

4.2 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. On the other hand, a semicolon is required if you write multiple statements on a single line:

```
val s = "hello"; println(s)
```

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 it to parse 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 a 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.

4.3 Variable scope

Variable declarations in Scala programs have a *scope* that defines where you can use the name. If you're a Java programmer, you'll find that Scala's scoping rules are almost identical to Java's. One difference between Java and Scala exists, however, in that Scala allows you to define variables of the same name in nested scopes. If you're a Java programmer, therefore, you may wish to at least skim this section quickly.

The most common example of scoping is that curly braces generally introduce a new scope, so anything defined inside curly braces leaves scope after the final closing brace.¹ 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) {
```

¹There 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 145.

```
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
}

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
    println(a)
}
println(a)
```

When executed, the script shown previously would print 2 then 1, because the `a` defined inside the curly braces is a different variable, which is in scope only until the closing curly brace.² 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.

You might have already noticed something that looks like shadowing in the interpreter:

```
scala> val a = 1
a: Int = 1
scala> val a = 2
a: Int = 2
```

²By 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.

```
scala> println(a)
```

```
2
```

In the interpreter, 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 time in the interpreter. The reason you can do this is that, conceptually, 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. It is usually better to choose a new, meaningful variable name rather than to shadow an outer variable.

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(" " * (4 - prod.length))
      print(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, the way you should 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`.

4.4 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 ChecksumAccumulator.scala
import scala.collection.mutable.Map

object ChecksumAccumulator {

    private val cache = Map[String, Int]()

    def calculate(s: String): Int =
        if (cache.contains(s))
            cache(s)
        else {
            val acc = new ChecksumAccumulator
```

```
for (c <- s)
    acc.add(c.toByte)
val cs = acc.checksum()
cache += (s -> cs)
cs
}
}
```

This singleton object is named `ChecksumAccumulator`, 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 `ChecksumAccumulator` singleton object has one method, `calculate`, which takes a `String` and calculates a checksum for the characters in the `String`. It also has one private field, `cache`, a mutable map in which previously calculated checksums are cached.³ The first line of the method, “`if (cache.contains(s))`,” checks the cache to see if the passed string is already contained as a key in the map. If so, it just returns the mapped value, “`cache(s)`.” Otherwise, it executes the else clause, which calculates the checksum. The first line of the else clause defines a `val` named `acc` and initializes it with a new `ChecksumAccumulator` instance. Because the keyword `new` is only used to instantiate classes, the new object created here is an instance of the `ChecksumAccumulator` 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 passes that to the `add` method of the `ChecksumAccumulator` instances to which `acc` refers. After the for-expression completes, the next line of the method invokes `checksum` on `acc`, which gets the checksum for the passed `String`, and stores it into a `val` named `cs`. In the next line, “`cache += (s -> cs)`,” the passed string key is mapped to the integer checksum value, and this

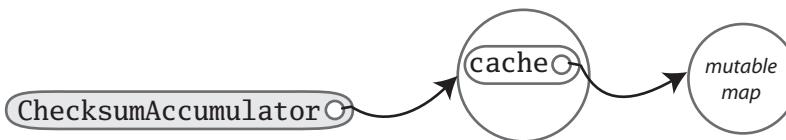
³We used a cache here to show a singleton object with a field. A cache such as this is a performance optimization that trades off memory for computation time. In general, you would likely use such a cache only if you encountered a performance problem that the cache solves, and might use a weak map, such as `WeakHashMap` in `scala.collection.jcl`, so that entries in the cache could be garbage collected if memory becomes scarce.

key-value pair is added to the cache map. The last expression of the method, “cs,” ensures the checksum is the result of 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 `calculate` method of singleton object `ChecksumAccumulator` like this:

```
ChecksumAccumulator.calculate("Every value is an object.")
```

A singleton object is more than a holder of static methods, however. It is a first-class object. You can think of a singleton object’s name, therefore, as a “name tag” attached to the object:



Defining a singleton object doesn’t define a type (at the Scala level of abstraction). Given just a definition of object `ChecksumAccumulator`, you can’t make a variable of type `ChecksumAccumulator`. Rather, `ChecksumAccumulator` is the name of the type defined by the singleton object’s companion class. However, all singleton objects extend a superclass and can mix in traits. Given each singleton object is an instance of its superclasses and mixed-in traits, you can invoke its methods via these types, refer to it from variables of these types, and pass it to methods expecting these types. We’ll show some examples of using a singleton object via supertype references in [Chapter 12](#).

One difference between classes and singleton objects is that singleton objects cannot take parameters, whereas classes can via primary and auxiliary constructors. Because you can’t instantiate a singleton object with the `new` keyword, you can’t pass parameters to its primary constructor. Each singleton object is implemented as an instance of a synthetic class referenced from a static variable, so they have the same initialization semantics as Java statics.⁴ In particular, a singleton object is initialized the first time someone

⁴The name of the synthetic class is the object name plus a dollar sign. Thus the synthetic class for the singleton object named `ChecksumAccumulator` is `ChecksumAccumulator$`.

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.

4.5 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 ChecksumAccumulator.calculate

object Summer {
    def main(args: Array[String]) {
        for (arg <- args)
            println(arg + ": " + calculate(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 `calculate` method defined in the `ChecksumAccumulator` 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 `ChecksumAccumulator`, place the code shown next in a file named `ChecksumAccumulator.scala`:

```
// In file ChecksumAccumulator.scala
class ChecksumAccumulator {
    private var sum = 0
```

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

object ChecksumAccumulator {
    def calculate(s: String): Int = {
        val acc = new ChecksumAccumulator
        for (c <- s)
            acc.add(c.toByte)
        acc.checksum()
    }
}
```

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, `ChecksumAccumulator.scala` and `Summer.scala`.

Neither `ChecksumAccumulator.scala` nor `Summer.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 Summer.scala # This won't work!
```

the Scala interpreter will complain that `Summer.scala` does not end in a result expression (assuming of course you didn't add any expression of your own after the `Summer` 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 ChecksumAccumulator.scala Summer.scala
```

This compiles your source files, but there may be a perceptible delay before the compilation finishes. The reason is that every time the compiler starts up,

it spends time scanning the contents of `jar` files and doing other initial work before it even looks at the fresh source files you submit to it. For this reason, the Scala distribution also includes a Scala compiler *daemon* called `fsc` (for fast Scala compiler). You use it like this:

```
fsc ChecksumAccumulator.scala Summer.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,⁵ in this case you'll give it the name of a standalone object containing a `main` method of the proper signature. You can run the `Summer` application, therefore, by typing:

```
scala Summer of love
```

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

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

4.6 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.

⁵The 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.

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, 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 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`.¹ 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 can just use the simple names (*i.e.*, names like `Boolean`, or `Char`, or `String`) everywhere.²

¹Packages will be covered in depth in [Chapter 13](#).

²You 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

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 raw String literal, which is described on [page 105](#).

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
```

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 0x0OFF 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

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.

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 = 1.23E45
```

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 can also be given as a general unicode character consisting of four hex digits and preceded by a \u, as in:

```
scala> val d = '\u0041'  
d: Char = A  
scala> val f = '\u005a'  
f: Char = Z
```

In fact, such unicode characters can appear anywhere in a Scala program. For instance you could also write an identifier such as

B\u0041\u005a

That identifier is treated as identical to BAZ, the result of expanding the two unicode characters in the code above.

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

For example:

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

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
```

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)

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 strings that contain a lot of escape sequences or strings that span multiple lines. For those, Scala includes a special syntax for *raw* strings. You start and end a raw string with three quotation marks in a row (""""). The interior of a raw 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 raw 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 literally³ 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.

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)`.⁴ In other words, class `Int` contains a method named `+` that takes an `Int` and returns an `Int` result. This `+` method is invoked when you add two `Ints`,⁵ as in:

³figuratively speaking

⁴By the way, the spaces around operators shown in this book usually just for style. `1+2` would compile just as fine as `1 + 2`.

⁵The Scala compiler will generally optimize this Scala-level method invocation down to native Java bytecode addition on primitive Java `ints` in the class files it generates.

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

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.⁶ 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
```

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”`.

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!
```

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

```
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!⁷

Postfix operators are methods that take no arguments, when they are 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!
```

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 117](#).

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:

⁷All is not necessarily lost, however. There is an extremely slight chance your program with the `*p` might compile as C++.

```
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 the left side for `null`, and if it is not `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.⁸

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 11.1](#) and [11.2](#). Also, see [Chapter 28](#) on how to write a good `equals` method.

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
```

⁸The automatic check does not look at the right-hand side, but any reasonable `equals` method should return `false` if its argument is `null`.

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 ^ 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, which in binary looks like 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

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

-1 in binary is 11111111111111111111111111111111. In the first example, `-1 >> 31`, -1 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 `>>` 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 >>> 31`, the leftmost bit is again shifted right until it is in the rightmost position, but this time filling with zeroes along the way. Thus the result this time is binary 00000000000000000000000000000001, or 1. In the final example, `1 << 2`, -the left operand, 1, is shifted left two positions (filling in with zeroes), resulting in binary 0000000000000000000000000000000100, or 4.

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.

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 c are values or variables, and $+++$ and $***$ are methods) will be evaluated $a +++ (b *** c)$, 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.

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
```

Table 5.3: Operator precedence.

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

Since the first characters are the same as in the previous example, 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 way operators are grouped. The associativity of an operator in Scala is determined by its *last* character. As mentioned in [Chapter 3](#) in [footnote 2](#) on [page 68](#), 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). However, no matter what associativity an operator has, its operands are always evaluated left to right. So if b is an expression that is not just a simple reference to an immutable value, then a :::: b is more precisely treated as the following block:

```
{ val x = a; b.::::(x) }
```

In this block a is still evaluated before b, and then the result of this evaluation is passed as an operand to b’s :::: method.

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 grouped right to left; otherwise, they are grouped left to right. For example,

`a :::: b :::: c` is treated as `a :::: (b :::: c)`. But `a * b *c` is treated as `(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 21](#). 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 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 that give new implementations of some of the operators that you have seen so far.

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

Table 5.5: Rich wrapper classes.

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

Chapter 6

Functional Objects

With the understanding of Scala basics you gained in previous chapters, you're ready to see how to design a more full-featured object in Scala. As mentioned previously, objects designed in the functional style are immutable. In other words, immutable objects are *functional objects*. Although you can easily create both mutable and immutable objects in Scala, in this chapter we'll create a class that models rational numbers as immutable objects. Along the way, we'll show you more aspects of object-oriented programming in Scala: class parameters and constructors, methods and operators, private members, overriding, checking preconditions, overloading, and self references.

6.1 A specification for class Rational

A *rational number* is a number that can be expressed as a ratio $\frac{n}{d}$, where n and d are integers, except that d cannot be zero. n is called the *numerator* and d the *denominator*. Examples of rational numbers are $\frac{1}{2}$, $\frac{2}{3}$, $\frac{112}{239}$, and $\frac{2}{1}$. Compared to floating-point numbers, rational numbers have the advantage that fractions are represented exactly, without any rounding or approximation.

The class we'll design in this chapter must model the behavior of rational numbers, including allowing them to be added, subtracted, multiplied, and divided. To add two rationals, you must first obtain a common denominator, then add the two numerators. For example, to add $\frac{1}{2} + \frac{2}{3}$, you multiply both parts of the left operand by 3 and both parts of the right operand by 2, which gives you $\frac{3}{6} + \frac{4}{6}$. Adding the two numerators yields the result, $\frac{7}{6}$. To mul-

ultiply two rational numbers, you can simply multiply their numerators and multiply their denominators. Thus, $\frac{1}{2} * \frac{2}{5}$ gives $\frac{2}{10}$, which can be represented more compactly in its “normalized” form as $\frac{1}{5}$. You divide by swapping the 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 in mathematics, rational numbers do not have mutable state. You can add one rational number to another, but the result will be a new rational number. The original numbers will not have “changed.” The immutable Rational class we’ll design in this chapter will have the same property. Each rational number will be represented by one Rational object. When you add two Rational objects, you’ll create a new Rational object to hold the sum.

This chapter will give you a glimpse of some of the ways Scala enables you to write libraries that feel like native language support. For example, at the end of this chapter you’ll be able to do this with class Rational:

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2
scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3
scala> (oneHalf / 7) + (1 - twoThirds)
res0: Rational = 17/42
```

6.2 Constructing a Rational

A good place to start designing class Rational is to consider how client programmers will create a new Rational object. Given you’ve decided to make Rational objects immutable, you’ll require that clients provide all data needed by an instance (in this case, a numerator and a denominator) when they construct the instance. Thus, you could start the design with this:

```
class Rational(n: Int, d: Int)
```

One of the first things to note about this line of code is that if a class doesn’t have a body, you don’t need to specify empty curly braces (though you could, of course, if you wanted to). The n and d in the parentheses after the class name, Rational, are called *class parameters*. The Scala compiler

will gather up these two class parameters and create a *primary constructor* that takes the same two parameters.

This example highlights a difference between Java and Scala. Java classes have constructors, which can take parameters, whereas 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.

The Scala compiler will compile any code you place in the class body, which isn't part of a field or a method definition, into the primary constructor. For example, you could print a debug message like this:

```
class Rational(n: Int, d: Int) {  
    println("Created "+n+"/"+d)  
}
```

Given this code, the Scala compiler would place the call to `println` into `Rational`'s primary constructor. The `println` call will, therefore, print its debug message whenever you create a new `Rational` instance:

```
scala> new Rational(1, 2)  
Created 1/2  
res0: Rational = Rational@a0b0f5
```

6.3 Reimplementing the `toString` method

When you created an instance of `Rational` in the previous example, the interpreter printed “`Rational@a0b0f5`.” The interpreter obtained this somewhat funny looking string by calling `toString` on the `Rational` object. By default, class `Rational` inherits the implementation of `toString` defined in class `java.lang.Object`, which just prints the class name, an @ sign, and a hexadecimal number. The result of `toString` is primarily intended to help programmers by providing information that can be used in debug print statements, log messages, test failure reports, and interpreter and debugger output. The form of the string currently provided by `toString` would not be especially helpful, because it doesn't give any clue as to the values of the fields of the `Rational` on which `toString` is being invoked. A more useful

implementation of `toString` would print out the values of the `Rational`'s numerator and denominator. You can *override* the default implementation by adding a method `toString` to class `Rational`, like this:

```
class Rational(n: Int, d: Int) {  
    override def toString = n+"/"+d  
}
```

The `override` modifier in front of a method definition signals that a previous method definition is overridden; more on this in [Chapter 10](#). Since `Rational` numbers will display nicely now, we removed the debug `println` statement we put into the body of previous version of class `Rational`. You can test the new behavior of `Rational` in the interpreter:

```
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
```

6.4 Checking preconditions

As a next step, we will turn our attention to a problem with the current behavior of the primary constructor. As mentioned at the beginning of this chapter, rational numbers may not have a zero in the denominator. Currently, however, the primary constructor happily accepts a zero passed as `d`:

```
scala> new Rational(5, 0)  
res6: Rational = 5/0
```

One of the benefits of object-oriented programming is that it allows you to encapsulate data inside objects so that you can ensure the data is valid throughout its lifetime. In the case of an immutable object such as `Rational`, this means that you must ensure the data is valid when the object is constructed (and, ensure that the object is actually immutable so the data can't be changed to an invalid state later). Given that a zero denominator is an

invalid state for Rational, you must not let a Rational be constructed if a zero is passed in the d parameter.

The best way to approach this problem is to define as a *precondition* of the primary constructor that d must be non-zero. A precondition is a constraint on values passed into a method or constructor, a requirement which callers must fulfill. One way to do that is to use the require method,¹ like this:

```
class Rational(n: Int, d: Int) {  
    require(d != 0)  
    override def toString = n + "/" + d  
}
```

The require method takes one boolean parameter. If the passed value is true, require will return normally. Otherwise, require will prevent the object from being constructed by throwing an IllegalArgumentException.

6.5 Adding fields

Now that the primary constructor is properly enforcing its precondition, we will turn our attention to supporting addition. To do so, we'll define a public add method on class Rational that takes another Rational as a parameter. To keep Rational immutable, the add method must not add the passed rational number to itself. Rather, it must create and return a brand new Rational that holds the sum. You might think you could write add this way:

```
class Rational(n: Int, d: Int) { // This won't compile  
    require(d != 0)  
    override def toString = n + "/" + d  
    def add(that: Rational): Rational =  
        new Rational(n * that.d + that.n * d, d * that.d)  
}
```

Unfortunately, given this code the compiler will complain:

```
<console>:11: error: value d is not a member of Rational
```

¹The require method is defined in a standalone object named Predef in package scala, whose members are imported automatically into every Scala source file.

```
new Rational(n * that.d + that.n * d, d * that.d)  
^  
<console>:11: error: value d is not a member of Rational  
      new Rational(n * that.d + that.n * d, d * that.d)  
      ^
```

Although class parameters `n` and `d` are in scope in the code of your `add` method, you can only access their value on the object on which `add` was invoked. Thus, when you say `n` or `d` in `add`'s implementation, the compiler is very happy to provide you with the values for these class parameters. But it won't let you say `that.n` or `that.d`, because `that` does not refer to the `Rational` object on which `add` was invoked.² To access `n` and `d` on `that`, you'll need to make `n` and `d` into fields. Here's how you could add the necessary fields:

```
class Rational(n: Int, d: Int) {  
    require(d != 0)  
    val numer: Int = n  
    val denom: Int = d  
    override def toString = numer + "/" + denom  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
}
```

In this version of `Rational`, we added two fields named `numer` and `denom`, and initialized them with the values of class parameters `n` and `d`.³ We also changed the implementation of `toString` and `add` so that they use the fields, not the class parameters. As a result, the compiler will happily let you use this class to add two rational numbers:

```
scala> val oneHalf = new Rational(1, 2)
```

²Actually, you could add a `Rational` to itself, in which case `that` would refer to the object on which `add` was invoked. But because you can pass any `Rational` object to `add`, the compiler still won't let you say `that.n`.

³Even though `n` and `d` are used in the body of the class, given they are only used inside constructors, the Scala compiler will not emit fields for them. Thus, given this code the Scala compiler will generate a class with two `Int` fields, one for `numer` and one for `denom`.

```
oneHalf: Rational = 1/2
scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3
scala> oneHalf add twoThirds
res0: Rational = 7/6
```

One other thing you can do now that you couldn't do before is access the numerator and denominator values from outside the object. Simply access the public numer and denom fields, like this:

```
scala> val r = new Rational(1, 2)
r: Rational = 1 / 2
scala> r.numer
res7: Int = 1
scala> r.denom
res8: Int = 2
```

6.6 Self references

As in Java, the keyword `this` refers to the object instance on which the currently executing method was invoked, or if used in a constructor, the object instance being constructed. 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!

6.7 Auxiliary constructors

Sometimes you need multiple constructors in a class. Constructors other than the primary constructor are called *auxiliary constructors* in Scala. For example, a rational number with a denominator of 1 can be written more succinctly as simply the numerator. Instead of $\frac{5}{1}$, for example, you can just write 5. It might be nice, therefore, if instead of writing `Rational(5, 1)`, client programmers could simply write `Rational(5)`. This would require adding an auxiliary constructor to `Rational` that takes only one argument, the numerator, with the denominator predefined to be 1. Here's what that would look like:

```
class Rational(n: Int, d: Int) {  
    require(d != 0)  
    val numer: Int = n  
    val denom: Int = d  
    def this(n: Int) = this(n, 1)  
    override def toString = numer + "/" + denom  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
}
```

Auxiliary constructors in Scala start with `def this(...)`. The body of `Rational`'s auxiliary constructor merely invokes the primary constructor, passing along its lone argument, `n`, as the numerator and 1 as the denominator. If you type the following into the interpreter:

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

you should see:

```
y: Rational = 3/1
```

In Scala, every auxiliary constructor must invoke another constructor of the same class as its first action. In other words, the first statement in every auxiliary constructor in every Scala class will have the form “`this(...)`.” The invoked constructor is either the primary constructor (as in the `Rational` example), or another auxiliary constructor that comes textually before the calling constructor. The net effect of this rule 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.

If you’re familiar with Java, you may wonder why Scala’s rules for constructors are a bit more restrictive than Java’s. In Java, a constructor must either invoke another constructor of the same class, or directly invoke a constructor of the superclass, as its first action.⁴ In a Scala class, only the primary constructor can invoke a superclass constructor. The increased restriction in Scala is really a design tradeoff that needed to be paid in exchange for the greater conciseness and simplicity of Scala’s constructors compared to Java’s.

6.8 Private fields and methods

In the previous version of `Rational`, we simply initialized `numer` with `n` and `denom` with `d`. As a result, the numerator and denominator of a `Rational` can be larger than needed. For example, the fraction, $\frac{66}{42}$, could be normalized to an equivalent reduced form, $\frac{11}{7}$, but `Rational`’s primary constructor doesn’t currently do this:

```
scala> new Rational(66, 42)
res15: Rational = 66/42
```

To normalize in this way, you need to divide the numerator and denominator by their *greatest common divisor*. For example, the greatest common divisor of 66 and 42 is 6. (In other words, 6 is the largest integer that divides evenly into both 66 and 42.) Dividing both the numerator and denominator of $\frac{66}{42}$ by 6 yields its reduced form, $\frac{11}{7}$. Here’s how you could accomplish that in Scala:

⁴If you’re not familiar with Java, or don’t know what a superclass is at this point, do not worry. Superclasses and the details of how constructor invocation and inheritance interact will be explained in [Chapter 10](#).

```
class Rational(n: Int, d: Int) {  
    require(d != 0)  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
    def this(n: Int) = this(n, 1)  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
    override def toString = numer+"/"+denom  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

In this version of Rational, we added a private field, g, and modified the initializers for numer and denom. Because g is private, it can be accessed inside the body of the class, but not outside. We also added a private method, gcd, which calculates the greatest common divisor of two passed Ints. For example, gcd(12, 8) is 4. To make a field or method private you simply place the `private` keyword in front of its definition. The purpose of the private “helper method” gcd is to factor out code needed by some other part of the class, in this case, the primary constructor. To ensure g is always positive, we pass the absolute value of n and d, which we obtain by invoking `abs` on them, a method you can invoke on any `Int` to get its absolute value. The Scala compiler will place the code for the initializers of Rational’s three fields into the primary constructor in the order in which they appear in the source code. Thus, g’s initializer, `gcd(n.abs, d.abs)`, will execute before the other two, because it appears first in the source. Field g will be initialized with the result, the greatest common divisor of the absolute value of the class parameters, n and d. Field g is then used in the initializers of numer and denom. By dividing n and d (after being normalized so that the denominator is positive) by their greatest common divisor, g, every Rational will be constructed in its normalized form:

```
scala> new Rational(66, 42)
```

```
res24: Rational = 11/7
```

6.9 Defining operators

The current implementation of Rational addition is OK as it stands, but it could be made more convenient to use. You might ask yourself why you can write:

`x + y`

if `x` and `y` are integers or floating-point numbers, but you have to write:

`x.add(y)`

or at least:

`x add y`

if they are rational numbers. There's no convincing reason why this should be so. Rational numbers are numbers just like other numbers. 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'll show you how.

The first step is to replace `add` by the usual mathematical symbol. This is straightforward, as `+` is a legal identifier in Scala. You can simply define a method with `+` as its name. While you're at it, you may as well implement a method named `*` that performs multiplication:

```
class Rational(n: Int, d: Int) {  
    require(d != 0)  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
    def this(n: Int) = this(n, 1)  
    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)  
  
override def toString = numer+ "/" +denom  
  
private def gcd(a: Int, b: Int): Int =  
    if (b == 0) a else gcd(b, a % b)  
}
```

With class Rational defined in this manner, you can now write:

```
scala> val x = new Rational(1, 2)  
x: Rational = 1/2  
  
scala> val y = new Rational(2, 3)  
y: Rational = 2/3  
  
scala> x + y  
res32: Rational = 7/6
```

As always, the operator syntax on the last input line is equivalent to a method call. You could also have written:

```
scala> x.+(y)  
res33: Rational = 7/6
```

but this would not have been as readable.

Another thing to note is that given Scala's rules for operator precedence, which were described in [Section 5.8](#), the `*` method will bind more tightly than the `+` method for `Rationals`. In other words, expressions involving `+` and `*` operations on `Rationals` will behave as expected. For example, `x + x * y` will execute as `x + (x * y)`, not `(x + x) * y`:

```
scala> x + x * y  
res34: Rational = 5/6  
  
scala> (x + x) * y  
res35: Rational = 2/3  
  
scala> x + (x * y)  
res36: Rational = 5/6
```

6.10 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 in this section.

An *alphanumeric identifier* starts with a letter or underscore, which can be followed by further letters, digits, or underscores. 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 ‘\$’ characters, even though it will compile; if they do this might lead to name clashes with identifiers generated by the Scala compiler.

Scala follows Java’s convention of camel-case identifiers, such as `toString` and `HashSet`.⁵ Although underscores are legal in identifiers, they are not used that often in Scala programs, in part to be consistent with Java, but also because underscores have many other non-identifier uses in Scala code. As a result, it is best to avoid identifiers like `to_string`, `__init__`, or `name_`.⁶ Camel-case names of fields, method parameters, local variables, and functions should start with lower case letter, for example: `length`, `flatMap`, and `s`. Camel-case names of classes and traits should start with an upper case letter, for example: `BigInt`, `List`, and `UnbalancedTreeMap`.⁷

One way in which Scala’s conventions depart from Java’s involves constant names. In Scala, the word *constant* does not just mean `val`. Even though a `val` does remain constant after it is initialized, it is still a variable. For example, method parameters are `vals`, but each time the method is called those `vals` can hold different values. A constant is more permanent. For example, `scala.Math.Pi` is defined to be the double value closest to the real value of π , the ratio of a circle’s circumference to its diameter. This value is unlikely to change ever, thus, `Pi` is clearly a constant. You can use also constants to give names to values that would otherwise be *magic numbers* in

⁵This style of naming identifiers is called *camel case* because the identifiers have humps consisting of the embedded capital letters.

⁶One consequence of using a trailing underscore is that if you attempt, for example, to write a declaration like this, “`val name_ : Int = 1,`” you’ll get a compiler error. The compiler will think you are trying to declare a `val` named “`name_`”. To get this to compile, you would need to insert an extra space before the colon, as in: “`val name_ : Int = 1,`”

⁷In Chapter 16.5, you’ll see that sometimes you may want to give a special kind of class known as a *case class* a name consisting solely of operator characters. For example, the Scala API contains a class named `:_`, which facilitates pattern matching on `Lists`.

your code: literal values with no explanation, which in the worst case appear in multiple places. You may also want to define constants for use in pattern matching, a use case that will be described in [Section 15.2](#). In Java, the convention is to give constants names that are all upper case, with underscores separating the words, such as MAX_VALUE or PI. In Scala, the convention is merely that the first character must be upper case. Thus, constants named in the Java style, such as X_OFFSET, will work as Scala constants, but the Scala convention is to use camel case for constants, such as XOffset.

An *operator identifier* consists of one or more operator characters. Operator characters are printable ASCII characters such as +, :, ?, ~ or #.⁸ Here are some examples of operator identifiers:

+ ++ ::: <?> :->

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 difference 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. For example, unary_+ used as a method name defines a unary ‘+’ operator. Or, myvar_= used as method name defines an assignment operator. In addition, the mixed identifier form myvar_= is generated by the Scala compiler to support *properties*; more on that in [Chapter 18](#).

A *literal identifier* is an arbitrary string enclosed in back ticks ` ... ` . Some examples of literal identifiers are:

⁸More 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, parentheses, square brackets, curly braces, single or double quote, or an underscore, period, semi-colon, comma, or back tick character.

```
`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 works 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`()`.

6.11 Method overloading

Back to class `Rational`. With the latest changes, you can now do addition and multiplication 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`. You must write `r * new Rational(2)`, which is not as nice.

To make `Rational` even more convenient, you can add new methods to the class that perform mixed addition and multiplication on rationals and integers. While you're at it, you may as well add methods for subtraction and division too:

```
class Rational(n: Int, d: Int) {  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def +(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    def +(i: Int): Rational =  
        new Rational(numer + i * denom, denom)  
  
    def -(that: Rational): Rational =
```

```
new Rational(  
    numer * that.denom - that.numer * denom,  
    denom * that.denom  
)  
  
def -(i: Int): Rational =  
    new Rational(numer - i * denom, denom)  
  
def *(that: Rational): Rational =  
    new Rational(numer * that.numer, denom * that.denom)  
  
def *(i: Int): Rational =  
    new Rational(numer * i, denom)  
  
def /(that: Rational): Rational =  
    new Rational(numer * that.denom, denom * that.numer)  
  
def /(i: Int): Rational =  
    new Rational(numer, denom * i)  
  
override def toString = numer+"/"+denom  
  
private def gcd(a: Int, b: Int): Int =  
    if (b == 0) a else gcd(b, a % b)  
}
```

There are now two versions each of the arithmetic methods: one that takes a rational as its argument and another that takes an integer. In other words, each of these method names is *overloaded*, because each name is now being used by multiple methods. For example, the name + is used by one method that takes a Rational and another that takes an Int. In a method call, the compiler picks the version of an overloaded method that correctly matches the types of the arguments. For instance, if the argument y in x.+(y) is a Rational, the compiler will pick the method + that takes a Rational parameter. But if the argument is an integer, the compiler will pick the method + that takes an Int parameter instead. You can try this out in the interpreter:

```
scala> val x = new Rational(2, 3)  
x: Rational = 2/3  
  
scala> x * x  
res37: Rational = 4/9  
  
scala> x * 2  
res38: Rational = 4/3
```

Scala's process of overloaded method resolution is very similar to Java's. In every case, the chosen overloaded version is the one that 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.

6.12 Implicit conversions

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 that takes a `Rational` argument—it couldn't because class `Rational` 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 that 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
```

Note that for an implicit conversion to work, it needs to be in scope. If you place the implicit method definition inside class `Rational`, it won't be in scope in the interpreter. For now, you'll need to define it directly in the interpreter.

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, including ways to bring them into scope where they are needed, in [Chapter 21](#).

6.13 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 31](#), 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 written 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. Conciseness will often be a big part of that readability, but you can take conciseness too far. By designing libraries that enable tastefully concise, readable, understandable client code, you can help those client programmers work productively.

6.14 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 this chapter that defining classes and using objects that have no mutable state is a quite natural way to code in Scala.

In general, immutable objects have several advantages and one potential disadvantage. First, immutable objects are often easier to reason about than mutable ones, because they do not have complex state spaces that change over time. Second, you can pass immutable objects around quite freely, whereas you may need to make defensive copies of mutable objects before passing them to other code. Third, there is no way for two threads concurrently accessing an immutable to corrupt its state once it has been properly constructed, because no thread can make any change at all to the state of an immutable. Fourth, immutable objects make safe hashtable keys. If a mutable object is mutated after it is placed into a `HashSet`, for example, that object may not be found the next time you look into the `HashSet`.

The only disadvantage of immutable objects is that they sometimes demand large object graphs to be copied where otherwise the update could be done in place. In some cases this can be awkward to express and it might also cause a performance bottleneck. As a result, it is not uncommon for libraries to provide a mutable alternative which makes it easier to change some elements in the middle of a large data structure. For example, you have as alternative to Scala's immutable `List` class the mutable classes `ListBuffer` and `Array`. We'll give you more information on designing mutable objects in Scala in [Chapter 18](#).

Although the final version of `Rational` shown in this chapter fulfills the requirements set forth at the beginning of the chapter, it could still be improved. We will in fact return to this example later in the book, as you learn aspects of Scala that you can use to make `Rational` better. For example, in [Chapter 28](#), you'll learn how to override `equals` and `hashcode` to allow `Rationals` to behave better when compared with `==` or placed into hash tables. In [Chapter 21](#), you'll learn how to place implicit method definitions in a companion object for `Rational`, so they can be more easily placed into scope when client programmers are working with `Rationals`.

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 expres-

sion 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 whether 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 built-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 Longs:

```
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 != null)
```

As mentioned previously, both while and do loops result in (), the unit value. One other construct that results in the unit value, which is relevant here, is reassignment to vars. For example, were you attempt to read lines in Scala using the following while loop idiom from Java (and C and C++), you'll run into trouble:

```
var line = ""  
while ((line = readLine()) != null)  
    println("Read: " + line)
```

When you compile this code, Scala will give you a warning that comparing values of type Unit and Null using != will always yield true. Whereas

in Java, assignment results in the value assigned, in this case a line from the standard input, in Scala assignment always results in the unit value, (). Thus, the value of the assignment “`line = readLine()`” will always be () and never null. As a result, this while loop’s condition will never be false, and the loop will, therefore, never terminate.

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 (y == 0) x else gcd(y, x % y)
```

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.

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.

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 120](#). 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 4)
|   println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
```

¹To be precise, the expression to the right of the <- symbol must support a method named `foreach`

Iteration 4

If you don't want to include the upper bound of the range in the values that are iterated over, use `until` instead of `to`:

```
scala> for (i <- 1 until 4)
|   println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
```

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 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 semicolons 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 variable bindings

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 new 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!  
}
```

for-yield is useful, because it lets you compute complex data structures without having to use temporary variables or producing side effects. Take the `printMultiTable` method shown on [page 90](#) as an example. Instead of printing out the elements of the table one by one, it's also possible (and even more concise) to construct the table as a string, using a for-yield:

```
val 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  
    }  
    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 11.3](#).

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 15](#).

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, 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.

²One difference from Java that you'll quickly notice in Scala is that unlike Java, Scala 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 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")
  }
```

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; they should not normally change the value computed in the main body or a `catch` clause 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 built-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 15](#). 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 cases 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 `break`s 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`,

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.

7.7 Conclusion

Scala's built-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 object, there are also functions nested within functions, function literals, and function values. This chapter takes you on a tour through all of these flavors of 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
Approximate.scala: def isGoodEnough(guess: Double): Boolean ...
Approximate.scala: Math.abs(guess*guess - 2.0) < 1.0E- ...
Approximate.scala: def approximate(guess: Double): Double ...
Approximate.scala: def approximateLoop(initialGuess: Double ...
LongLines.scala: def processFile(filename: String, width: ...
...
```

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 Local 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.

There is just one problem with the “many small functions” approach. All these helper function names have a tendency to pollute the program namespace. In the interpreter shell this not so much of a problem, but once functions are packaged in reusable classes and objects, its desirable to hide the helper functions from clients of a class. They often do not make sense individually, and you often want to keep enough flexibility to delete the helper methods if you later rewrite the class a different way.

In Java, your main tool for this purpose is the `private` method. This private-method approach works in Scala as well, as described in [Chapter ??](#), but Scala offers an additional approach: you can define functions inside other functions. Just like local variables, such `local 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 local functions can access the parameters of their enclosing function. 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)  
    }  
}
```

Simpler, isn't it? This use of an enclosing function's parameters is a common and useful example of the general nesting Scala provides. The nesting and scoping described in [Section 4.3](#) applies to all Scala constructs, including functions. It's a simple principle, but very powerful, especially in a language with first-class functions.

8.3 First-class functions

Scala has *first-class functions*. Not only can you define functions and call them, but you can write down functions as unnamed *literals* and then pass them around as *values*. We introduced function literals in [Chapter 2](#) and showed the basic syntax in [Figure 2.2 on page 60](#).

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. The distinction is much like that between classes (source code) and objects (runtime).

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`.

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.

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 usual parentheses 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
```

Because `increase`, in this example, is a `var`, you can reassign it a different function value later on.

```
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 its body by curly braces and put one statement per line, thus forming a `block`. 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
```

So now you have seen the nuts and bolts of function literals and function values. Careful library writers will give you a lot of opportunities to use them. For example, a `foreach` method is available for all collections. 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)
```

Methods like `foreach` and `filter` are described further later in the book. [Chapter 16](#) talks about their use in class `@List@`, and [Chapter 17](#) talks about their use with other collection types than `List`.

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*, because the targeted usage of an expression—in this case an argument to `someNumbers.filter()`—is allowed to influence the typing of that expression—in this case to determine the type of the `x` parameter. The precise details of target typing are not important for you to learn. You can simply start by writing a function literal without the argument type, and, if the compiler gets confused, add in the type. Over time you’ll get a feel for which situations the compiler can and cannot puzzle out.

A second way to remove useless characters is to leave out parentheses around parameter whose type is inferred. In the previous example, the parentheses around `(x)` are extraneous:

```
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)
res9: 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 162](#) 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)
res10: 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, suppose you write `_ + _` by itself:

```
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)
res11: Int = 15
```

Note that `_ + _` expands 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. Multiple underscores mean multiple parameters, not reuse of a single parameter repeatedly. 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, rather than writing `println(_)`, you could 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)
res12: 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)
res13: 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.² 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)
res14: 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 `def` into a function value. For example, if you have a local 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` 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

²The generated class extends trait `Function3`, which declares the three-arg `apply` method.

generates a new function class whose `apply` method takes one argument. When invoked with that one argument, this function's `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 162](#), 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)
res15: 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 type³ 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)
```

Had Scala adopted the classical functional tradition that partially applied functions are OK everywhere, this code would type check. 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

³Scala's universal root type, `Any`, will be described in Chapter 10.

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)
res17: 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: Int) => 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 not a closure in the strictest sense, because `(x: Int) => x + 1` is already closed as written. But any function literal with free variables, such as `(x: Int) => x + more`, is an open term. Therefore, any function value created at runtime from `(x: Int) => x + 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)
res19: Int = 10009
```

Intuitively, Scala's closures capture variables themselves, not the value to which variables refer.⁴ 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.

```
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
res21: 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?

⁴By contrast, Java's inner classes do not allow you to access modifiable variables in surrounding scopes at all, and so there is no difference between capturing a variable and capturing its currently held value.

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)
res22: Int = 11
scala> inc9999(10)
res23: Int = 10009
```

It makes no difference that the `more` in this case is a parameter to a method call that has already returned. The Scala compiler rearranges things in cases like this so that the captured parameter lives out on the heap, instead of the stack, and thus can outlive the method call that created it. This rearrangement is all taken care of automatically, so you don’t have to worry about it. Capture any variable you like, `val`, `var`, or parameter.

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?
```

This notation tells the compiler to pass each element of `arr` as its own argument to `echo`, rather than all of it as a single argument.

8.9 Tail recursion

In [Section 7.2](#) on page 143, 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.⁵ 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: Double) : Double =
  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:

```
def approximateLoop(initialGuess: Double): Double = {
  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.⁶ 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,

⁵A recursive function is one that calls itself.

⁶The benefit of avoiding vars was described in [Step 7](#) on page 76.

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 double approximate(double);  
Code:  
 0:  aload_0  
 1:  astore_3  
 2:  aload_0  
 3:  dload_1  
 4:  invokevirtual #24; //Method isGoodEnough:(D)Z  
 7:  ifeq    12  
10:  dload_1  
11:  dreturn  
12:  aload_0  
13:  dload_1  
14:  invokevirtual #27; //Method improve:(D)D  
17:  dstore_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:

```
def boom(x: Int): Int =  
  if (x == 0) throw new Exception("boom!")  
  else boom(x - 1) + 1
```

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:

```
def bang(x: Int): Int =  
  if (x == 0) throw new Exception("bang!")  
  else bang(x - 1)
```

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:nottailcalls
```

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 .<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. Scala only optimizes directly recursive calls back to the same function making the call. 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 given you a grand tour of functions in Scala. In addition to methods, Scala provides local functions, function literals, and function values. In addition to normal function calls, Scala provides partially applied functions and functions with repeated parameters. When possible, function calls are implemented as optimized tail calls, and thus many nice-looking recursive functions run just as quickly as hand-optimized versions that use `while` loops.

The next chapter will build on these foundations and show how Scala's rich support for functions helps you abstract over control.

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 `FileMatcher`, 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 some dynamic languages, but Scala does not allow pasting together code at runtime 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. Take a look, now, 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 three searching methods from before 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 you may as well 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

The previous example has demonstrated that higher-order functions can help reduce code duplication as you implement an API. Another important use of higher-order functions is to put them in an API itself to make client code more concise. A good example is provided by the special-purpose looping methods of Scala's collection types.¹ Many of these are listed in [Table 3.1](#) in [Chapter 3](#), but take a look at just one example for now to see why these methods are so useful.

Consider `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 negative number:

```
def containsNeg(nums: List[Int]): Boolean = {
```

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`. See [Chapter 17](#) for a discussion.

```
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> containsNeg(List(1, 2, 3, 4))
res0: Boolean = false

scala> containsNeg(List(1, 2, -3, 4))
res1: 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 containsNeg(nums: List[Int]) = nums.exists(_ < 0)
```

This version of `containsNeg` yields the same results as the previous:

```
scala> containsNeg(Nil)
res2: Boolean = false

scala> containsNeg(List(0, -1, -2))
res3: Boolean = true
```

The `exists` method represents a control abstraction. It is a special-purpose looping construct provided by the Scala library rather than 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 `containsOdd` method, to test whether a list contains odd numbers, you might write it like this:

```
def containsOdd(nums: List[Int]): Boolean = {
  var exists = false
  for (num <- nums)
    if (num % 2 == 1)
```

```
exists = true
exists
}
```

If you compare the body of `containsNeg` with that of `containsOdd`, you'll find that everything is repeated except that a 0 is changed to a 1. Using `exists`, you could write this instead:

```
def containsOdd(nums: List[Int]) = nums.exists(_ % 2 == 1)
```

The body of the code in this version is again identical to the body of the corresponding `containsNeg` 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.

There are many other looping methods in Scala's standard library. As with `exists`, they can often shorten your code if you recognize opportunities to use them.

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, non-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)
res4: 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)
res5: 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)
res6: Int = 3
```

These `first` and `second` functions are just an illustration of the currying process. They are not directly connected to the `curriedSum` function. Nevertheless, 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)
res7: 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)
res8: Int = 4
```

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.

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)
res9: 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. Earlier in the chapter you saw `filesMatching`, a very specialized control pattern. Consider now a more widely used coding pattern: “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:

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.

```
def withPrintWriter(file: File, op: PrintWriter => Unit) {  
    val writer = new 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 in which you're passing in exactly one argument, you can opt to use curly braces to surround the argument 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`. This curly braces technique will work, however, only if you're passing in one argument. Here's an attempt at violating that rule:

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

```
scala> g.substring { 7, 9 }
<console>:1: error: ';' expected but ',' found.
           g.substring { 7, 9 }
                           ^
```

Because you are attempting to pass in two arguments to `substring`, you get an error when you try to surround those arguments with curly braces. Instead, you'll need to use parentheses:

```
scala> g.substring(7, 9)
res11: java.lang.String = wo
```

The purpose of this ability to substitute curly braces for parentheses for passing in one argument is to enable client programmers to write function literals between curly braces. This can make a method call feel more like a control abstraction. Take the `withPrintWriter` method defined previously as an example. In its most recent form, `withPrintWriter` takes two arguments, so you can't use curly braces. Nevertheless, because the function passed to `withPrintWriter` is the last argument in the list, you can use currying to pull the first argument, the `File`, into a separate argument list. This will leave the function as the lone parameter of the second argument list. Here's how you'd need to define `withPrintWriter`:

```
def withPrintWriter(file: File)(op: PrintWriter => Unit) {
    val writer = new PrintWriter(file)
    try {
        op(writer)
    } finally {
        writer.close()
    }
}
```

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 186](#), 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:

```
var assertionsEnabled = true  
  
def myAssert(predicate: () => Boolean) =  
    if (assertionsEnabled && !predicate())  
        throw new AssertionError
```

The definition is fine, but using it is a little bit awkward:

³You'll call this `myAssert`, not `assert`, because Scala provides an `assert` of its own, which will be described in [Section 14.1](#) on page 257.

```
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 AssertionError
```

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 AssertionError
```

This formulation is also legal, of course, and the code using this version of `boolAssert` would still look exactly as before:

```
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 "`x / 0 == 0`" will yield an exception in `boolAssert`'s case:

```
scala> var assertionsEnabled = false
assertionsEnabled: Boolean = false
scala> boolAssert(x / 0 == 0)
java.lang.ArithmetricException: / by zero
    at .<init>(<console>:8)
    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(x / 0 == 0)
```

9.6 Conclusion

This chapter has shown you how to build on Scala's rich function support to build control abstractions. You can use functions within your code to factor out common control patterns, and you can take advantage of higher-order functions in the Scala library to reuse control patterns that are common across all programmers' code. This chapter has also shown how to use currying and by-name parameters so that your own higher-order functions can be used with a concise syntax.

In the previous chapter and this one, you have seen quite a lot of information about functions. The next few chapters will go back to discussing more object-oriented features of the language.

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 A two-dimensional layout library

As a running example in this chapter and the next, we’ll create a library for building and rendering two-dimensional layout elements. Each element will represent a rectangle filled with text. For convenience, the library will provide factory methods named “elem” that construct new elements from passed data. For example, you’ll be able to create a layout element containing a string using a factory method with the following signature:

```
elem(s: String): Element
```

As you can see, elements will be modeled with a type named `Element`. You’ll be able to call `above` or `beside` on an element, passing in a second element, to get a new element that combines the two. For example, the following expression 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
```

Layout elements are a good example of a system in which objects can be constructed from simple parts with the aid of composing operators. In this chapter, we'll define classes that enable element objects to be constructed from arrays, lines, and rectangles—the simple parts. We'll also define 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.

10.2 Abstract classes

Our first task is to define type `Element`, which represents layout elements. Since elements are two dimensional rectangles of characters, it makes sense to include a member, `contents`, that 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` will be `Array[String]`. Here's what it will look like:

```
abstract class Element {
    def contents: Array[String]
}
```

In this class, `contents` is declared as a method that has no implementation. In other words, the method is an *abstract* member of class `Element`. A class with abstract members must itself be declared abstract, which is done by writing an `abstract` modifier in front of the `class` keyword:

```
abstract class Element ...
```

The `abstract` modifier signifies that the class may have abstract members that do not have an implementation. As a result, you cannot instantiate an abstract class. If you try to do so, you'll get a compiler error:

```
scala> new Element
<console>:5: error: class Element is abstract; cannot be instantiated
          new Element
                  ^
```

Later in this chapter you'll see how to create subclasses of class `Element`, which you'll be able to instantiate because they fill in the missing definition for `contents`.

Note that the `contents` method in class `Element` does 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 (or allowed) on method declarations. Methods that do have an implementation are called *concrete*.

Another bit of terminology distinguishes between *declarations* and *definitions*. Class `Element` *declares* the abstract method `contents`, but currently *defines* no concrete methods. In the next section, however, we'll enhance `Element` by defining some concrete methods.

10.3 Defining parameterless methods

As a next step, we'll add methods to `Element` that reveal its width and height:

```
abstract class Element {
    def contents: Array[String]
    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 none of `Element`'s three methods has a parameter list, not even an empty one. For example, instead of

```
def width(): Int
```

the method is defined without parentheses:

```
def width: Int
```

Such *parameterless methods* are quite common in Scala. By contrast, methods defined with empty parentheses, such as `@def height(): Int@`, are called *empty-paren methods*. The recommended convention is to use a parameterless method whenever there are no parameters *and* the method neither changes nor depends on mutable state.

This convention supports the *uniform access principle*,¹ which says that client code should not be affected by a decision to implement an attribute as a field or method. For instance, we could have chosen to implement `width` and `height` as fields instead of methods, simply by changing the `def` in each definition to a `val`:

```
abstract class Element {  
    def contents: Array[String]  
    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 each `Element` object. So it depends on the usage profile of a class whether an attribute is better represented as a field or method, 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 so long as the access function is *pure*, *i.e.*, it does not have any side effects and does not depend on mutable state. The client should not need to care either way.

¹Meyer, *Object-Oriented Software Construction* [Mey00]

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 and empty-paren methods. In particular, you can override a parameterless method with an empty-paren method, and *vice versa*. You can also leave off the empty parentheses on an invocation of any function that takes no arguments. 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 parentheses in Scala function calls. However, it is recommended to still write the empty parentheses when the invoked method represents more than a property of its receiver object. For instance, empty parentheses are appropriate if the method performs I/O, or writes reassignable variables (`vars`), or reads `varss` other than the receiver's fields, 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:

```
"hello".length // no () because no side-effect  
println()      // better to not drop the ()
```

To summarize, it is encouraged style in Scala to define methods that take no parameters and have no side effects as parameterless methods, *i.e.*, leaving off the empty parentheses. On the other hand, you should never define a method that has side-effects without parentheses, because then invocations of that method would look like a field selection. So your clients might be surprised to see the side effects. Similarly, whenever you invoke a function that has side effects, be sure to include the empty parentheses when you write the invocation. As described in [Step 3](#) in [Chapter 2](#), another way to think about this is if the function you're calling performs an operation, use the parentheses, but if it merely provides access to a property, leave the parentheses off.

10.4 Extending classes

We still need to be able to create new element objects. You have already seen that “new Element” cannot be used for this because class Element is abstract. To instantiate an element, therefore, we will need to create a subclass that extends Element and implements the abstract contents method. Here’s one possible way to do that:

```
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 after the class name to express this:

```
... extends Element ...
```

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. Given ArrayElement extends Element, class ArrayElement is called a *subclass* of class Element. Conversely, Element is a *superclass* of ArrayElement.

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. Thus, class Element implicitly extends class AnyRef. You can see these inheritance relationships in [Figure 10.1](#).

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 example, the contents method in ArrayElement overrides (or, alternatively: implements) the abstract method contents in class Element.²

²One flaw with this design is that because the returned array is mutable, clients could change it. For now we’ll keep things simple, but in an iteration of ArrayElement in [Chapter 10](#).

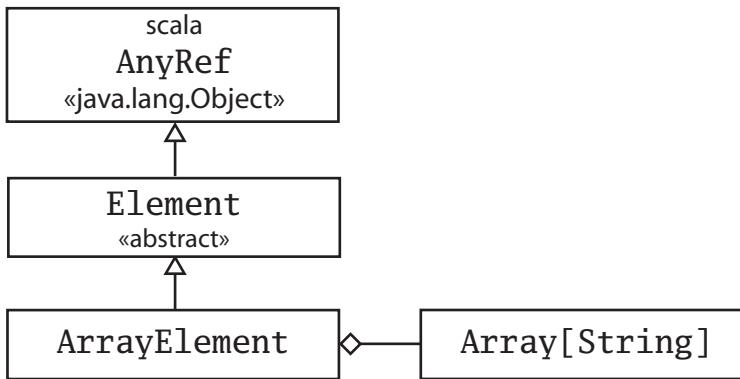


Figure 10.1: Class diagram for `ArrayElement`.

On the other hand, class `ArrayElement` inherits the `width` and `height` methods from class `Element`. For example, given an `ArrayElement` `ae`, you can query its `width` using `ae.width`, just as if `width` were defined in class `ArrayElement`:

```

scala> val ae = new ArrayElement(Array("hello", "world"))
ae: ArrayElement = ArrayElement@d94e60
scala> ae.width
res1: Int = 5

```

Subtyping means that a value of the subclass can be used wherever a value of the superclass is required. For example:

```
val e: Element = new ArrayElement("hello")
```

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`, and as a result, the type `ArrayElement` is compatible with the type `Element`.³

ter 14, we'll solve this problem by returning a *defensive copy* of the array instead. Another problem is we aren't currently ensuring that every `String` element of the `contents` array has the same length. We'll take care of this as well in [Chapter 14](#).

³For more perspective on the difference between subclass and subtype, see the glossary entry for [subtype](#).

Figure 10.1 also shows the *composition* relationship that exists between `ArrayElement` and `Array[String]`. This relationship is called composition because class `ArrayElement` is “composed” out of class `Array[String]`, in that the Scala compiler will place into the binary class it generates for `ArrayElement` a field that holds a reference to the passed `conts` array. We’ll discuss some design considerations concerning composition and inheritance later in this chapter.

10.5 Overriding methods and fields

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 namespace. This makes it possible for a field to 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(conts: Array[String]) extends Element {  
    val contents: Array[String] = conts  
}
```

The field `contents` (defined with a `val`) in this version of `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 namespaces for definitions in place of Java's four. Java's four namespaces are fields, methods, types, and packages. By contrast, Scala's two namespaces are:

- values (fields, methods, packages, and singleton objects)
- 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.⁴

10.6 Defining parametric fields

Consider again the definition of class `ArrayElement` shown in the previous section. 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 sign that there may be some unnecessary redundancy and repetition in your code.

You can avoid the code smell by combining the parameter and the field in a single *parametric 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 that defines at the same time a parameter and field with the same name. Specifically, 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:

⁴The reason that packages share the same namespace 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 246](#).

```
class ArrayElement(x123: Array[String]) extends Element {  
    val contents: Array[String] = x123  
}
```

You can also prefix a class parameter with `var`, in which case 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 you can do for any other class member. Consider for instance the following class definitions:

```
class Cat {  
    val dangerous = false  
}  
class Tiger (  
    override val dangerous: Boolean,  
    private var age: Int  
) extends Cat
```

Tiger's definition is a shorthand for the following alternate class definition with an overriding member `dangerous` and a private member `age`:

```
class Tiger(param1: Boolean, param2: Int) extends Cat {  
    override val dangerous = param1  
    private var age = param2  
}
```

Both members are initialized from the corresponding parameters. We chose the names of those parameters, `param1` and `param2`, arbitrarily. The important thing was that they not clash with any other name in scope.

10.7 Invoking superclass constructors

You now have a complete system consisting of two classes: an abstract class `Element`, which is extended by a concrete class `ArrayElement`. You might also envision other ways to express an element. For example, it might be convenient to make it easy for clients to create a layout element consisting of a single line given by a string. 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 subclasses. For example, here's a `LineElement` class that extends `ArrayElement`:

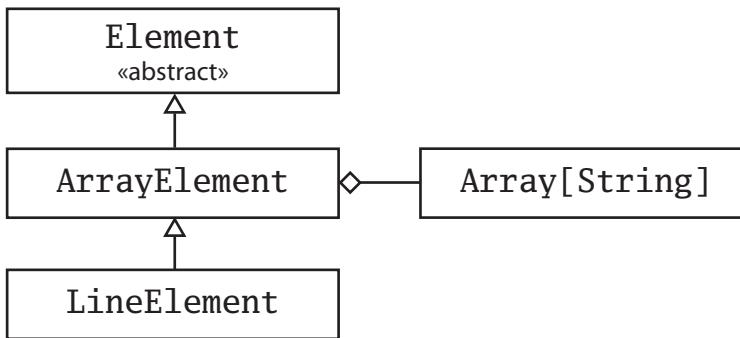


Figure 10.2: Class diagram for LineElement.

```

class LineElement(s: String) extends ArrayElement(Array(s)) {
    override def width = s.length
    override def height = 1
}
  
```

With the new subclass, the inheritance hierarchy for layout elements now looks as shown in [Figure 10.2](#).

Since `LineElement` extends `ArrayElement`, and `ArrayElement`'s constructor takes a parameter (an `Array[String]`), `LineElement` needs to pass an argument to the primary constructor of its superclass. To invoke a superclass constructor, you simply place the argument or arguments you want to pass in parentheses following the name of the superclass. For example, class `LineElement` passes `Array(s)` to `ArrayElement`'s primary constructor by placing it in parentheses after the superclass `ArrayElement`'s name:

```

... extends ArrayElement(Array(s)) ...
  
```

10.8 Using Override modifiers

Note that the definitions of `width` and `height` in `LineElement` carry an `override` modifier. In [Chapter 6](#), you saw this modifier in the definition of a `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 an 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` override concrete definitions in class `Element`, the new definitions need to be flagged with `override`.

This rule provides useful information for the compiler that helps avoid some hard-to-catch errors and makes system evolution safer. For instance, if you happen to misspell the method or accidentally give it a different parameter list, the compiler will respond with an error message:

```
$ scalac LineElement.scala
.../LineElement.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 defined a library of 2D drawing methods. You 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 with this signature:

```
def hidden(): Boolean
```

Your new method will be 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`. Perhaps 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 the most common manifestation of what is called the “fragile base class” problem. The problem is that if you add new members to base classes (which we usually call superclasses) in a class hierarchy, you risk breaking client code.

Scala cannot completely solve the fragile base class problem, but it improves on the situation compared 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 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 your client would get a compile-time error, which is usually much preferable.

10.9 Polymorphism and dynamic binding

You saw in [Section 10.4](#) that a variable of type `Element` could refer to an object of type `ArrayElement`. The name for this phenomenon is *polymorphism*, which means “many shapes” or “many forms.” In this case, `Element` objects can have many forms.⁵ So far, you’ve seen two such forms: `ArrayElement` and `LineElement`. You can create even more forms of `Element` by defining new `Element` subclasses. For example, here’s how you could define a new form of `Element` that has a given width and height and is filled everywhere with a given character:

```
class UniformElement(
    ch: Char,
    override val width: Int,
    override val height: Int
) extends Element {
    private val line = ch.toString * width
    def contents = Array.make(height, line)
}
```

The inheritance hierarchy for class `Element` now looks as shown in [Figure 10.3](#). As a result, Scala will accept all of the following assignments, because the assigning expression’s type conforms to the type of the defined variable:

⁵This kind of polymorphism is called *subtyping polymorphism*. Another kind of polymorphism in Scala, called *universal polymorphism*, is discussed in [Chapter 19](#).

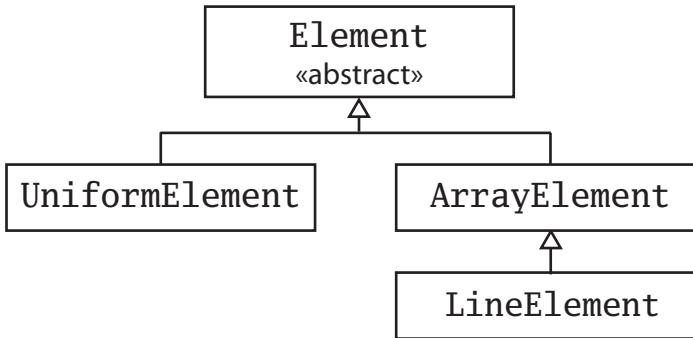


Figure 10.3: Class hierarchy of layout elements

```
val e1: Element = new ArrayElement(Array("hello", "world"))
val ae: ArrayElement = new LineElement("hello")
val e2: Element = ae
val e3: Element = new UniformElement(2, 3, 'x')
```

If you check the inheritance hierarchy, you'll find that in each of these four val definitions, the type of the expression to the right of the equals sign is below the type of the val being initialized to the left of the equals sign.

The other half of the story, however, is that method invocations on variables and expressions are *dynamically bound*. This means that the actual method implementation invoked is determined at run time based on the class of the object, not the type of the variable or expression. To demonstrate this behavior, we'll temporarily remove all existing members from our Element classes, add a method named demo to Element. We'll override demo in in ArrayElement and LineElement, but not in UniformElement:

```
abstract class Element {
    def demo() {
        println("Element's implementation invoked")
    }
}

class ArrayElement extends Element {
    override def demo() {
        println("ArrayElement's implementation invoked")
    }
}
```

```
}

class LineElement extends ArrayElement {
    override def demo() {
        println("LineElement's implementation invoked")
    }
}

// UniformElement inherits Element's demo
class UniformElement extends Element
```

If you enter this code into the interpreter, you can then define this method that takes an `Element` and invokes `demo` on it:

```
def invokeDemo(e: Element) {
    e.demo()
}
```

If you pass an `ArrayElement` to `invokeDemo`, you'll see a message printed indicating `ArrayElement`'s implementation of `demo` was invoked, even though the type of the variable, `e`, on which `demo` was invoked is `Element`:

```
scala> invokeDemo(new ArrayElement)
ArrayElement's implementation invoked
```

Similarly, if you pass a `LineElement` to `invokeDemo`, you'll see a message that indicates `LineElement`'s `demo` implementation was invoked:

```
scala> invokeDemo(new LineElement)
LineElement's implementation invoked
```

The behavior when passing a `UniformElement` may at first glance look suspicious, but it is correct:

```
scala> invokeDemo(new UniformElement)
Element's implementation invoked
```

Because `UniformElement` does not override `demo`, it inherits the implementation of `demo` from its superclass, `Element`. Thus, `Element`'s implementation is the correct implementation of `demo` to invoke when the class of the object is `UniformElement`.

10.10 Declaring final members

Sometimes when designing an inheritance hierarchy, you want to ensure that a member cannot be overridden by subclasses. In Scala, as in Java, you do this by adding a `final` modifier to the member. For example, you could place a `final` modifier on `ArrayElement`'s `demo` method, like this:

```
class ArrayElement extends Element {  
    final override def demo() {  
        println("ArrayElement's implementation invoked")  
    }  
}
```

Given this version of `ArrayElement`, an attempt to override `demo` in its subclass, `LineElement`, would not compile:

```
error: error overriding method demo in class ArrayElement of type ()Unit;  
method demo cannot override final member  
override def demo() {
```

You may also at times want to ensure that an entire class not be subclassed. To do this you simply declare the entire class final by adding a `final` modifier to the class declaration. For example, here's how you would declare `ArrayElement` final:

```
final class ArrayElement extends Element {  
    override def demo() {  
        println("ArrayElement's implementation invoked")  
    }  
}
```

With this version of `ArrayElement`, any attempt at defining a subclass would fail to compile:

```
error: illegal inheritance from final class  
class LineElement(s: String) extends ArrayElement(Array(s)) {
```

We'll now remove the `final` modifiers and `demo` methods, go back in the earlier implementation of the `Element` family, and focus our attention in the remainder of this chapter to completing a working version of the layout library.

10.11 Using composition and inheritance

Composition and inheritance are two ways to define a new class in terms of another existing class. If what you’re after is primarily code reuse, you should in general prefer composition to inheritance. Only inheritance suffers from the fragile base class problem, in which you can inadvertently break subclasses by changing a superclass.

One question you can ask yourself about an inheritance relationship is whether it models an *is-a* relationship.⁶ For example, it would be reasonable to say that `ArrayElement` *is-an* `Element`. Another question you can ask is whether clients will want to use the subclass type as a superclass type.⁷ In the case of `ArrayElement`, we do indeed expect clients will want to use an `ArrayElement` as an `Element`.

Now, if you ask these questions about the inheritance relationships shown in Figure 10.3, do any of the relationships seem suspicious? In particular, does it seem obvious to you that a `LineElement` *is-an* `ArrayElement`? Do you think clients would ever need to use a `LineElement` as an `ArrayElement`? In fact, we defined `LineElement` as a subclass of `ArrayElement` primarily to reuse `ArrayElement`’s definition of `contents`. Perhaps it would be better, therefore, to define `LineElement` as a direct subclass of `Element`, like this:

```
class LineElement(s: String) extends Element {  
    val contents = Array(s)  
    override def width = s.length  
    override def height = 1  
}
```

Given this implementation of `LineElement`, the inheritance hierarchy for `Element` now looks as shown in Figure 10.4.

10.12 Implementing `above`, `beside`, and `toString`

As a next step, we’ll 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:

⁶Meyers, *Effective C++* [Mey91]

⁷Eckel, *Thinking in Java* [Eck98]

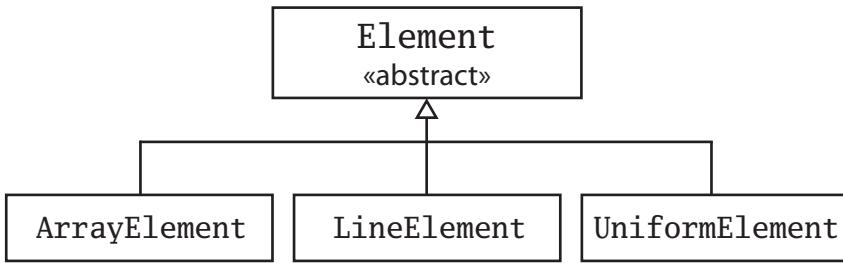


Figure 10.4: Class hierarchy with revised LineElement.

```

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. Some other array methods will be explained in this chapter, and a comprehensive list of all methods will be given in [Chapter 17](#).

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. To keep things simple in this chapter, however, we’ll leave this as is and only pass elements of the same length to `above`. In [Chapter 14](#), we’ll make an enhancement to `above` so that clients can use it to combine elements of different widths.

The next method to implement is `beside`. To put two elements beside each other, we’ll create a new element in which every line results from concatenating corresponding lines of the two elements. As before, to keep things simple we’ll start by assuming the two elements have the same height. This leads to the following design of method `beside`:

```

def beside(that: Element): Element = {
  val contents = new Array[String](this.contents.length)
  for (i <- 0 until this.contents.length)
    contents(i) = this.contents(i) + that.contents(i)
  new ArrayElement(contents)
}
  
```

The `beside` method first allocates a new array, `contents`, and fills it with the concatenation of the corresponding array elements in `this.contents` and `that.contents`. It finally produces a new `ArrayElement` containing `contents`.

Although this implementation of `beside` works, it is in an imperative style, the telltale sign of which is the loop in which we index through arrays. The method could alternatively be abbreviated to one expression:

```
new ArrayElement(  
    for ((line1, line2) <- this.contents zip that.contents)  
        yield line1 + line2  
)
```

Here, the two arrays `this.contents` and `that.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")
```

will evaluate to:

```
Array((1, "a"), (2, "b"))
```

If one of the two operand arrays is longer than the other, `zip` will drop the remaining elements. In the expression above, the third element of the left operand, 3, 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. Scala's pattern-matching system will be described in detail in [Chapter 15](#). For now, you can just think of this as a way to define two `vals`, `line1` and `line2`, for each step of the iteration.

The `for` expression has a `yield` part and therefore yields 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 of this code is the same as in the first version of `beside`, but it is obtained in a purely functional, less error-prone 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.

With the addition of these three methods, class `Element` now looks like this:

```
abstract class Element {  
  
    def contents: Array[String]  
  
    def width: Int =  
        if (height == 0) 0 else contents(0).length  
  
    def height: Int = contents.length  
  
    def above(that: Element): Element =  
        new ArrayElement(this.contents ++ that.contents)  
  
    def beside(that: Element): Element =  
        new ArrayElement(  
            for ((line1, line2) <- this.contents zip that.contents)  
                yield line1 + line2  
        )  
  
    override def toString = contents mkString "\n"  
}
```

10.13 Defining a factory object

You now have a hierarchy of classes for layout elements. This hierarchy could be presented to your clients “as is.” But you might also choose to hide the hierarchy behind 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. This hiding will both make your library simpler for clients to understand, because less detail is exposed, and provide you with more opportunities to change your library’s implementation later without breaking client code.

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.

With the advent of these factory methods, it makes sense to change the implementation of class Element so that it goes through the elem factory methods rather than creating new ArrayElement instances explicitly. To call the factory methods without qualifying them with Element, the name of the singleton object, we will import Element.elem at the top of the source file.

In other words, instead of invoking the factory methods with `Element.elem` inside class `Element`, we'll import `Element.elem` so we can just call the factory methods by their simple name, `elem`. Here's what class `Element` will look like after these changes:

```
import Element.elem

abstract class Element {

    def contents: Array[String]

    def width: Int =
        if (height == 0) 0 else contents(0).length

    def height: Int = contents.length

    def above(that: Element): Element =
        elem(this.contents ++ that.contents)

    def beside(that: Element): Element =
        elem(
            for ((line1, line2) <- this.contents zip that.contents)
                yield line1 + line2
        )

    override def toString = contents mkString "\n"
}
```

In addition, given the factory methods, the three subclasses `ArrayElement`, `LineElement` and `UniformElement` could now be private, because they need no longer be accessed directly by clients. In Scala, you can define classes and singleton objects inside other classes and singleton objects. One way to make the `Element` subclasses private, therefore, is to place them inside the `Element` singleton object and declare them private there. The classes will still be accessible to the three `elem` factory methods, where they are needed. Here's how that will look:

```
object Element {

    private class ArrayElement(val contents: Array[String]) extends Element

    private class LineElement(s: String) extends Element {
        val contents = Array(s)
        override def width = s.length
        override def height = 1
    }
}
```

```
}

private class UniformElement(
    ch: Char,
    override val width: Int,
    override val height: Int
) extends Element {
    private val line = ch.toString * width
    def contents = Array.make(height, line)
}

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)
}
```

With these changes, the layout library is ready for use.

10.14 Putting it all together

A fun way to test almost all elements of the layout library is to write a program that draws a spiral with a given number of edges. This `Spiral` program will do just that:

```
import Element.elem

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]) {
    val errMsg = "Please enter integer number of sides"
    val output =
        if (args.length == 0)
            errMsg
        else
            try {
                val nSides = args(0).toInt
                if (nSides < 1)
                    "Please enter integer greater than zero"
                else
                    spiral(nSides, 0)
            }
            catch {
                case e: NumberFormatException => errMsg
            }
            println(output)
    }
}
```

Because `Spiral` is a standalone object with a `main` method with the proper signature, it is a Scala application. `Spiral` takes one command-line argument, an integer, and draws a spiral with the specified number of edges. For example, you could draw a six-edge spiral like this:

```
$ scala Spiral 6
+----+
|
```

```
| +-+  
| + |  
|   |  
+---+
```

Here's a larger example:

```
$ scala Spiral 17  
+-----+  
|  
| +-----+  
| | | | | | | |
| | +-----+ |  
| | | | |  
| | | +----+ | |  
| | | | | | |  
| | | | | ++ | |  
| | | | | | | |  
| | | | +--+ | |  
| | | | | | |  
| | +-----+ | |  
| | | | |  
| +-----+ |  
| | |  
+-----+
```

We'll work with the layout library again in [Chapter 14](#).

10.15 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.

Chapter 11

Scala’s Hierarchy

Now that you’ve seen the details of class inheritance in the previous chapter, it is a good time to take a step back and look at Scala’s class hierarchy as a whole. In Scala, every class inherits from a common superclass named `Any`. Because every class is a subclass of `Any`, the methods defined in `Any` are “universal” methods: They may be invoked on any object. Scala also defines some interesting classes at the bottom of the hierarchy, `Null` and `Nothing`, which essentially act as common *subclasses*. For example, just as `Any` is a superclass of every other class, `Nothing` is a subclass of every other class. In this chapter, we’ll give you a tour of Scala’s class hierarchy.

11.1 Scala’s class hierarchy

Figure 11.1 shows an outline of Scala’s class hierarchy. At the top of the hierarchy is class `Any`, which defines methods that include the following:

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def hashCode: Int
def toString: String
```

Because every class inherits from `Any`, every object in a Scala program 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`, so 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. We’ll show an example later in this chapter.

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 at run time as Java’s primitive values. The instances of these classes are all written as literals in Scala. For example, 42 is an instance of Int, ‘x’ is an instance of Char, and false an instance of Boolean. You cannot create instances of these classes using new. This is enforced by the “trick” that value classes are all defined to be both abstract and final. So if you were to write:

```
scala> new Int
```

you would get:

```
<console>:5: error: class Int is abstract; cannot be
instantiated
      new Int
           ^
```

The other value class, Unit, corresponds roughly to Java’s void type; it is used as the result type of a method that does not otherwise return an interesting result. Unit has a single instance value, which as discussed in [Section 7.1](#), 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 *, and Boolean has methods named || and &&. Value classes also inherit all methods from class Any. You can test this in the interpreter:

```
scala> 42.toString
res4: java.lang.String = 42
scala> 42.hashCode
res5: Int = 42
scala> 42.equals 42
res6: 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 im-

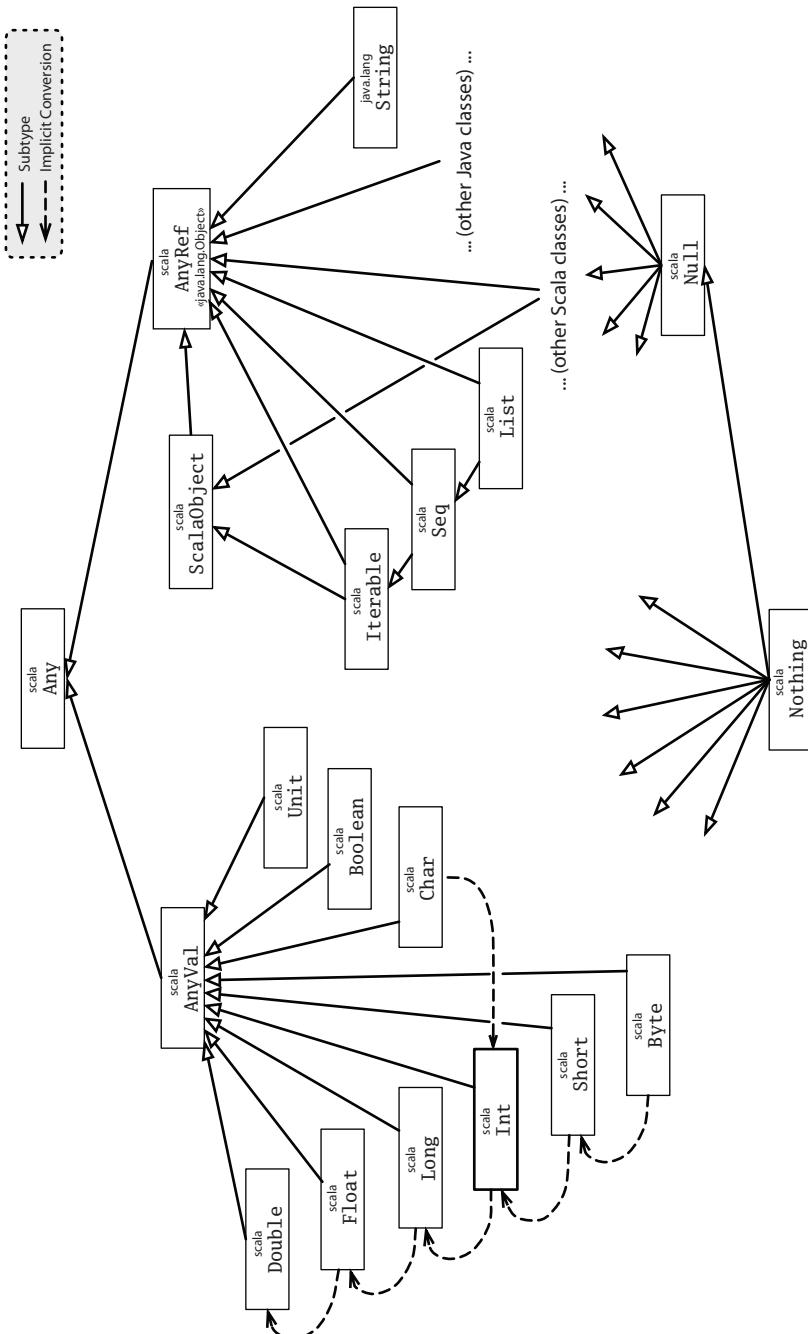


Figure 11.1: Class hierarchy of Scala.

plicit conversions between different value class types. For example, an instance of class `scala.Int` is automatically widened (by an implicit conversion) to an instance of class `scala.Long` when required.

As mentioned in [Section 5.9](#), implicit conversions are used to add more functionality to value types. For instance the type `Int` supports all of the operations below.

```
scala> 42 max 43
res7: Int = 43

scala> 42 min 43
res8: Int = 42

scala> 1 until 5
res9: Range = Range(1, 2, 3, 4)

scala> 1 to 5
res10: Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> 3.abs
res11: Int = 3

scala> (-3).abs
res12: 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` that is undefined in `Int` but defined in `RichInt`. Similar “booster classes” and implicit conversions exist for the other value classes. Implicit conversions will be discussed in detail in [Chapter 21](#).

The other subclass of the root class `Any` is class `AnyRef`. This is the base class of all *reference classes* in Scala. As mentioned previously, 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`.¹ One way to think of `java.lang.Object`, therefore, is as the way `AnyRef` is implemented on the Java Platform. Thus, although you can use `Object` and `AnyRef` interchangeably in Scala programs on the Java Platform, the recommended style is to use `AnyRef` everywhere.

¹The reason the `AnyRef` alias exists, instead of just using the name `java.lang.Object`, is because Scala was originally designed to work on both the Java and .NET platforms. On .NET, `AnyRef` is an alias for `System.Object`.

Scala classes are different from Java classes in that they also inherit from a special marker trait called `ScalaObject`. The idea is that the `ScalaObject` contains methods that the Scala compiler defines and implements in order to make execution of Scala programs more efficient. Right now, Scala object contains a single method, named `$tag`, which is used internally to speed up pattern matching.

11.2 How primitives are implemented

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:

```
// This is 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 that 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)
res13: Boolean = true

scala> def isEqual(x: Any, y: Any) = x == y
isEqual: (Any,Any)Boolean

scala> isEqual(42, 42)
res14: 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 = "abcd".substring(2)
x: java.lang.String = cd

scala> val y = "abcd".substring(2)
y: java.lang.String = cd

scala> x == y
res15: Boolean = true
```

In Java, the result of comparing `x` with `y` would be `false`. The programmer should have used `equals` in this case, but it is easy to forget.

However, there are situations where you need reference equality instead of user-defined equality. For example, in some situations where efficiency is paramount, you would like to *hash cons* with some classes and compare their

instances with reference equality.² For these cases, class AnyRef defines an additional eq method, which cannot be overridden and 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. For 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
res16: Boolean = true
scala> x eq y
res17: Boolean = false
scala> x ne y
res18: Boolean = true
```

Equality in Scala is discussed further in [Chapter 28](#).

11.3 Bottom types

At the bottom of the type hierarchy in [Figure 11.1](#) 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 that itself inherits from `AnyRef`). `Null` is not compatible with value types. You cannot, for example, assign a `null` value to an integer variable:

```
scala> val i: Int = null
<console>:4: error: type mismatch;
 found   : Null(null)
 required: Int
```

²You hash cons instances of a class by caching all instances you have created in a weak collection. Then, any time you want a new instance of the class, you first check the cache. If the cache already has an element equal to the one you are about to create, you can reuse the existing instance. As a result of this arrangement, any two instances that are equal with `equals()` are also equal with reference equality.

```
val i: Int = null
```

Type `Nothing` is at the very bottom of Scala's class hierarchy. Type `Nothing` 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 like this:

```
def error(message: String): Nothing =
    throw new RuntimeException(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, `x / y`, has type `Int`, whereas the `else` branch, the call to `error`, has type `Nothing`. Because `Nothing` is a subtype of `Int`, the type of the whole conditional is `Int`, as required.

11.4 Conclusion

In this chapter we showed you the classes at the top and bottom of Scala's class hierarchy. Now that you've gotten a good foundation on class inheritance in Scala, you're ready to understand mixin composition. In the next chapter, you'll learn about traits.

Chapter 12

Traits

Traits are a fundamental unit of code reuse in Scala. A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Unlike with inheritance, where a class must inherit from just one superclass, a class can mix in any number of traits.

This chapter shows you how traits work and shows you two of the most common ways they are useful: widening thin interfaces to rich ones, and defining stackable modifications. It also shows how to use the common `Ordered` trait, and it compares traits to the multiple inheritance of other languages.

12.1 How traits work

A trait definition looks just like a class definition except that it uses the keyword `trait`:

```
trait Printable {  
    def show() {  
        println(this)  
    }  
}
```

This trait is named `Printable`. It does not declare a superclass, so it has the default superclass of `AnyRef`. It defines one method, named `print`, and that method is concrete. It's a simple trait, just enough to show how traits work.

Once a trait is defined, it can be *mixed in* to a class using the `with` keyword. Scala programmers “mix in” traits rather than inherit from them, be-

cause mixing in a trait has important differences from the multiple inheritance found in many other languages. This issue is discussed in [Section 12.6](#). For example, here is a class that mixes in the `Printable` trait:

```
class Frog extends AnyRef with Printable {  
    override def toString = "a frog"  
}
```

Methods inherited from a trait can be used just like methods inherited from a superclass:

```
scala> val frog = new Frog  
frog: Frog = a frog  
  
scala> frog.show()  
a frog
```

A trait is also usable as a type. Here is an example using `Printable` as a type:

```
scala> var pr: Printable = frog  
pr: Printable = a frog  
  
scala> pr.show()  
a frog
```

The type of `pr` is `Printable`, a trait. Any object inheriting from `Printable` can be stored into `pr`, including `Frogs`.

As a syntactic shorthand, you can extend a trait directly if you do not care to extend a more specific class. The superclass will be taken to be the same as the traits superclass. In this case, the superclass of both `Frog` and `Printable` is `AnyRef`, so you could have defined `Frog` 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, i.e. parameters passed to the primary constructor of a class. In other words, although you could define a class like this:

```
scala> trait NoPoint(x: Int, y: Int) {}
<console>:1: error: traits or objects may not have parameters
          trait NoPoint(x: Int, y: Int) {}
                                         ^
```

The following attempt to define a trait would not compile:

```
trait NoPoint(x: Int, y: Int) { // Does not compile
}
```

One final thing. Traits can use `super`, but unlike in a normal class, `super` calls are virtual, dynamically bound. If you write “`super.toString`” in a class, then you know exactly which method will be called. When you write the same thing in a trait, the precise method to be called is only determined when you mix the trait into a concrete class. This curious behavior of `super` is key to allowing traits to be stackable modifications as described in [Section 12.5](#). The precise rules are given in [Section 12.6](#).

12.2 Thin versus rich 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 enrich a *thin* interface, making it into a *rich* interface.

Thin versus rich interfaces are a commonly faced trade off in object-oriented design. The trade off is between the implementers and the clients of an interface. A rich interface has many methods, which make it convenient for the caller. Clients can pick a method that exactly matches the functionality they need. A thin interface, on the other hand, has fewer methods, and thus is easier on the implementers. Clients calling into a thin interface, however, have to write more code. Given the smaller selection of methods to call, they will have to choose a less perfect matches and write extra code to adapt them to their needs.

Java’s interfaces are most often thin rather than rich. For example, the `CharSequence` interface of Java 1.4 is a thin interface common to all string-like classes that hold a sequence of characters. Here’s its definition when seen as a Scala trait:

```
trait CharSequence {
  def length: Int
```

```
def charAt(index: Int): Int
def substring(start: Int, end: Int): String
def toString(): String
}
```

Most of the dozens of methods in class `String` would apply to any `CharSequence`, but nonetheless `CharSequence` provides only four methods. If `CharSequence` instead included the full `String` interface, there would be a large burden on implementers of `CharSequence`. Every implementer would have to implement all of the dozens of methods in `String`.

Unlike Java interfaces, Scala traits can contain concrete methods. You just saw one example of this: the `show()` method of the `Printable` trait is concrete. For this reason, Scala traits make rich interfaces more convenient than they are in Java.

Adding a concrete method to a trait tilts the thin-rich trade off heavily towards rich 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, rich interfaces are less work to provide in Scala than in a language without traits.

To enrich an interface using traits, 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 enrichment trait, and end up with a class that has all of the thick interface available.

12.3 Example: Rectangular objects

Rectangular objects are a common example in graphics libraries where interface enrichment can really help. Such libraries frequently have many different classes that represent something rectangular. Windows, bitmap images, regions selected with the mouse, and rectangles themselves are all rectangular. To make these rectangular objects convenient to use, it is nice if the library writer provides geometric queries such as `width`, `height`, `left`, `right`, `topLeft`, and so on. However, there are many such methods it would be nice to have, so ordinarily only the most commonly used rectangular objects in a graphics library will support all of them. If such a library were

written in Scala, to contrast, the library writer could use traits to easily supply all of these convenience methods on all the classes they'd like.

To see how, first imagine what the code would look like without traits. There would be some basic geometry classes like `Point` and `Rectangle`:

```
class Point(val x: Int, val y: Int)
class Rectangle(val topLeft: Point, val bottomRight: Point) {
    def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
}
```

This `Rectangle` class takes two points in its primary constructor: the coordinates of the top-left corner and the coordinates of the bottom-right corner. It then implements many convenience methods such as `left`, `right`, and `width` by performing simple calculations on these two points.

Another class a graphics library might have is a 2-D graphical widget.

```
abstract class Component {
    def topLeft: Point
    def bottomRight: Point
    def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
}
```

Notice that the definitions of `left`, `right`, and `width` are exactly the same in the two classes. They will also be the same, aside from minor variations, in any other class of rectangular objects.

This repetition can be eliminated by using an enrichment trait. Call the trait `Rectangular`. `Rectangular` has two abstract methods for retrieving the top-left and bottom-right coordinates of the object. It can then supply concrete implementations of all the other geometric queries. The trait looks like this:

```
trait Rectangular {
    val topLeft: Point
```

```
    val bottomRight: Point  
    def left = topLeft x  
    def right = bottomRight x  
    def width = right - left  
    // and many more geometric methods...  
}
```

Class `Component` can then mix in this trait to get all the geometric query methods:

```
abstract class Component extends Rectangular {  
}
```

Similarly, `Rectangle` itself can be marked as rectangular:

```
class Rectangle(val topLeft: Point, val bottomRight: Point)  
extends Rectangular {  
}
```

Given these definitions, you can make a `Rectangle` and then call methods like `width` and `left` on it:

```
scala> val rect = new Rectangle(new Point(1,1), new Point(10,10))  
rect: Rectangle = Rectangle@ea288f  
  
scala> rect.left  
res2: Int = 1  
  
scala> rect.right  
res3: Int = 10  
  
scala> rect.width  
res4: Int = 9
```

12.4 The standard Ordered trait

Comparison is another domain where a rich interface is convenient. Whenever you compare two objects that are ordered, it is convenient if you use a single method call to ask about the precise comparison you want. If you want “is less than,” you would like to call `<`, and if you want “is less than or equal,” you would like to call `<=`. With a thin comparison interface, you

would have just one of these methods, and you would sometimes have to write things like “ $(x < y) \mid\mid (x == y)$.” A rich interface would provide you with all of the usual comparison operators, thus allowing you to directly write things like “ $x \leq y$.” In Scala, the standard `Ordered` trait allows you to implement one comparison method and then enrich the surrounding class to have four different variations.

Before looking at `Ordered`, imagine what you might do without it. Suppose you took the `Rational` class from [Chapter 6](#) and added comparison operations to it. You would end up with something like this:

```
final class Rational(n: Int, d: Int) {  
    // ...  
  
    def <(that: Rational) =  
        this.numer*that.denom > that.numer*this.denom  
  
    def >(that: Rational) = that < this  
    def <=(that: Rational) = (this < that) || (this == that)  
    def >=(that: Rational) = (this > that) || (this == that)  
}
```

This class defines four comparison operators (`<`, `>`, `<=`, and `>=`), and it’s a classic demonstration of the costs of defining a rich 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 rational numbers regarding `<=`. In a comparison context, `<=` is *always* used to mean “less than or equals.” Overall, there is quite a lot of boilerplate code in this class which would be the same in any other class that implements comparison operations.

This problem is so common that the Scala provides a trait to help with it. The trait is called `Ordered`. To use it, you replace all of the individual comparison methods with a single `compare` method. The `Ordered` trait then defines `<`, `>`, `<=`, and `>=` for you in terms of this one method.

Here is how it looks if you define comparison operations on `Rational` by using the `Ordered` trait:

```
final class Rational(n: Int, d: Int)  
extends Ordered[Rational]
```

```
{  
// ...  
  
def compare(that: Rational) =  
    this.numer*that.denom - that.numer*this.denom  
}
```

There are just two things to do. First, this version of Rational mixes in the Ordered trait. Unlike the traits you have seen so far, Ordered requires you to specify a *type parameter* when you mix it in. Type parameters are not discussed in detail until Chapter 19, but for now all you need to know is that when you mix in Ordered, you must actually mix in Ordered[*C*], where *C* is the class whose elements you compare. In this case, Rational mixes in Ordered[Rational].

The second thing you need to do is define a `compare` method for comparing two objects. This method should compare the receiver, `this`, with the object passed as an argument to the method. It should return an integer that is zero if the objects are the same, negative if receiver is less than the argument, and positive if the receiver is greater than the argument. In this case, the comparison method of Rational uses a formula based on converting the fractions to a common denominator and then subtracting the resulting numerators.

Given this mixin and the definition of `compare`, class Rational now has all four comparison methods.

```
scala> val half = new Rational(1,2)  
half: Rational = 1/2  
  
scala> val third = new Rational(1,3)  
third: Rational = 1/3  
  
scala> half < third  
res5: Boolean = false  
  
scala> half > third  
res6: Boolean = true
```

Any time you implement a class that is ordered by some comparison, you should consider mixing in the Ordered trait. If you do, you will provide the class's users with a rich set of comparison methods.

Beware that the `Ordered` trait does not define `equals` for you, because for technical reasons it is unable to do so. The problem is that `equals` is a universal method which takes an arbitrary object as a parameter, whereas the methods of `Ordered` all take arguments of a specific type parameter. That's why `equals` cannot be defined in terms of the other methods of class `Ordered`. So you still need to define `equals` explicitly. You'll find out how to go about this in [Chapter 28](#).

12.5 Traits as stackable modifications

You have now seen one major use of traits: turning a thin interface into a rich one. Now let us turn to a second major use: providing stackable modifications to classes. Traits let you *modify* the methods of a class, and they do so in a way that you can *stack* those modifications with each other.

Consider an example: modifications to a queue of integers. A queue of integers has `put` and `get` operations, where `put` places integers in the queue, and `get` takes them back out. Queues are first-in, first-out, so `get` should return the integers in the same order they were put in the queue.

Given any class that implements such a queue, here are some modifications you might want to make to it:

- *Doubling*: double all integers that are put in the queue
- *Incrementing*: increment all integers that are put in the queue
- *Filtering*: filter out negative integers from a queue

These three examples are all *modifications*. They modify the behavior of an underlying queue class rather than defining a full queue class themselves. These three are also *stackable*. You can select any of the three you like, mix them into a class, and obtain a new class that has all of the modifications you chose. Scala's traits are carefully designed to support this kind of stackable modification.

To see how it all works, start by imagining how queues themselves would look in code. An abstract `IntQueue` class would look as follows:

```
abstract class IntQueue {  
    def get(): Int  
    def put(x: Int)  
}
```

There's a `get()` method that returns integers, and there's a `put()` method for adding new integers to the queue. A simple implementation of a queue could use an `ArrayBuffer`:

```
import scala.collection.mutable.ArrayBuffer
class StandardQueue extends IntQueue {
    private val buf = new ArrayBuffer[Int]
    def get() = buf.remove(0)
    def put(x: Int) = buf += x
}
```

This implementation has a private field holding an array buffer. The `get` method removes an entry from one end of the buffer, while the `put` method adds elements to the other end. Here's how this implementation looks when you use it.

```
scala> val queue = new StandardQueue
queue: StandardQueue = StandardQueue@cc82658
scala> queue.put(10); queue.put(20)
scala> queue.get()
res7: Int = 10
scala> queue.get()
res8: Int = 20
```

So far so good. Now take a look at using traits to modify this behavior. The following trait modifies a queue so that elements are doubled when they are put in the queue.

```
trait Doubling extends IntQueue {
    abstract override def put(x: Int) = super.put(2 * x)
}
```

This short trait has two funny things going on. The first is that it declares a superclass, `IntQueue`. This declaration means that the trait can only be mixed into a class that also extends `IntQueue`. You can mix `Doubling` into `StandardQueue` but not into `Rational`.

The second funny thing is that the trait has a `super` call on a method declared abstract. Such calls are illegal for normal classes, because they will certainly fail at run time. For a trait, however, such a call can actually

succeed. Since super calls in a trait are dynamically bound, the super call in trait Doubling will work so long as the trait is mixed in *after* another trait or class which gives a concrete definition to the method.

This arrangement 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, and it means that the trait must be mixed into some class that has a concrete definition of the method in question.

There is a lot going on with such a simple trait, isn't there! Take a look now at how it looks to use the trait.

```
scala> class MyQueue extends StandardQueue with Doubling
defined class MyQueue

scala> val queue = new MyQueue
queue: MyQueue = MyQueue@b94819

scala> queue.put(10)

scala> queue.get()
res10: Int = 20
```

This interpreter session defines a new class `MyQueue` that extends `StandardQueue` and mixes in `Doubling`. It then puts a 10 on the queue, but because `Doubling` has been mixed in, the 10 is doubled to a 20. When the code then gets an integer from the queue, it sees a 20.

Note that `MyQueue` defines no new code. It simply identifies a class and mixes in a trait. In this common situation, you could supply “`StandardQueue with Doubling`” directly to `new` instead of defining a named class. It would look like this:

```
scala> val queue = new StandardQueue with Doubling
queue: StandardQueue with Doubling = $anon$1@7359cb

scala> queue.put(10)

scala> queue.get()
res12: Int = 20
```

That's all the code for the first modification, doubling all integers put on the queue. To see how to stack modifications, we need to define the other

ones. Here are implementations of the other two modifications, adding one to all integers, and filtering out negative integers.

```
trait Incrementing extends IntQueue {
    abstract override def put(x: Int) = super.put(x + 1)
}

trait Filtering extends IntQueue {
    abstract override def put(x: Int) =
        if (x >= 0) super.put(x)
}
```

Given these modifications, you can now pick and choose which ones you want for a particular queue. For example, here is a queue that both filters negative numbers and adds one to all numbers that it keeps.

```
scala> val queue = (new StandardQueue
|   with Incrementing
|   with Filtering)
queue: StandardQueue with Incrementing with Filtering =
$anon$1@934ea0

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res13: Int = 1

scala> queue.get()
res14: Int = 2
```

The order of mixins is significant. The precise rules are given in the following section, but, roughly speaking, traits further to the right take effect last. When you call a method on a class with mixins, the method in the trait furthest to the right is called first. If that method calls `super`, it invokes the method in the next trait to its left, and so on. In the previous example, `Filtering`'s `put` is invoked first, so it removes integers that were negative to begin with. `Incrementing`'s `put` is invoked second, so it adds one to those integers that remain.

If you reverse the order, then first integers are incremented, and *then* the integers that are still negative are discarded.

```
scala> val queue = (new StandardQueue
```

```
|   with Filtering
|   with Incrementing)
queue: StandardQueue with Filtering with Incrementing =
$anon$1@7be49d

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res15: Int = 0

scala> queue.get()
res16: Int = 1

scala> queue.get()
res17: Int = 2
```

Overall, code written in this style gives you a great deal of flexibility. You can define sixteen different classes by mixing in these three traits in different combinations and orders. That's a lot of flexibility for a small amount of code, so you should keep your eyes open for opportunities to arrange code as stackable modifications.

12.6 Why not multiple inheritance?

Traits are a way to inherit from multiple class-like constructs, but they differ in important ways from the multiple inheritance present in many languages. One difference is especially important: the interpretation of `super`.

With multiple inheritance, the method called by a `super` call can be determined right where the call appears. With traits, the method called is determined by 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 traditional multiple inheritance. Imagine the following code, but this time interpreted as multiple inheritance instead of trait mixin.

```
val queue = new StandardQueue with Incrementing with Doubling
queue.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 `Doubling` will get

called. Doubling doubles its argument and calls `super.put`, and that is it. No incrementing happened! Likewise, if the rule is that the first superclass wins, the resulting queue would increment integers but not double them. Thus neither ordering works.

Your next try might be to make an explicit subclass for incrementing, doubling queues, and have it call both `super` methods explicitly. Now you have new problems. For example, suppose you try the following:

```
trait MyQueue extends StandardQueue
with Incrementing with Doubling {
    def put(x: Int) = {
        Incrementing.super.put(x)
        Doubling.super.put(x)
    }
}
```

The verbosity of this attempt is the least of its problems. Now what happens is that the base class's `put` method gets called *twice*—once with an incremented value and once with a doubled value, but neither time with an incremented, doubled value.

There is simply no good solution to this problem using multiple inheritance. You have to back up in your design and factor the code differently. To contrast, the traits solution in Scala is straightforward. You simply mix in `Incrementing` and `Doubling`, and the Scala treatment of `super` makes it all work out. Something is clearly different here from traditional multiple inheritance, but what?

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, *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 call `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.

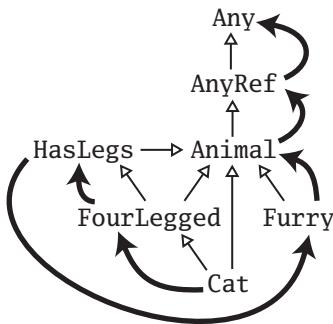


Figure 12.1: Linearization of class Cat

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 `Cat` which inherits from a superclass `Animal` and two traits `Furry` and `FourLegged`. `FourLegged` extends in turn another trait `HasLegs`.

```

class Animal
trait Furry extends Animal
trait FourLegged extends Animal with HasLegs
trait HasLegs extends Animal
class Cat extends Animal with Furry with FourLegged
  
```

The linearization of `Cat` is computed from back to front as follows. The *last* part of the linearization of `Cat` is the linearization of its superclass, `Animal`. This linearization is copied over without any changes. The second to last part is the linearization of the first mixin, trait `Furry`, but all classes that are already in the linearization of `Animal` are left out now, so that each class appears only once in `Cat`’s linearization. This is preceded by the linearization of `FourLegged`, 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 `Cat` is `Cat` itself.

[Figure 12.1](#) presents a diagram that depicts the linearization of class `Cat` graphically. If you apply this schema to the previous classes and traits, you should get the linearizations listed in [Table 12.1](#).

Class	Linearization
Animal	Animal, AnyRef, Any
Furry	Furry, Animal, AnyRef, Any
FourLegged	FourLegged, HasLegs, Animal, AnyRef, Any
HasLegs	HasLegs, Animal, AnyRef, Any
Cat	Cat, FourLegged, HasLegs, Furry, Animal, AnyRef, Any

Table 12.1: Linearization of several classes

12.7 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 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 tends to be awkward to inherit from a trait in a Java class. Inheriting from a Scala class, meanwhile, is exactly like inheriting from a Java class. As one exception, a Scala trait with only abstract members translates directly to a Java interface, so you should feel free to define such traits even if you expect Java code to inherit from it. See [Chapter 29](#) for more information on working with Java and Scala together.

If you plan to distribute it in compiled form, and you expect outside groups to write classes inheriting from it, then you might lean towards using an abstract class. The issue is that when a trait gains or loses a member, any classes that inherit from it must be recompiled, even if they have not changed. If outside clients will only call into the behavior, instead of inheriting from it, then using a trait is fine.

If efficiency is very important, lean towards using a 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.

12.8 Conclusion

This chapter has shown you how traits work and how to use them in several common idioms. As you have seen, traits are similar to multiple inheritance, but because they interpret `super` using linearization, they both avoid some of the difficulties of multiple inheritance, and they let you write traits that are stackable with each other. Along the way, you have seen how to use the `Ordered` trait and how to write your own enrichment traits, two of the most common uses of traits in Scala code.

Now that you have seen all of these facets, it is worth stepping back and taking another look at traits as a whole. Traits do not merely support the idioms described in this chapter. They are a fundamental unit of code that is reusable through inheritance. Because of this nature, many experienced Scala programmers start with traits when they are at the early stages of implementation. Each trait can hold less than an entire concept, a mere fragment of a concept. As the design solidifies, the fragments can be combined into more complete concepts by using trait mixin.

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 larger modules.

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's worth reading this chapter even if you already know Java.

Looking ahead, [Chapter 27](#) 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 previous example, class `Navigator` goes into the package named `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 that 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’s 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 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 underscore '_' instead of an asterisk '*' (after all, *, is a valid identifier in Scala!). The third import clause above corresponds to Java's import of static class fields.

These three imports give you a taste of what imports can do, but Scala imports are actually much more general. For one, 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 27](#).

Another way Scala imports are flexible is that they 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. This is useful to avoid ambiguities. Say you have two packages `Fruits` and `Notebooks` which both define a class `Apple`. If you want to get just the notebook named `Apple`, and not the fruit, you could still use two imports on demand like this:

```
import Notebooks._  
import Fruits.{Apple => _, _}
```

This would import all `Notebooks` and all `Fruits` except for the `Apple` fruit.

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 Implicit imports

Scala adds some imports implicitly to every program. In essence, it is as if the following three import clauses had been added to the top of every source file with extension “.scala”:

```
import java.lang._ // everything in the java.lang package
import scala._      // everything in the Scala package
import Predef._     // everything in the Predef object
```

The `java.lang` package contains standard Java classes. It is always implicitly imported on the JVM implementation of Scala. The .NET implementation would import package `System` instead, which is the .NET analogue of `java.lang`. Because `java.lang` is imported implicitly, you can write `Integer` instead of `java.lang.Integer`, for instance.

As you have no doubt realized by now, the `scala` package contains the standard Scala library, with many common classes and objects. Because `scala` is imported implicitly, you can write `List` instead of `scala.List`, for instance.

The `Predef` object contains many definitions of types, methods, and implicit conversions that are commonly used on Scala programs. Because `Predef` is imported implicitly, you can write `assert` instead of `Predef.assert`, for instance.

The three import clauses above are treated a bit specially in that later imports overshadow earlier ones. For instance, the `StringBuilder` class is defined both in package `scala` and, from Java version 1.5 on, also in package `java.lang`. Because the `scala` import overshadows the `java.lang` import, the simple name `StringBuilder` will refer to `scala.StringBuilder`, not `java.lang.StringBuilder`.

13.4 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 {
```

```
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

```
(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 from anywhere.

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 singleton 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 in package `bobsrockets`. In particular, the access to `Navigator` in object `Test` is permitted, because `Test` is contained in package `tests`, which is contained in `bobsrockets`. On the other hand, all 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 `Leg0fJourney` 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 would be legal from within `Navigator`:

```
current  
this.current
```

The following access, though, is not allowed, even if it appears inside class `Navigator`:

```
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 19.7](#) 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](#).

<code>private[_root_]</code>	same as public access
<code>private[bobsrockets]</code>	access within outer package
<code>private[navigation]</code>	same as package visibility in Java
<code>private[Navigator]</code>	same as <code>private</code> in Java
<code>private[LegOfJourney]</code>	same as <code>private</code> in Scala
<code>private[this]</code>	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 don't have any subclasses.

13.5 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 27](#) 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

Assertions and Unit Testing

Two important ways to check that the behavior of the software you write is as you expect are assertions and unit tests. In this chapter, we'll show you several options you have in Scala to write and run them.

14.1 Assertions

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 of `assert`. The expression `assert(condition, explanation)` tests `condition`, and, if it does not hold, throws an `AssertionError` that contains the given explanation. The type of explanation is `Any`, so you can pass any object as the explanation. The `assert` method will call `toString` on it to get a string explanation to place inside the `AssertionError`.

For example, in the method named “above” of class `Element`, shown in [Section 10.14](#), you might place an `assert` after the calls to `widen` to make sure that the widened elements have equal widths:

```
def above(that: Element): Element = {  
    val this1 = this widen that.width  
    val that1 = that widen this.width  
    assert(this1.width == that1.width)  
    elem(this1.contents ++ that1.contents)
```

¹The `assert` method is defined in the `Predef` singleton object, whose members are automatically imported into every Scala source file.

```
}
```

Another way you might choose do this is to check the widths at the end of the `widen` method, right before you return the value. You can accomplish this by storing the result in a `val`, performing an assertion on the result, then mentioning the `val` last so the result is returned if the assertion succeeds. You can do this more concisely, however, with a convenience method in `Predef` named `ensuring`:

```
private 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  
  } ensuring (w <= _.width)
```

The `ensuring` method can be used with any result type because of an implicit conversion. Although it looks in this code as if we're invoking `ensuring` on `widen`'s result, which is type `Element`, we're actually invoking `ensuring` on a type to which `Element` is implicitly converted. The `ensuring` method takes one argument, a predicate function that takes a result type and returns Boolean. `ensuring` will pass the result to the predicate. If the predicate returns true, `ensuring` will return the result. Otherwise, `ensuring` will throw an `AssertionError`.

In this example, the predicate is “`w <= _.width`”. The underscore is a placeholder for the one argument passed to the predicate, the `Element` result of the `widen` method. If the width passed as `w` to `widen` is less than or equal to the width of the result `Element`, the predicate will result in true, and `ensuring` will result in the `Element` on which it was invoked. Because this is the last expression of the `widen` method, `widen` itself will then result in the `Element`.

Assertions (and `ensuring` checks) can be enabled and disabled using the JVM's `-ea` and `-da` command-line flags. When enabled, each assertion serves as a little test that uses the actual data encountered as the software runs. In the remainder of this chapter, we'll focus on the writing of external unit tests, which provide their own test data and run independently from the application.

14.2 Unit testing in Scala

You have many options for unit testing in Scala, from established Java tools, such as JUnit and TestNG, to new tools written in Scala, such as ScalaTest, specs, and ScalaCheck. In the remainder of this chapter, we'll give you a quick tour of these tools. We'll start with ScalaTest.

ScalaTest provides several ways to write tests, the simplest of which is to create classes that extend `org.scalatest.Suite` and define test methods in those classes. A `Suite` represents a suite of tests. Test methods start with "test". Here's an example:

```
import org.scalatest.Suite
import Element.elem

class ElementSuite extends Suite {

    def testUniformElement() {
        val ele = elem('x', 2, 3)
        assert(ele.width == 2)
    }
}
```

This test attempts to create an element of width 2 and asserts that the width of the resulting element is indeed 2. Were this assertion to fail, you would see a message that indicated an assertion failed. You'd be given a line number, but wouldn't know the actual and expected values that were unequal. You could find out by placing a string message in the assertion that includes both values, but a more concise approach is to use the triple equals operator, which ScalaTest provides for this purpose:

```
assert(ele.width === 2)
```

Were this assertion to fail, you would see a message such as "3 did not equal 2" in the failure report. This would tell you that `ele.width` wrongly returned 3. The triple equals operator does not differentiate between the actual and expected result. It just indicates that the left operand did not equal the right operand. If you wish to emphasize this distinction, you could alternatively use ScalaTest's `expect` method, like this:

```
expect(2) {
    ele.width
}
```

With this expression you indicate that you expect the code between the curly braces to result in 2. Were the code between the braces to result in 3, you'd see the message, "Expected 2, but got 3" in the test failure report.

If you want to check that a method throws an expected exception, you can use ScalaTest's `intercept` method, like this:

```
intercept(classOf[IllegalArgumentException]) {  
    elem('x', -2, 3)  
}
```

If the code between the curly braces completes abruptly with an instance of the passed exception class, `intercept` will return the caught exception, in case you want to inspect it further. Most often, you'll probably only care that the expected exception was thrown, and ignore the result of `intercept`, as is done in this example. On the other hand, if the code does not throw an exception, or throws a different exception, `intercept` will throw an `AssertionError`, and you'll get a helpful error message in the failure report, such as:

```
Expected IllegalArgumentException to be thrown,  
but NegativeArraySizeException was thrown.
```

The goal of ScalaTest's triple equals operator and its `expect` and `intercept` methods is to help you write assertion-based tests that are clear and concise. In the next section, we'll show you how to use this syntax in JUnit and TestNG tests written in Scala.

14.3 Using JUnit and TestNG

The most popular unit testing framework on the Java Platform is JUnit, an open source tool written by Kent Beck and Erich Gamma. You can write JUnit tests in Scala quite easily. Here's an example using JUnit 3.8.1:

```
import junit.framework.TestCase  
import junit.framework.Assert.assertEquals  
import junit.framework.Assert.fail  
import Element.elem  
  
class ElementTestCase extends TestCase {  
    def testUniformElement() {
```

```
    val ele = elem('x', 2, 3)
    assertEquals(2, ele.width)
    assertEquals(3, ele.height)
    try {
        elem('x', -2, 3)
        fail()
    }
    catch {
        case e: IllegalArgumentException => // expected
    }
}
```

Once you compile this class, JUnit will run it like any other `TestCase`. JUnit doesn't care that it was written in Scala. If you wish to use ScalaTest's assertion syntax in your JUnit 3 test, however, you can do this instead:

```
import org.scalatest.junit.JUnit3Suite
import Element.elem

class ElementSuite extends JUnit3Suite {

    def testUniformElement() {
        val ele = elem('x', 2, 3)
        assert(ele.width === 2)
        expect(3) { ele.height }
        intercept(classOf[IllegalArgumentException]) {
            elem('x', -2, 3)
        }
    }
}
```

Trait `JUnit3Suite` extends `TestCase`, so once you compile this class, JUnit will run it just fine, even though it uses ScalaTest's more concise assertion syntax. Moreover, because `JUnit3Suite` mixes in ScalaTest's trait `Suite`, you can alternatively run this test class with ScalaTest's runner. The goal is to provide a gentle migration path to enable JUnit users to start writing JUnit tests in Scala that take advantage of the conciseness afforded by Scala. ScalaTest also has a `JUnit3WrapperSuite`, which enables you to run existing JUnit tests written in Java with ScalaTest's runner.

ScalaTest offers similar integration classes for JUnit 4 and TestNG, both of which make heavy use of annotations. We'll show an example using TestNG, an open source tool written by Cédric Beust and Alexandru Popescu. As with JUnit, you can simply write TestNG tests in Scala, compile them, and run them with TestNG's runner. Here's an example:

```
import org.testng.annotations.Test
import org.testng.Assert.assertEquals
import Element.elem

class ElementTests {

    @Test def verifyUniformElement() {
        val ele = elem('x', 2, 3)
        assertEquals(ele.width, 2)
        assertEquals(ele.height, 3)
    }

    @Test {
        val expectedExceptions =
            Array(classOf[IllegalArgumentException])
    }

    def elemShouldThrowIAE() {
        elem('x', -2, 3)
    }
}
```

If you prefer to use ScalaTest's assertion syntax in your TestNG tests, however, you can extend trait `TestNGSuite`, like this:

```
import org.scalatest.testng.TestNGSuite
import org.testng.annotations.Test
import Element.elem

class ElementSuite extends TestNGSuite {

    @Test def verifyUniformElement() {
        val ele = elem('x', 2, 3)
        assert(ele.width === 2)
        expect(3) { ele.height }
        intercept(classOf[IllegalArgumentException]) {
            elem('x', -2, 3)
        }
    }
}
```

```
    }  
}  
}
```

As with JUnit3Suite, you can run a TestNGSuite with either TestNG or ScalaTest, and ScalaTest also provides a TestNGWrapperSuite that enables you to run existing TestNG tests written in Java with ScalaTest. To see an example of JUnit 4 tests written in Scala, see [Section 29.2](#).

14.4 Using fixtures

To facilitate the creation of test fixtures, test data that is created anew for each test, JUnit 3 provides methods `setUp` and `tearDown`. JUnit 4 and TestNG provide annotations, such as TestNG’s `BeforeTest` and `AfterTest`, which mark methods responsible for setting up and tearing down fixtures. You can use these approaches when writing tests in Scala, however, they usually work by reassigning instance variables that refer to the fixture objects before each test is executed, which requires `vars`. If you prefer to write such tests without `vars`, you can use ScalaTest’s `FunSuite` family of traits.

The “Fun” in `FunSuite` stands for functional. It is so named because instead of implementing tests as methods, each test in a `FunSuite` is a function value. Here’s an example:

```
import org.scalatest.fun.FunSuite  
import Element.elem  
  
class ElementSuite extends FunSuite {  
  
  test("elem result should have passed width and height") {  
    val ele = elem('x', 2, 3)  
    assert(ele.width === 2)  
    assert(ele.height === 3)  
  }  
}
```

“`test`” is a method defined in `FunSuite`, which will be invoked by the primary constructor of `ElementSuite`. You specify the name of the test as a string between the parentheses, and the test code itself between curly braces. The test code is a function passed as a by-name parameter to `test`, which registers it for later execution. One benefit of `FunSuite` is you need not

name all your tests starting with “test”. In addition, you can more easily give long names to your tests, because you need not encode them in camel case, as you must do with test methods.

If you want to write tests that need the same mutable fixture objects, you can extend one of the traits `FunSuite1` through `FunSuite9`. If you need three fixture objects, for example, you would extend `FunSuite3`. Here’s an example that extends `FunSuite1`, to initialize a `StringBuilder` fixture object for each test:

```
import org.scalatest.fun.FunSuite1

class EasySuite extends FunSuite1[StringBuilder] {

    testWithFixture("easy test") {
        sb =>
            sb.append("easy!")
            assert(sb.toString === "Testing is easy!")
    }

    testWithFixture("fun test") {
        sb =>
            sb.append("fun!")
            assert(sb.toString === "Testing is fun!")
    }

    def withFixture(f: StringBuilder => Unit) {
        val sb = new StringBuilder("Testing is ")
        f(sb)
    }
}
```

In the class declaration of this example, we parameterized `FunSuite1` with the type of the lone fixture object, `StringBuilder`. We then defined two tests with `testWithFixture`. The function values provided here take the fixture object, a `StringBuilder`, as a parameter and use it in the test code. Note that the fixture object, referenced by `sb`, is mutated by both tests with the call to `append`. Lastly, we provided a `withFixture` method that takes a test function. This method creates a new `StringBuilder`, initializes it to “Testing is ”, and passes it to the test function. When ScalaTest runs

this suite, it will pass each test function to `withFixture`. The `withFixture` method will create and initialize a new `StringBuilder` object and pass that to the test function. In this way, each test function will get a fresh copy of the fixture.

14.5 Tests as specifications

We mentioned that one of the advantages of FunSuite was that it makes it easier to give tests longer names, because you need not write the test name in camel case. This supports a *behavior-driven development* testing style, in which each test is intended to represent a *specification* of one aspect of expected behavior of the code under test. The specs testing framework, an open source tool written in Scala by Eric Torreborre, supports this style of testing using an embedded DSL for writing specification-style tests. For example, you could use specs to write this:

```
"UniformElement's width" should {
    "equal the value passed to the primary constructor" in {
        val ele = elem('x', 2, 3)
        ele.width must be_== 2
    }
}
```

The goal of specs is to enable you to write tests that read more like natural language, and which generate descriptive failure messages. You can use specs standalone, but it is also integrated with ScalaTest and JUnit, so you can run specs tests with those tools as well.

14.6 Property-based testing

Perhaps the most powerful testing tool for Scala is ScalaCheck, an open source framework written by Rickard Nilsson, which is inspired by Haskell’s QuickCheck tool. ScalaCheck allows you to write properties that the code under test must obey. ScalaCheck will generate test data for you, and run tests that check that the property holds. Here’s an example of using ScalaCheck from a ScalaTest suite:

```
import org.scalatest.scalacheck.CheckSuite
```

```
import Element.elem
import org.scalatest.Arbitrary._
import org.scalatest.Prop._

class ElementSuite extends CheckSuite {
    def testUniformElement() {
        checkProperty(
            (w: Int) => {
                w > 0 ==> (elem('x', w, 3).width == w)
            }
        )
    }
}
```

The `checkProperty` method, defined in `CheckSuite`, makes it easy to write property-based tests inside ScalaTest, JUnit, and TestNG test suites. In this example, we check a property that the `elem` factory should obey. ScalaCheck properties are expressed as function values that take as parameters the required test data, which will be generated by ScalaCheck. In this case, the test data is an integer named `w` that represents a width. Inside the body of the function, you see:

```
w > 0 ==> (elem('x', w, 3).width == w)
```

The `==>` symbol is an *implication operator*. It implies that whenever the left hand expression is true, the property on the right must hold true. Thus in this case, the property on the right must hold true whenever `w` is greater than 0. The property in this case is a Boolean expression that will yield true if the width passed to the `elem` factory is the same as the width of the `Element` returned by the factory. With this small amount of code, ScalaCheck will generate possibly hundreds of values for `w` and test each one, looking for a value for which the property doesn't hold. If the property holds true for every value ScalaCheck tries, `checkProperty` returns normally. Otherwise, `checkProperty` will complete abruptly with an `AssertionError` that contains information including the value that caused the failure.

14.7 Conclusion

The testing frameworks shown in this chapter have many more features than we covered. To get the details, please consult their documentation.

Chapter 15

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. These 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 be new to you, though. Case classes are Scala's way to allow pattern matching on objects without requiring a large amount of boilerplate. In the common case, all you need to do is 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.

15.1 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, but it's a close call. 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)
```

The factory methods are particularly nice when you nest them. Because there are not noisy `new` keywords sprinkled throughout the code, you can take in the expression's structure at a glance.

```
scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))
```

The second syntactic convenience is that 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 just presented. 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 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 it's 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!¹ Almost as awkward, imagine implementing it as a long sequence of `if` statements, type tests, and type casts.

¹Gamma, et. al., *Design Patterns* [Gam94]

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`. In this case, no code is specified for that second case, so if that case runs it does nothing and returns `()`.

15.2 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 `Var(x)` matches any variable expression, binding `x` to the name of the variable. Used as an expression, `Var(x)`—exactly the same syntax—recreates an equivalent object, assuming `x` is already bound to the variable's name. Since the syntax of patterns is so transparent, 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 _ =>
}
```

Wildcards 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 `val` or singleton object can 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.

```
f() match {
    case 0 => "zero"
    case somethingElse => "not zero: " + somethingElse
}
```

Variable or constant?

Constant patterns can have symbolic names. You have seen this already when Nil is used as a pattern. Here is a related example, where a pattern match involves the constants E (2.71828) and Pi (3.14159).

```
scala> import Math.{E, Pi}
import Math.{E, Pi}

scala> E match {
    | case Pi => "strange math? Pi = "+Pi
    | case _ => "OK"
```

```
| }  
res10: java.lang.String = OK
```

As expected, E does not match Pi, so the “strange math” case is not used.

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 lowercase letter is taken to be a pattern variable; all other references are taken to be constants. To see the difference, create a lowercase 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
```

In this case the compiler will not even let you add a default case at all. Since `pi` is a variable pattern, it will match all inputs, and so no cases following it can be reached.

```
scala> E match {  
|   case pi => "strange math? Pi = "+pi  
|   case _ => "OK"  
| }  
<console>:9: error: unreachable code  
      case _ => "OK"  
                  ^
```

If you need to, you can still use a lowercase 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 even though they start with lowercase letters. 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.

```
scala> E match {
|   case `pi` => "strange math? Pi = "+pi
|   case _ => "OK"
| }
res13: java.lang.String = OK
```

As you can see, the back-tick syntax for identifiers is used for two different purposes in Scala to help you code your way out of unusual circumstances. Here you see that it can be used to treat a lowercased identifier as a constant in a pattern match. Earlier on, in [Section ??](#), you saw that they can also be used to treat a keyword as an ordinary identifier, *e.g.* writing `Thread.`yield`()` to treat `yield` as an identifier rather than a keyword.

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 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")
res14: Int = 3
scala> generalSize(Map(1 -> 'a', 2 -> 'b'))
res15: Int = 2
scala> generalSize(Math.Pi)
res16: 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 verbose 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-final (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
 ^

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))
res17: Boolean = true

scala> isIntIntMap(Map("abc" -> "abc"))
res18: 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 _ => "no"
    | }
isStringArray: (Any)java.lang.String

scala> val as = Array("abc")
as: Array[java.lang.String] = Array(abc)

scala> isStringArray(as)
res19: java.lang.String = yes

scala> val ai = Array(1, 2, 3)
ai: Array[Int] = Array(1, 2, 3)

scala> isStringArray(ai)
res20: 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.

15.3 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>:10: 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'

```

15.4 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 simplifyAll(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) =>
    simplifyAll(e) // '-' is its own inverse
  case BinOp("+", e, Number(0)) =>
    simplifyAll(e) // '0' is a neutral element for '+'
  case BinOp("*", e, Number(1)) =>
    simplifyAll(e) // '1' is a neutral element for '*'
  case UnOp(op, e) =>
    UnOp(op, simplifyAll(e))
  case BinOp(op, l, r) =>
    BinOp(op, simplifyAll(l), simplifyAll(r))
  case _ => expr
}

```

This version of `simplify` will apply simplification rules everywhere 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 `simplifyAll` recursively to the operand `e` and then rebuilds 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>:17: error: unreachable code
           case UnOp("-", UnOp("-", e)) => e
                           ^
```

15.5 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, nothing 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 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 25](#). 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. For example, in this case you add an `@unchecked` annotation to the variable `e`, with “`e: @unchecked`.” The `@unchecked` annotation has a special meaning for pattern matching. If a match’s selector expression carries this annotation, exhaustivity checking for the patterns that follow will be suppressed.

15.6 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"
res21: Option[java.lang.String] = Some(Paris)
scala> capitals get "North Pole"
res22: 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")
res23: String = Tokyo
scala> show(capitals get "North Pole")
res24: 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 for Java, 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. For Scala, the approach would not work at all, because it is possible to store value types in hash maps, and null is not a legal element for a value type. For instance, a `HashMap[Int, Int]` cannot return null to signify “no-element”.

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.

15.7 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 simple example:

```
val withDefault: Option[Int] => Int = {  
    case Some(x) => x  
    case None => 0  
}
```

This match expression has two cases. The first case matches a Some, and returns the number inside the Some. The second case matches a None, and returns a default value of zero. Here is this function in use:

```
scala> withDefault(Some(10))  
res25: Int = 10  
  
scala> withDefault(None)  
res26: Int = 0
```

This facility is quite useful for the actors library, described in [Chapter 30](#). Here is some typical actors code. It passes a pattern match 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 compile 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))
res27: Boolean = true

scala> second.isDefinedAt(List())
res28: Boolean = false
```

The typical example of a partial function is a pattern matching function literal like the one in the previous example. In fact, such an expression gets translated by the Scala compiler to a partial function by translating the patterns twice – once for the implementation of the real function, and once to test whether the function is defined or not. For instance, the function literal `{ case x :: y :: _ => y }` above gets translated to the following partial function value:

```
new PartialFunction[List[Int], Int] {
    def apply(xs: List[Int]) = xs match {
        case x :: y :: _ => y
    }
    def isDefinedAt(xs: List[Int]) = xs match {
        case x :: y :: _ => true
        case _ => false
    }
}
```

This translation takes effect whenever the declared type of a function literal is `PartialFunction`. If the declared type is just `Function1`, or is missing, the function literal is instead translated to a complete function.

In general, you should try to work with complete functions whenever possible, because using partial functions allows for runtime 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 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.

15.8 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:

$$\begin{array}{cc} a & 1 \\ \hline b * c & n \end{array} + -$$

3

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](#):

```
import layout.Element
import Element.elem
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 where to put parentheses, the code 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 ".0") 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 that 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
```

15.9 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 16

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.

16.1 List literals

You have seen lists already in the preceding chapters, so you know that a list containing the elements 'a', 'b', and '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.

16.2 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 11.3](#) 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()

```

16.3 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))) :: 
              (0 :: (1 :: (0 :: Nil))) :: 
              (0 :: (0 :: (1 :: Nil))) :: Nil
val empty  = Nil

```

¹[Chapter 19](#) gives more details on covariance and other kinds of variance.

Table 16.1: Basic list operations.

What it is	What it does
<code>empty.isEmpty</code>	returns <code>true</code>
<code>fruit.isEmpty</code>	returns <code>false</code>
<code>fruit.head</code>	returns "apples"
<code>fruit.tail.head</code>	returns "oranges"
<code>diag3.head</code>	returns <code>List(1, 0, 0)</code>

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`.

16.4 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 16.1](#).

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)
```

16.5 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 15](#), 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 22](#).

This is all you need to know about lists in Scala to be able to use them correctly. However, there are 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.

16.6 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 our concatenation method (let’s call it `append`). In order not to mix things up 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 parameter² which

²Type parameters are explained in more detail in Chapter 19

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
```

Unlike with arrays, `length` is a relatively expensive operation on lists. It needs to traverse the whole list to find its end and therefore takes time proportional to the number of elements in 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 both run in constant time, `init` and `last` need to traverse the whole 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 to 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, except that the elements are reversed:

```
xs.reverse.init == xs.tail.reverse
xs.reverse.tail == xs.init.reverse
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. List concatenation `xs :: ys` takes time proportional to the length of its first argument xs. Hence, the total complexity of rev 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 implementation possible. You will see in [Section 16.7](#) how to speed it up.

Prefixes and suffixes: drop, take and splitAt

The drop and take operations generalize tail and init in that they return arbitrary prefixes or suffixes of a list. The expression `xs take n` returns the first n elements of the list xs. If n is greater than `xs.length`, the whole list xs is returned. The operation `xs drop n` returns all elements of the list xs except the first n ones. If n is greater than `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 most efficiently with 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
```

The `mkString` method has two overloaded variants which let you drop some or all of its arguments. The first variant only takes a separator string:

```
xs mkString sep = xs mkString ("", sep, "")
```

The second variant lets you omit all arguments:

```
xs.mkString = xs mkString ""
```

Here are some examples:

```
scala> abcde mkString ("[", ",", "]")
res17: String = [a,b,c,d,e]

scala> abcde mkString ""
res18: String = abcde

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 order ($n \log(n)$), 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)
```

Partially applied functions

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 missing argument is the list that should be sorted.

16.7 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))
```

`List.range` is a utility method that creates a list of all integers in some range. It is used twice in this example: once to generate a list of integers from 1 (including) until 5 (excluding), and in a second time to generate a list of integers from 1 until i , for each value of i taken from the first list. The `map` in this expression generates a list of tuples (i, j) where $j < i$. The outer `flatMap` in this example generates this list for each i between 1 and 5, and then concatenates all the results.

Note that the same List can alternatively be constructed with a for-expression:

```
for (i <- List.range(1, 5); j <- List.range(1, i)) yield (i, j)
```

You'll learn more about the interplay of for-expressions and list operations in [Chapter 23](#).

The third map-like operation is `foreach`. Unlike `map` and `flatMap`, `foreach` takes a procedure (a function 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 it returns a pair of lists. One list contains all elements for which the predicate is true, while 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, $\text{Some}(x)$ is returned. Otherwise, if p is false for all elements, None is returned. Here are some 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 . Here are some 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

$$xs \text{ span } p = (xs \text{ takeWhile } p, xs \text{ dropWhile } p)$$

Like `splitAt`, `span` avoids traversing 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:

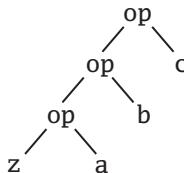
```
scala> def sum(xs: List[Int]): Int = (0 /: xs) (_ + _)
sum: (List[Int])Int

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
```

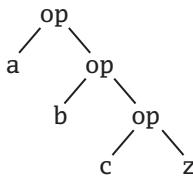
This gives an extra space at the beginning. To remove the space, you can use this slight variation:

```
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 the next section of 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 22](#).

Example: List reversal using fold

Earlier in the chapter you saw an implementation of method `reverse` whose running 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> words sort (_.length > _.length)
res2: List[java.lang.String] = List(quick, brown, fox, the)

```

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`.

16.8 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 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, which you saw briefly earlier in the chapter, creates a list consisting of a range of numbers. Its simplest form is `List.range(from, to)`, which 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 are `step` values apart, starting at `from`. The `step` can be positive or 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, whereas Scala's type system requires every method of a class to be available on every instance of that class. Thus, `unzip` cannot go in the `List` class. It might be possible to extend Scala's type system in the future so that it accepts methods that only apply to some instances of a class, 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)
```

The `flatten` method 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
```

16.9 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) => 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 short-hand 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:

```
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 shown previously?

```
(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 of type inference used in functional languages such as ML or Haskell. However, Scala's local type inference deals much more gracefully with object-oriented subtyping than the Hindley/Milner style 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.

16.10 Conclusion

Now you have seen many ways to work with lists. You have seen the basic operations like `head` and `tail`, the first-order operations like `reverse`, the higher-order operations like `map`, and the utility methods in the `List` object. Along the way, you have learned how Scala's type inference works.

Lists are a real work horse in Scala, so you will benefit from knowing how to use them. For that reason, this chapter has delved deep into how to use lists. Lists are just one kind of collection that Scala supports, however. The next chapter is broad, rather than deep, and shows you how to use a variety of Scala's collection types.

Chapter 17

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 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.

17.1 Overview of the library

[Figure 17.1](#) 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 is 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 it includes any negative numbers. The library would search forever.

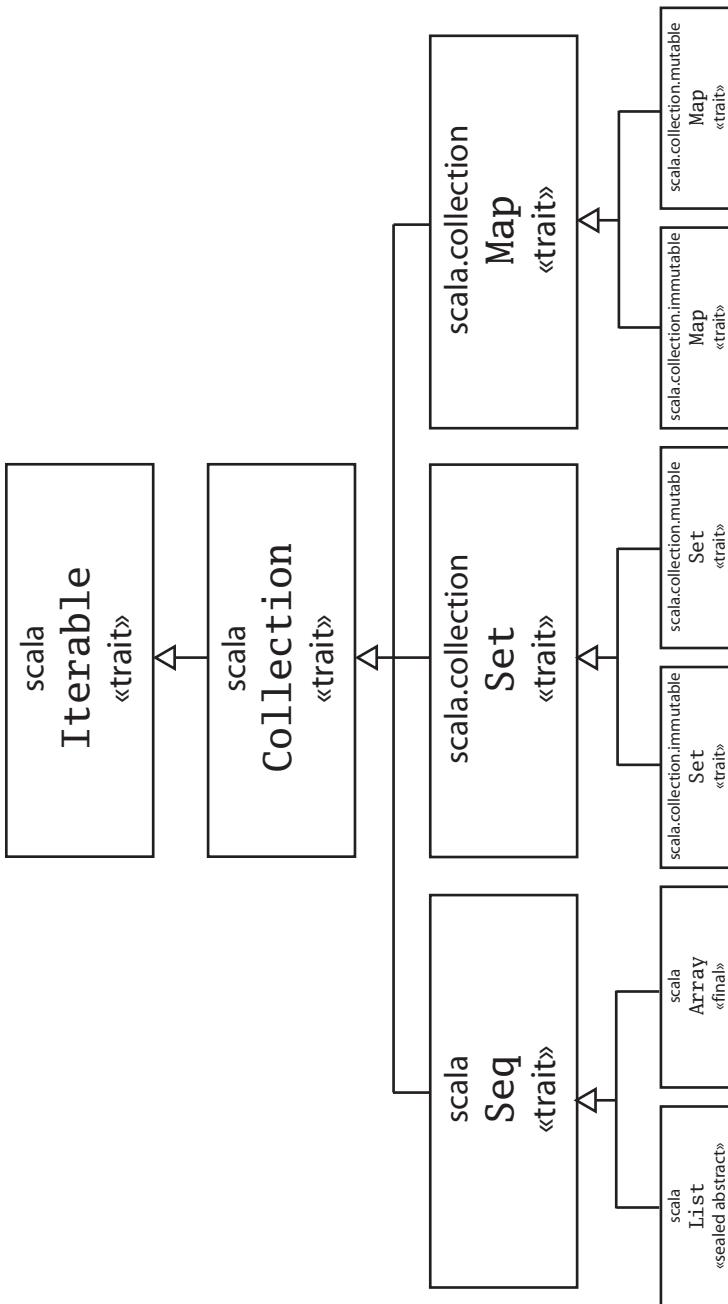


Figure 17.1: Scala collections class hierarchy.

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 17.1](#), 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.

17.2 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`.

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 fact, arrays are treated as objects with `apply` methods. So the application above is in fact equivalent to:

```
scala> numbers.apply(3)
res1: 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)
|   print(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)
|   print(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 15](#). 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)
```

17.3 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
```

¹This syntax is actually a special case of *pattern matching*, as described in detail in Chapter 15.

```
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 18](#) 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 catch mistakes.

17.4 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 several times 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
```

Table 17.1: 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)

```
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 17.1. 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 17.2.

Creating a mutable map looks like this:

```
scala> val map = mutable.Map.empty[String, Int]
map: scala.collection.mutable.Map[String,Int] = Map()
```

Table 17.2: 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 17.2](#).

17.5 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)
```

17.6 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 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.

Table 17.3: 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

Furthermore, immutable collections can usually be stored more compactly than mutable ones if the number of elements stored in the collection is small. For instance an empty mutable map in its default representation `HashMap` takes up about 80 bytes and about 16 more are added for each entry that's added to it. An empty immutable Map is a single object that's shared between all references, so referring to it essentially costs just a single pointer field. What's more, the current Scala compiler stores immutable maps and sets with up to four entries in a single object, which typically takes up between 16 and 40 bytes, depending on the number of entries stored in the collection. So for small maps and sets, the immutable versions are much more compact than the mutable ones. Given that many collections are small, switching them to be immutable can give important space savings and performance advantages.

The main immutable types are given in in [Table 17.3](#). 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 way this is implemented is through aliases in the `Predef` object, which is implicitly imported into every Scala program. Here are the relevant definitions:

```
object Predef {
    type Set[T] = scala.collection.immutable.Set[T]
    type Map[K,V] = scala.collection.immutable.Map[K,V]
    val Set = scala.collection.immutable.Set
    val Map = scala.collection.immutable.Map
    ...
}
```

So `Map` is the same as `Predef.Map`, which is defined to be the same as

`scala.collection.immutable.Map`. This holds both for `Map` type and the `Map` object.

The third column of [Table 17.3](#) 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`. To convert a mutable map `x` to an immutable one, you can write `Map.empty ++ x` which joins all elements of `x` to the immutable empty map. Likewise, to convert a mutable set `x` to an immutable one, you can write `Set.empty ++ x`.

Conversion in the opposite direction is also possible: `x.toArray` converts a list `x` to an array. To convert an immutable set or map to a mutable one, 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
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 17.4](#) and [Table 17.5](#). 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.

Table 17.4: 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 3)
<code>nums.contains(3)</code>	check for inclusion (returns true)

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 `+=`.

```
scala> val authorized = Set("Nancy", "Jane")
authorized: scala.collection.immutable.Set[java.lang.String]
= Set(Nancy, Jane)

scala> authorized += "Bill"
<console>:6: error: value += is not a member of
scala.collection.immutable.Set[java.lang.String]
        authorized += "Bill"
```

Table 17.5: 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)

^

If you declare `authorized` as a `var` instead of a `val`, however, then the collection can be “updated” with `+=` even though it is immutable. A new collection will be created and then `authorized` will be reassigned to refer to the new collection.

```
scala> var authorized = Set("Nancy", "Jane")
authorized: scala.collection.immutable.Set[java.lang.String]
= Set(Nancy, Jane)
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
```

The effect of this expansion is similar to Java's assignment operators `+=`, `-=`, etc., but it is more general because every operator ending in `=` can be converted. There's one difference with Java, though, which is a potential trap: the precedence of an assignment operator in Scala is determined by its first character, as is the case for all operators. There's no special treatment of assignment operators when it comes to precedence. So `+=` has the same precedence as `+` in Scala. This means that for a variable `x` the expression `x += y + z` parses as `(x += y) + z`, and therefore expands to `(x = x + y) + z`. This is probably not what you intended (in fact, it gives a compile time error). The correct way to write the expression is with parentheses around the right operand: `x += (y + z)`.

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. Immutable collections 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.

17.7 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 cleaner and easier to work with.

Chapter 18

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.

18.1 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) {  
        require(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 withdrawal 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 would be a class that caches the result of an expensive operation in a field for optimization purposes. To pick an example, assume the following unoptimized class `Keyed` with an expensive operation `computeKey`:

```
class Keyed {  
    def computeKey: Int = ... // this will take some time  
    ...  
}
```

Provided that `computeKey` neither reads nor writes any `vars`, one can make `Keyed` more efficient by adding a cache:

```
class MemoKeyed extends Keyed {  
    private var keyCache: Option[Int] = None  
    override def computeKey: Int = {  
        if (!keyCache.isDefined) keyCache = Some(super.computeKey)  
        keyCache.get  
    }  
}
```

Using `MemoKeyed` instead of `Keyed` can speed up things, because the second time the result of the `computeKey` operation is requested, the value stored in the `keyCache` field can be returned instead of running `computeKey` once again. But except for this speed gain, the behavior of class `Keyed` and `MemoKeyed` is exactly the same. Consequently, if `Keyed` is purely functional, then so is `MemoKeyed`, even though it contains a reassignable variable.

18.2 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 requirements 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) { require(0 <= x && x < 24); h = x }  
    def minute = m  
    def minute_=(x: Int) { require(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 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 (you'll see examples of this in [Chapter 33](#)).

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 zero value to that field. As was explained in [Chapter 17](#), the zero value depends on the field's type. It is 0 for numeric types, `false` for booleans, and `null` for reference types. This 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.¹

The `celsius` variable definition is followed by a getter `fahrenheit` and a setter `fahrenheit_=` that access the same temperature, but in degrees 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
```

¹Abstract variables are explained in [Chapter 20](#)

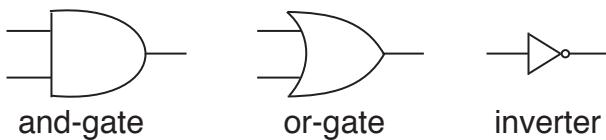


Figure 18.1: Basic gates.

18.3 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 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.

18.4 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 18.1 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:

```
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

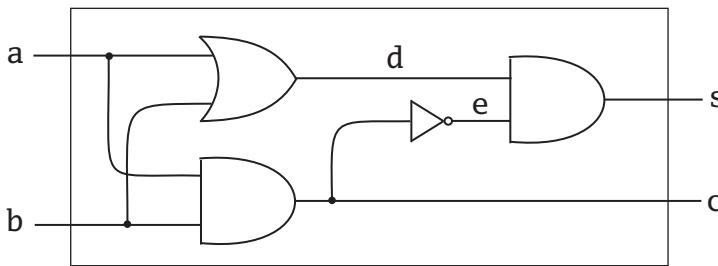


Figure 18.2: A half adder circuit.

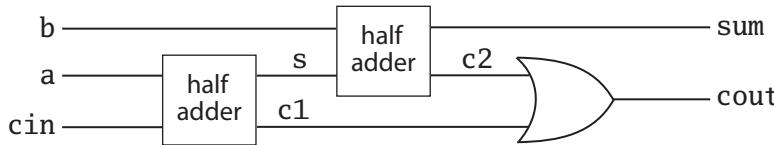


Figure 18.3: A full adder circuit.

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)
}
  
```

A picture of this half-adder is shown in Figure 18.2.

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 18.3, which takes two inputs a and b as well as a carry-in `cin` and which produces a sum output defined by $\text{sum} = (a + b + \text{cin}) \% 2$ and a carry-out output defined by $\text{cout} = (a + b + \text{cin}) / 2$.

```
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.

18.5 The Simulation API

The simulation API is shown in Figure 18.4. 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
```

```
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 18.4: The Simulation class.

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. After all, you don't usually want your simulation objects to manipulate the current time, except possibly in the case where your simulation models time-travel.

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 20](#) 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 “ $\Rightarrow \text{Unit}$,” *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. Note that `afterDelay` is a curried function. It's a good example of the principle set forward in [Chapter ??](#) that currying can be used to make method calls look more like built-in syntax.

The created work item still needs to be 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 first item in the agenda, removes it from the agenda and executes it. It does this 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.

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 15](#) 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()
  }
}
```

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.

18.6 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 once at the time the action is installed, and thereafter 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`.

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 in 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
```

Because we are going to access the members of the `MySimulation` object repeatedly, an import of the object keeps the subsequent code shorter.

```
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
```

18.7 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 19

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.

19.1 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. The goal is a class that works like this:

```
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
res0: 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 Queue2[T](smele: List[T]) {
    // smele is elems reversed
}

```

```
def head = smele.last
def tail = smele.init
def append(x: T) = new Queue2(x :: smele)
}
```

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 19.1 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.

```
class Queue[T](  
    private val leading: List[T],  
    private val 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(x: T) =  
        new Queue(leading, x :: trailing)  
}
```

Figure 19.1: Simple functional queues

Now, there are 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.

19.2 Information hiding

Private constructors

The implementation of `Queue` is now quite good with regards to efficiency. One might object, though, that this efficiency is paid for by exposing a need-

lessly 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 (
    private val leading: List[T],
    private val 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>:6: error: constructor Queue cannot be accessed in
object $iw
        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:

```
def this(elems: T*) = this(elems.toList, Nil)
```

Recall that T^* is the notation for repeated parameters, as described in [Section 8.8](#).

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 '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. Because of this, the apply method in object Queue can 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)`. This expression expands to `Queue.apply(1, 2, 3)` since Queue is an object instead of a function. As a result, 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: private classes

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 19.2](#) 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. Instead of hiding individual constructors and methods, this version hides the whole implementation class.

```
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[T](xs.toList, Nil)  
  
    private class QueueImpl[T](  
        private val leading: List[T],  
        private val 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(q.leading.tail, q.trailing)  
        }  
  
        def append(x: T) =  
            new QueueImpl(leading, x :: trailing)  
    }  
}
```

Figure 19.2: Type abstraction for functional queues

19.3 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 type `S` then `Queue[T]` should be a subtype of `Queue[S]`. This property is called *covariant* subtyping.

In Scala, generic types have by default *nonvariant* subtyping. That is, with `Queue` defined as above, queues with different element types would never be in a subtype relation. A `Queue[String]` would not be usable as a `Queue[AnyRef]`. However, you can demand covariant 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 covariant in that parameter. By adding this single character, you are telling Scala that you want `Queue[String]`, for example to be a subtype of `Queue[AnyRef]`. As described later, the compiler will check that `Queue` is defined in a way that this subtyping is sound.

Besides `+`, there is also a prefix `-` which indicates *contravariant* 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!). Whether a type parameter is covariant, contravariant, or nonvariant is called collectively the parameter's *variance*.

In a purely functional world, many types are naturally covariant. 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 nonvariant. 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 it 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 `c2`'s 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 typesoundness.

Which operation is to blame for the runtime fault? It must be the second one which uses covariance. The other statements are too simple and fundamental. Thus, 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`. You cannot use `set` with an `Int` argument on a `Cell` of `String`, for example.

In fact, if you pass the covariant version of `Cell` to the Scala compiler, you would get a compile-time error:

```
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 compiles, but executing the program will cause an `ArrayStore` exception to be thrown when `a2[0]` is stored into.

```
Exception in thread "main" java.lang.ArrayStoreException:
java.lang.Integer
at JavaArrays.main(JavaArrays.java:8)
```

What happens here is that 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. For compatibility reasons, though, it has persisted in Java 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] = Array(abc)

scala> val a2: Array[Any] = a1
<console>:5: error: type mismatch;
 found   : Array[java.lang.String]
```

```
required: Array[Any]
val a2: Array[Any] = a1
^
```

What happened is that Scala treats arrays as nonvariant, so an `Array[String]` is not considered to conform to an `Array[Any]`. 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[java.lang.Object] = Array(abc)
```

The cast is always legal at compile-time, and it will always succeed at run-time, because the JVM's underlying run-time model treats arrays a covariant, just as Java the language does. But you might get `ArrayStore` exceptions afterwards, again just as you would in Java.

19.4 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.

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 though a Scala compiler would yield:

```
Queues.scala:11: 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. The compiler then checks each use of a type parameter. Covariant type parameters may only be used in covariant positions, while contravariant type parameters may

only be used in contravariant positions. A nonvariant parameter may be used in any position, and is the only kind of parameter that can be used in a non-variant position.

To classify the positions, the compiler starts from the declaration of a type parameter and then moves inward through deeper nesting levels. Positions at the top level of the declaring class are classified as covariant. By default, positions at deeper nesting levels are classified the same as that at enclosing levels. There are a handful of exceptions where the classification changes. Positions at a method parameter are classified as the *flip* of positions outside the method, where a flip changes the classification from covariant to contravariant or *vice versa*. Likewise, positions at type parameters of a method are flipped.

A variance change is sometimes performed for the type argument position of a type such as $C[T]$, depending on the variance of the corresponding type parameter. If C 's type parameter is covariant, then the classification stays the same. If C 's parameter is contravariant, then a flip occurs. If C 's parameter is nonvariant, then the current classification is changed to nonvariant.

As a somewhat contrived example, consider the following class definition, where the variance of several positions is annotated with $^+$ (for covariant) or $-$ (for contravariant).

```
abstract class C[-T, +U] {  
    def f[W-](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. 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.

You see from this discussion that it's quite hard to keep track of variance positions. That's why it's a welcome relief that the Scala compiler does this job for you.

Once the variances are computed, the compiler checks that each type parameter is only used in positions that are classified appropriately. In this

case, T is only used in contravariant positions, and U is only used in covariant positions. So class C is type correct.

19.5 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.

Fortunately, there's a way to get unstuck: You can generalize the append method by making it polymorphic and using a lower bound for its type parameter. Here's a new formulation of Queue that implements this idea.

```
class Queue[+T] {  
    def append[U >: T](x: U) =  
        new Queue[U](leading, x :: trailing)  
    ...  
}
```

The new definition gives append a type parameter U, and requires that U is a supertype of T. So the element type T of the queue becomes a lower bound for U. The parameter to append is now of type U instead of type T, and the return value of the method is now Queue[U] instead of Queue[T].

As an example, suppose there is a class Fruit with two subclasses, Apple and Orange. With the new definition of class Queue, it is possible to append an Orange to a Queue[Apple]. The result will be a Queue[Fruit].

This revised definition of append is type correct. Intuitively, if T is a more specific type than expected (*e.g.*, Apple instead of Fruit), a call to append will still work, because U (Fruit) will still be a supertype of T (Apple). Technically, what happens is that a flip operation occurs for lower bounds. The type parameter U is in a contravariant position (1 flip), while the lower bound (>: T) is in a covariant position (2 flips).

The new definition of append is arguably better than the old one, because it is more general. Unlike the old version, the new definition allows appending 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.

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, and if they get it wrong, some important instance methods will no longer be visible. Variance being a tricky business, users usually get it wrong, and they come away thinking that wildcards and generics are overly complicated. With definition-side variance, you express your intent to the compiler, and the compiler will double check that the methods you want available will indeed be available.

19.6 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, whenever you write the function type `A => B`, Scala expands this to `Function1[A, B]`. The definition of `Function1` in the standard library uses both covariance and contravariance:

```
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.

19.7 Object private 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 times in a row on a list where `leading` is empty. The wasteful copying could be avoided by adding some judicious side effects. [Figure 19.3](#) 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

```
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[U](leading, x :: trailing)
}
```

Figure 19.3: Optimized functional queues

to the implementation of the Queue operation; since leading and trailing are private variables, the effect is not visible to clients of Queue. So by the terminology established in Chapter 18, the new version of Queue still defines purely functional objects, in spite of the fact that they now contain assignable fields.

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 private. Recall from Chapter 13 that object

private members can be accessed only from within the object in which they are defined. It turns out that accesses to variables from the same 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 containing object that has a statically weaker type than the type the object was defined with. For accesses to object private values this is impossible.

Scala's variance checking rules contain a special case for object private 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 19.3](#) 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.

```
Queues.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],
^
Queues.scala:1: error: covariant type T occurs in
contravariant position in type List[T] of parameter of
setter trailing_=
                  private var trailing: List[T])) {
```

19.8 Conclusion

In this chapter you have seen several techniques for information hiding: private constructors, factory methods, type abstraction, and object private 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 flexible variance annotations: lower bounds for method type parameters, and `private[this]` annotations for local fields and methods.

Chapter 20

Abstract Members

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 declared. 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 declare abstract fields and even abstract types.

An example is the following trait `Abstract` which declares 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.

20.1 Abstract vals

An abstract val declaration 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 val declaration 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 val declaration resembles an abstract parameterless method declaration 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 every time 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 returns a different value every time it's 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 a def. Abstract method declarations, on the other hand, may be implemented by both concrete method definitions and concrete val definitions. Given the

abstract class A below, class C1 would be a legal implementation but class C2 would not.

```
abstract class A {  
    val v: String    // 'v' for value  
    def m: String   // 'm' for method  
}  
  
class C1 extends A {  
    val v: String  
    val m: String // OK to override a 'def' with a 'val'  
}  
  
class C2 extends A {  
    def v: String // ERROR: cannot override a 'val' with a 'def'  
    def m: String  
}
```

20.2 Abstract vars

Like an abstract val, an abstract var declares just a name and a type, but not an initial value. For instance, here is a class `AbstractTime` which declares 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 18](#) 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 declare an abstract var `x`, you implicitly declare 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'  
}
```

20.3 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, an abstract type declaration 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 declared. 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 of modeling the 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:

```
BuggyAnimals.scala:7: error: class Cow needs to be
abstract, since method eat in class Animal of type
(Food)Unit is not defined
class Cow extends Animal {
^

BuggyAnimals.scala:8: 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 modeling. 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 modeled 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  
    override 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@674bf6  
  
scala> cow eat (new Fish)  
<console>:10: 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. The meaning of cow.SuitableFood is, “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 such as cow 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 SuitableFoods 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.

20.4 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. Nonetheless, 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 differences from the previous situation are 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 the 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 would you 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 can neither create an instance of an abstract type, nor 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, declare an abstract method that 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:

```
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 dubious, for at least two reasons. First, if you have some amount of currency

¹ There’s some promising recent research on *virtual classes*, which would allow this, but virtual classes are not currently supported in Scala.

(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 that 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 {  
    type Currency = Dollar  
    def make(x: Long) = new Dollar { val amount = x }  
    abstract class Dollar extends AbstractCurrency {  
        def designation = "USD"  
    }  
}
```

```
}
```

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 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 *value* `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 `Currency`'s `toString` method as follows:

```
override def toString =
  ((amount.toDouble / CurrencyUnit.amount.toDouble)
    formatted ("%." + decimals(CurrencyUnit.amount) + "f")
    + " " + designation)
```

Here, `formatted` is a method which Scala makes available on several classes including `Double`. (Scala uses rich wrappers, described in [Section 5.9](#), to make `formatted` available.) This method returns a string which is the receiver `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 `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(n: Long): Int =
  if (n == 1) 0 else 1 + decimals(n / 10)
```

Here are some other currency zones:

```
object Europe 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 that 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, `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.

The final version of the `CurrencyZone` class is shown in [Figure 20.1](#). 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
res1: Japan.Currency = 12110 JPY

scala> Europe.Euro from res1
res2: Europe.Currency = 75.95 EUR

scala> US.Dollar from res2
res3: US.Currency = 99.95 USD
```

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 + res3
res4: currencies.US.Currency = 199.95
```

On the other hand, you cannot add amounts of different currencies:

```
scala> US.Dollar + Europe.Euro
<console>:7: error: type mismatch;
 found   : currencies.Europe.Euro
 required: currencies.US.Currency
          US.Dollar + Europe.Euro
                           ^
```

Figure 20.1: The full code of class CurrencyZone

```
abstract class CurrencyZone {  
    type Currency <: AbstractCurrency  
    def make(x: Long): Currency  
    abstract class AbstractCurrency {  
        val amount: Long  
        def designation: String  
        def +(that: Currency): Currency =  
            make(this.amount + that.amount)  
        def *(x: Double): Currency =  
            make((this.amount * x).toLong)  
        def -(that: Currency): Currency =  
            make(this.amount - that.amount)  
        def /(that: Double) =  
            make((this.amount / that).toLong)  
        def /(that: Currency) =  
            this.amount.toDouble / that.amount  
        def from(other: CurrencyZone#AbstractCurrency)  
            : Currency =  
            make(Math.round(  
                other.amount.toDouble * Converter.exchangeRate  
                (other.designation)(this.designation)))  
        private def decimals(n: Long): Int =  
            if (n == 1) 0 else 1 + decimals(n / 10)  
        override def toString =  
            ((amount.toDouble / CurrencyUnit.amount.toDouble)  
            formatted ("%." +decimals(CurrencyUnit.amount)+"f")  
            +" "+designation)  
    }  
    val CurrencyUnit: Currency  
}
```

So the type abstraction has done its job—it prevents us from performing calculations that 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 currencies 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.

20.5 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 take advantage of abstract members. They support a simple yet effective principle for systems structuring: When designing a class make everything that is not yet known into an abstract member. The type system will then drive the development of your model, just as you saw with the currency case study. It does not matter whether the unknown is a type or a method or a variable or a value does not matter. In Scala, all of these can be declared abstract.

Chapter 21

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 to a class, not fields or interfaces.

Scala's answer is implicit parameters and conversions. These can make existing libraries much more pleasant to deal with by letting you leave out tedious, obvious details that obscure the interesting parts of your code. 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.

21.1 Implicit conversions

Before delving into the details of implicits, take a look at a typical example of their use. 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, all you must do is 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 be a fine `RandomAccessSeq[Char]`, except that 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("abc123") exists (_.isDigit)  
res0: Boolean = true
```

But you can also leave out the conversion and *still* get the same behavior:

```
scala> "abc123" exists (_.isDigit)  
res1: 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 expression 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`.

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.

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:

```
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")  
res2: String = x y z
```

The last expression is equivalent to the following one, where the conversion shows up explicitly:

```
scala> printWithSpaces(stringWrapper("xyz"))  
res3: 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.

21.2 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`, where `convert` is some available implicit conversion. 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. With one exception, the compiler will 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 object 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 object 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 every time 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 only inserted if 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 have an object with two implicit conversions:

```
object MyConversions {  
    implicit def stringWrapper(s: String): RandomAccessSeq[Char] = ...  
    implicit def int2string(x: Int): String = ...  
}
```

In your application, you want to make use of the `stringWrapper` conversion, but you don't want integers to be converted automatically to strings by means of the `int2string` conversion. You can achieve this by importing only one conversion, but not the other:

```
import MyConversions.stringWrapper  
... // code making use of stringWrapper
```

In this example, it was important that the implicit conversions had names, because only that way one could selectively import one and not the other.

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 a `String` but need the same content 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.

21.3 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>:5: 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`. So far, the compiler is looking at an ordinary type error. Before giving up, though, 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.

Converting Doubles to Ints might raise some eyebrows, because it's a dubious idea to have something that causes a loss in precision happen invisibly. So this is not really a conversion we recommend. It makes much more sense to go the other way, from some more constrained type to a more general one. For instance, an `Int` can be converted without loss of precision to a `Double`, so an implicit conversion from `Int` to `Double` makes sense. In fact, that's exactly what happens. The `scala.Predef` object, which is implicitly imported into every Scala program, defines implicit conversions that convert “smaller” numeric types to “larger” ones. For instance, you will find in `Predef` the following conversion:

```
implicit def int2double(x: Int): Double = x.toDouble
```

That's why in Scala `Int` values can be stored in variables of type `Double`. There's no special rule in the type system for this; it's just an implicit conversion that gets applied.²

21.4 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

²The Scala compiler backend will treat the conversion specially, however, translating it to a special “`i2d`” bytecode. So the compiled image is the same as in Java.

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 is not a normal Scala type, 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  
res4: Rational = 1/1  
  
scala> oneHalf + 1  
res5: 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
res6: 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’s not syntax! Instead, `->` is a method of the class ArrowAssoc, a class defined inside the standard Scala preamble (`scala.Predef`). The preamble also defines an implicit conversion from Any to ArrowAssoc. When you write `1->"one"`, the compiler inserts a conversion from 1 to ArrowAssoc so that the `->` method can be found. Here are the relevant definitions:

```
package scala
object Predef {
    class ArrowAssoc[A](x: A) {
        def -> [B](y: B): Tuple2[A, B] = Tuple2(x, y)
    }
    implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] =
        new ArrowAssoc(x)
    ...
}
```

This rich wrappers 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. Similarly, if you see a class named RichSomething, *e.g.* RichInt or RichString, that class is likely adding syntax-like methods to type Something.

You have already seen this rich wrappers pattern for the basic types described in [Chapter 5](#). As you can now see, these rich wrappers apply more widely, often letting you get by with a library where programmers in other languages would develop separate domain-specific languages.

21.5 Implicit parameters

The remaining place the compiler inserts implicits is within argument 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 an argument implicitly:

```
scala> implicit val favoriteNumber = 4
favoriteNumber: Int = 4
scala> printSomething
4
```

The compiler sees that the call to `printSomething` is missing an argument list, and that the associated parameter list is marked `implicit`. It then goes looking for an implicit value it can pass, and it finds one in `favoriteNumber`. The resulting call is the same as if `favoriteNumber` were passed explicitly.

It's also possible to have several implicit parameters. In that case you only need to give the `implicit` keyword once; it holds for the whole parameter list that follows:

```
scala> def printMore(implicit x: Int, animal: String) =
|   println(x+" "+animal+"s")
printMore: (implicit Int, String)Unit
scala> implicit val favoriteAnimal = "cat"
favoriteAnimal: java.lang.String = cat
scala> printMore
4 cats
```

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 IllegalArgumentException("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 12](#) 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 provides implicit `orderer` methods for many common types. You can thus use this `maxList` method with a variety of types:

```
scala> maxList(List(1,5,10,3))
res9: Int = 10

scala> maxList(List(1.5, 5.2, 10.7, 3.14159))
res10: 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 custom 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 an implicit 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.

21.6 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 IllegalArgumentException("empty list!")
    case x :: Nil => x
    case x :: rest =>
      val maxRest = maxList2(rest) // (orderer) is implicit
      if (x > maxRest) x        // orderer(x) is implicit
      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 these two insertions of implicits, 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. All uses of `ordered` are implicit. Surprisingly, this coding pattern is actually fairly common. The im-

plicit 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. 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...
```

Because this pattern is common, Scala lets you 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.

21.7 Debugging implicits

Implicits are an extremely powerful feature in Scala, but one which is sometimes difficult to get right and debug. Here are a couple of tips for debugging implicits.

```
object ImplicitsExample {  
    implicit val favoriteNumber = 4  
    def printSomething(implicit x: Int) = println(x)  
    printSomething  
}
```

Figure 21.1: 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 21.2: The example from Figure 21.1, after type checking and insertion of implicits. The implicit parameter is in bold face.

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`. You would wonder why the following does not work:

```
scala> val chars: List[Char] = "xyz"  
<console>:9: error: type mismatch;  
      found   : java.lang.String("xyz")  
      required: List[Char]  
          val chars: List[Char] = "xyz"  
                           ^
```

In that case it helps to write the `stringWrapper` conversion explicitly, to find out what went wrong.

```
scala> val chars: List[Char] = stringWrapper("xyz")  
<console>:9: 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 (such as 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 21.1](#) and [Fig. 21.2](#).

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.

21.8 Conclusion

Implicits are a powerful, code-condensing feature of Scala. This chapter has shown you Scala's rules about implicits, and it has shown you several common programming situations where you can profit from using implicits.

As a word of warning, however, implicits can make code confusing if they are used too frequently. Thus, before adding a new implicit conversion, first ask whether you can achieve a similar effect through other means, such as inheritance, mixin composition, or method overloading. If all of these fail, however, and you feel like a lot of your code is still tedious and redundant, then implicits might just be able to help you out.

Chapter 22

Implementing Lists

Lists have been ubiquitous in this book. Class `List` is probably the most commonly used structured data type in Scala. [Chapter 16](#) showed you how to use lists. 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 genericity concepts in particular. So studying class `List` will deepen your knowledge in these areas.

22.1 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 22.3](#).

```
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, as discussed in [Chapter 19](#). 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.¹

The ‘`::`’ class

Class ‘`::`’, pronounced “cons” for “construct,” represents non-empty lists. It’s named that way in order to support pattern matching with the infix ‘`::`’. You have seen in [Section 16.5](#) 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`). Recall from [Section ??](#) that 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`.

¹To 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

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 drop(n: Int) =
  if (isEmpty) Nil
  else if (n == 0) this
  else tail.drop(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 15](#) on pattern matching.²

```
abstract class Fruit
class Apple extends Fruit
class Orange extends Fruit
```

²The reason `Expr` extends `Product` is to simplify the discussion. You get similar results if `Expr` has no declared supertype.

Now look at what happens when you construct lists of fruits.

```
scala> val fruits = new Apple :: Nil  
fruits: List[Apple] = List(Apple@585fa9)  
scala> val morefruits = new Orange :: fruits  
morefruits: List[Fruit] = List(Orange@cd6798, Apple@585fa9)
```

The `fruits` value is treated as a List of Apples, as expected. However, the definition of `morefruits` 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 `Fruit`, which is the most precise common supertype of the original list element type (*i.e.* `Apple`) and the type of the element to be added (*i.e.* `Orange`).

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]`.

With this formulation of ‘`::`’, you can check how 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 19](#) 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]` thus kills two birds with one stone: It removes a typing problem, and it 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
=           (because :: and ::: are right-associative)
prefix.head :: (prefix.tail ::: this)
=           (because :: binds to the right)
(prefix.tail ::: this) :::(prefix.head)
=           (because ::: binds to the right)
this :::(prefix.tail) :::(prefix.head)
```

22.2 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 `ListBuffer` from its 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.

22.3 The List class in practice

The implementations of list methods given in Section 22.1 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 22.1](#). 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 `tl`.

Since the `ListBuffer` class is contained in package `scala.collection.mutable`, it can access the `tl` 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 `tl` 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 add elements incrementally and then do one `toList` operation at the end. In such cases, no copying is necessary.

22.4 Functional at the outside

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 every time 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

```
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 opts 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.

22.5 Conclusion

Now you have seen how the `List` class is implemented. It is one of the most heavily used data structures in Scala, and it has a refined implementation. The beginning of the story is that `List` has two subclasses, `Nil` and `::`, both of which are case classes. Instead of recursing through this structure, however, many core list methods are implemented using a `ListBuffer`. `ListBuffer`, in turn, is carefully implemented so that it can efficiently build lists without allocated extraneous memory. It is functional from the outside, but uses mutation internally to speed up the common case where a buffer is discarded after `toList` is been called. After studying all of this, you now know the list classes inside and out, and you might have learned an implementation trick or two.

Chapter 23

For-Expressions Revisited

Chapter 16 demonstrated that higher-order functions such as `map`, `flatMap`, `filter` provide powerful constructions for dealing with lists. But sometimes the level of abstraction required by these functions makes a program a bit hard to understand. Here's an example. Say you are given a list of persons, each defined as an instance of a class `Person`. Class `Person` has fields indicating the person's name, whether (s)he is male, and his/her children. Here's the class definition:

```
scala> case class Person(name: String,  
                         isMale: Boolean,  
                         children: Person*)
```

Here's a list of some sample persons:

```
val lara = Person("Lara", false)  
val bob = Person("Bob", true)  
val julie = Person("Julie", false, lara, bob)  
val persons = List(lara, bob, julie)
```

Now, let's say you want to find out the names of all pairs of mothers and their children in that list. Using `map`, `flatMap` and `filter`, you can formulate the following query:

```
scala> persons filter (p => !p.isMale) flatMap (p =>  
|   (p.children map (c => (p.name, c.name))))  
res5: List[(String, String)] = List((Julie,Lara), (Julie,Bob))
```

The query does its job, but it's not exactly trivial to write or understand. Is there a simpler way? In fact, there is. Remember the for-expressions in [Chapter 7](#)? Using a for-expression, the same example can be written as follows:

```
scala> for (val p <- persons; !p.isMale; c <- p.children)
|   yield (p.name, c.name)
res6: List[(String, String)] = List((Julie,Lara), (Julie,Bob))
```

The result of this expression is exactly the same as the previous result. What's more, most readers would agree that the for-expression is much clearer than the previous query which used higher-order functions.

However, the two queries are not as dissimilar as it might seem. In fact, it turns out that the Scala compiler will translate the second query into the first one. More generally, all for-expressions that `yield` a result are translated by the compiler into combinations of invocations of the higher-order methods `map`, `flatMap`, and `filter`. For-loops without `yield` are also translated into a smaller set of higher-order functions: this time it's just `filter` and `foreach`.

In this chapter, you'll find out first about the precise rules of writing for-expressions. After that, you'll see how they can make combinatorial problems easier to solve. Finally, you'll learn how for-expressions are translated, and what this implies for growing the Scala language into new application domains.

23.1 For-Expressions

Generally, a for-expression is of the form

```
for ( seq ) yield expr .
```

Here, `seq` is a sequence of *generators*, *definitions* and *filters*, with semicolons between successive elements. An example is the for-expression

```
for (p <- persons; n = p.name; if (n startsWith "To"))
yield n
```

The for-expression above contains one generator, one definition, and one filter. You can also enclose the sequence in braces instead of parentheses, then the semicolons become optional:

```
for {
    p <- persons           // a generator
    n = p.name              // a definition
    if (n startsWith "To")   // a filter
} yield n
```

A *generator* is of the form

$$pat \leftarrow expr .$$

The expression *expr* typically returns a list, even though you will see later that this can be generalized. The pattern *pat* gets matched one-by-one against all elements of that list. If the match succeeds, the variables in the pattern get bound to the corresponding parts of the element, just the way it is described in [Chapter 15](#). But if the match fails, no `MatchError` is thrown. Instead, the element is simply discarded from the iteration.

In the most common case, the pattern *pat* is just a variable *x*, as in *x* \leftarrow *expr*. In that case, the variable *x* simply iterates over all elements returned by *expr*.

A *definition* is of the form

$$pat = expr .$$

This definition simply introduces *x* as a name for the value of *expr*. So it has the same effect as a `val` definition:

$$\text{val } pat = expr$$

A *filter* is of the form

$$\text{if } expr .$$

Here, *expr* is an expression of type Boolean. The filter drops from the iteration all elements for which *expr* returns false.

Every for-expression starts with a generator. If there are several generators in a for-expression, later generators vary more rapidly than earlier ones. You can verify this easily with the following simple test:

```
scala> for (x <- List(1, 2); y <- List("one", "two"))
| yield (x, y)
res0: List[(Int, java.lang.String)] =
List((1,one), (1,two), (2,one), (2,two))
```

23.2 The N-Queens Problem

A particularly suitable application area of for-expressions are combinatorial puzzles. An example of such a puzzle is the 8-queens problem: Given a standard chess-board, place 8 queens such that no queen is in check from any other (a queen can check another piece if they are on the same column, row, or diagonal). To find a solution to this problem, it's actually simpler to generalize it to chess-boards of arbitrary size. Hence, the problem is to place N queens on a chess-board of $N \times N$ squares, where the size N is arbitrary. We'll start numbering cells at one, so the upper-left cell of an $N \times N$ board has coordinate $(1, 1)$, and the lower-right cell has coordinate (N, N) .

To solve the N-queens problem, note that you need to place a queen in each row. So you could place queens in successive rows, each time checking that a newly placed queen is not in check from any other queens that have already been placed. In the course of this search, it might arrive that a queen that needs to be placed in row k would be in check in all fields of that row from queens in row 1 to $k - 1$. In that case, you need to abort that part of the search in order to continue with a different configuration of queens in columns 1 to $k - 1$.

An imperative solution to this problem would place queens one by one, moving them around on the board. But it looks difficult to come up with a scheme that really tries all possibilities.

A more functional approach represents a solution directly, as a value. A solution consists of a list of coordinates, one for each queen placed on the board. Note, however, that a full solution can not be found in a single step. It needs to be built up gradually, by occupying successive rows with queens.

This suggests a recursive algorithm. Assume that you have already generated all solutions of placing k queens on a board of size $N \times N$, where k is less than N . Each such solution can be presented by a list of length k of coordinates (row, column), where both row and column numbers range from 1 to N . It's convenient to treat these partial solution lists as stacks, where the coordinates of the queen in row k come first in the list, followed by the coordinates of the queen in row $k - 1$, and so on. The bottom of the stack is the coordinate of the queen placed in the first row of the board. All solutions together are then represented as a list of lists, with one element for each solution.

Now, to place the next queen in row $k + 1$, generate all possible extensions of each previous solution by one more queen. This yields another list of

solution lists, this time of length $k + 1$. Continue the process until you have obtained all solutions of the size of the chess-board N . This algorithmic idea is embodied in function `placeQueens` below:

```
def queens(n: Int): List[List[(Int, Int)]] = {
    def placeQueens(k: Int): List[List[(Int, Int)]] =
        if (k == 0)
            List(List())
        else
            for { queens <- placeQueens(k - 1)
                  column <- 1 to n
                  queen = (k, column)
                  if isSafe(queen, queens)
            } yield queen :: queens
    placeQueens(n)
}
```

The outer function `queens` in the program above simply calls `placeQueens` with the size of the board n as its argument. The task of the function application `placeQueens(k)` is to generate all partial solutions of length k in a list. Every element of the list is one solution, represented by a list of length k . So `placeQueens` returns a list of lists.

If the parameter k to `placeQueens` is 0, this means that it needs to generate all solutions of placing zero queens on zero rows. There is exactly one such solution: place no queen at all. This is represented as a solution by the empty list. So if k is zero, `placeQueens` returns `List(List())`, a list consisting of a single element that is the empty list. Note that this is quite different from the empty list `List()`. If `placeQueen` returns `List()`, this means *no solutions*, instead of a single solution consisting of no placed queens.

In the other other case, where k is not zero, all the work of `placeQueens` is done in a for-expression. The first generator of that for-expression iterates through all solutions of placing $k-1$ queens on the board. The second generator iterates through all possible columns on which the k 'th queen might be placed. The third part of the for-expression defines the newly considered queen position to be the pair consisting of row k and each produced column. The fourth part of the for-expression is a filter which checks with `isSafe`

whether the new queen is safe from check of all previous queens (the definition `isSafe` will be discussed a bit later).

If the new queen is not in check from any other queens, it can form part of a partial solution, so `placeQueens` generates with `queen :: queens` a new solution. If the new queen is not safe from check, the filter returns `false`, so no solution is generated.

The only remaining bit is the `isSafe` method which is used to check whether a given queen is in check from any other element in a list of queens. Here is its definition:

```
def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =  
    queens forall (q => !inCheck(queen, q))  
  
def inCheck(q1: (Int, Int), q2: (Int, Int)) =  
    q1._1 == q2._1 || // same row  
    q1._2 == q2._2 || // same column  
    (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // on diagonal
```

The `isSafe` method expresses that a queen is safe with respect to some other queens if it is not in check from any other queen. The `inCheck` method expresses that queens `q1` and `q2` are mutually in check. It returns `true` in one of three cases:

- (1) If the two queens have the same row coordinate,
- (2) If the two queens have the same column coordinate,
- (3) If the two queens are on the same diagonale, that is if the difference between their rows and the difference between their columns are the same.

The first case, that the two queens have the same row coordinate, cannot happen in the application because `placeQueens` already takes care to place each queen in a different row. So one might remove the test without changing the functionality the program as a whole.

23.3 Querying with For-Expressions

The for-notation is essentially equivalent to common operations of database query languages. For instance, say we are given a database `books`, represented as a list of books, where `Book` is defined as follows.

```
case class Book(title: String, authors: String*)
```

Here is a small example database, represented as an in-memory list:

```
val books: List[Book] = List(  
    Book("Structure and Interpretation of Computer Programs",  
        List("Abelson, Harold", "Sussman, Gerald J.")),  
    Book("Principles of Compiler Design",  
        List("Aho, Alfred", "Ullman, Jeffrey"))),  
    Book("Programming in Modula-2",  
        List("Wirth, Niklaus"))),  
    Book("Introduction to Functional Programming"),  
        List("Bird, Richard"))),  
    Book("The Java Language Specification",  
        List("Gosling, James", "Joy, Bill", "Steele, Guy", "Bracha, Gilad"))))
```

Then, to find the titles of all books whose author's last name is "Gosling":

```
scala> for (b <- books; a <- b.authors if a startsWith "Gosling")  
| yield b.title  
res0: List[String] = List(The Java Language Specification)
```

Or, to find the titles of all books that have the string "Program" in their title:

```
scala> for (b <- books if (b.title indexOf "Program") >= 0)  
| yield b.title  
res1: List[String] = List(Structure and Interpretation of Computer Programs,
```

Or, to find the names of all authors that have written at least two books in the database.

```
scala> for (b1 <- books; b2 <- books if b1 != b2;  
|         a1 <- b1.authors; a2 <- b2.authors if a1 == a2)  
| yield a1  
res2: List[String] = List(Ullman, Jeffrey, Ullman, Jeffrey)
```

The last solution is not yet perfect, because authors will appear several times in the list of results. You still need to remove duplicate authors from result lists. This can be achieved with the following function.

```

scala> def removeDuplicates[A](xs: List[A]): List[A] = {
|   if (xs.isEmpty) xs
|   else xs.head :: 
|     removeDuplicates(xs.tail filter (x => x != xs.head))
| }
removeDuplicates: [A](List[A])List[A]

scala> removeDuplicates(res2)
res3: List[java.lang.String] = List(Ullman, Jeffrey)

```

Maybe it's worth noting that the last expression in method `removeDuplicates` can be equivalently expressed using a for-expression:

```
xs.head :: removeDuplicates(for (x <- xs.tail if x != xs.head) yield x)
```

23.4 Translation of For-Expressions

Every for-expression can be expressed in terms of the three higher-order functions `map`, `flatMap` and `filter`. Here is the translation scheme, which is also used by the Scala compiler.

Translating for-expressions with one generator

First, assume you have a simple for-expression

```
for (x <- expr1) yield expr2
```

where x is a variable. Such an expression is translated to

```
expr1.map(x => expr2) .
```

Translating for-expressions starting with a generator and a filter

Now, let's consider for-expressions that combine a leading generator with some other elements. A for-expression of the form

```
for (x <- expr1 if expr2) yield expr3
```

is translated to

```
for (x <- expr1 filter (x => expr2)) yield expr3
```

The translation above gives in turn another for-expression which is shorter by one element than the original for-expression because an `if` element has been turned into an application of `filter` on the first generator expression. The translation then continues with this second expression, so in the end one obtains:

```
expr1 filter (x => expr2) map (x => expr3)
```

The same translation scheme also applies if there are further elements following the filter. Let's say `seq` is an arbitrary sequence of generators, definitions and filters. Then

```
for (x <- expr1 if expr2; seq) yield expr3
```

is translated to:

```
for (x <- expr1 filter expr2; seq) yield expr3
```

Then translation continues with that second expression, which is again shorter by one element than the original one.

Translating for-expressions starting with two generators

The next case handles for-expression that start with two filters, as in:

```
for (x <- expr1; y <- expr2; seq) yield expr3
```

Again, let's assume that `seq` is an arbitrary sequence of generators, definitions and filters. In fact, `seq` might also be empty, and in that case there would not be a semicolon after `expr2`. The translation scheme stays the same in each case. The for-expression above is translated to an application of `flatMap`:

```
expr1.flatMap(x => for (y <- expr2; seq) yield expr3)
```

This time, there is another for-expression in the closure passed to `flatMap`. That for-expression (which is again simpler by one element than the original) is in turn translated with the same rules.

The three translation schemes given so far are sufficient to translate all for-expressions that contain just generators and filters, and where generators

bind only simple variables. Take for instance the query *find all authors who have published at least two books* from [Section 23.3](#):

```
for (b1 <- books; b2 <- books if b1 != b2;
      a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
yield a1
```

This query translates to the following `map/flatMap/filter` combination:

```
books flatMap (b1 =>
  books filter (b2 => b1 != b2) flatMap (b2 =>
    b1.authors flatMap (a1 =>
      b2.authors filter (a2 => a1 == a2) map (a2 =>
        a1))))
```

The translation scheme presented so far does not yet handle generators that bind whole patterns instead of simple variables. It also does not yet cover definitions. These two aspects will be explained in the next two sub-sections.

Translating patterns in generators

The translation scheme becomes more complicated if the left hand side of generator is a pattern *pat* other than a simple variable. Still relatively easy to handle is the case where the for-expression binds a tuple of variables. In that case, almost the same scheme as for single variables applies. A for-expression of the form

```
for ((x1, ..., xn) <- expr1) yield expr2
```

translates to

```
expr.map { case (x1, ..., xn) => expr2 }
```

Things become a bit more involved if the left hand side of the generator is an arbitrary pattern *pat* instead of a single variable or a tuple. In this case,

```
for (pat <- expr1) yield expr2
```

translates to

```
expr1 filter {
    case pat => true
    case _ => false
} map {
    case pat => expr2
}
```

That is, the generated items are first filtered and only those that match *pat* are mapped. Therefore, it's guaranteed that a pattern-matching generator will never throw a `MatchError`.

The scheme above only treated the case where the for-expression contains a single pattern-matching generator. Analogous rules apply if the for-expression contains other generators, filters, or definitions. Because these additional rules don't add much new insight, they are omitted from discussion here. If you are interested, you can look them up in the Scala Language Specification [Ode08].

Translating definitions

The last missing situation is where a for-expression contains embedded definitions. Here's a typical case:

```
for (x <- expr1; y = expr2; seq) yield expr3
```

Assume again that *seq* is a (possibly empty) sequence of generators, definitions, and filters. This expression is translated to the following one:

```
for ((x, y) <- for (x <- expr1) yield (x, expr2); seq)
    yield expr3
```

So you see that *expr₂* is evaluated each time there is a new *x* value being generated. This re-evaluation is necessary, because *expr₂* might refer to *x* and so needs to be re-evaluated for changing values of *x*. For you as a programmer the conclusion is that it's probably not a good idea to have definitions embedded in for-expressions that do not refer to variables bound by some preceding generator, because re-evaluating such expressions would be wasteful. For instance, instead of

```
for (x <- 1 to 1000; y = expensiveComputationNotInvolvingX)
    yield x * y
```

it's usually better to write

```
{ val y = expensiveComputationNotInvolvingX
  for (x <- 1 to 1000) yield x * y
}
```

Translating for-loops

The previous subsections have explained how for-expressions that contain a `yield` are translated. What about for-loops that simply perform a side-effect without returning anything? Their translation is similar, but simpler than for-expressions. In principle, wherever the previous translation scheme used a `map` or a `flatMap` in the translation, the translation scheme for for-loops uses just a `foreach`. For instance, the expression

```
for (x <- expr1) body
```

translates to

```
expr1 foreach (x => body)
```

A larger example is the expression

```
for (x <- expr1; if expr2; y <- expr3) body
```

This expression translates to

```
expr1 filter (x => expr2) foreach (x =>
  expr3 foreach (y => body))
```

For example, the following expression sums up all elements of a matrix represented as a list of lists:

```
var sum = 0
for (xs <- xss; x <- xs) sum += x
```

This loop is translated into two nested `foreach` applications:

```
var sum = 0
xss foreach (xs =>
  xs foreach (x =>
    sum += x))
```

23.5 Going the other way

The previous section has shown that for-expressions can be translated into applications of the higher-order functions `map`, `flatMap`, and `filter`. In fact, one could equally well go the other way: Every application of a `map`, a `flatMap`, or a `filter` can be represented as a for-expression. Here are implementations of the three methods in terms of for-expressions. The methods are contained in an object `Demo`, to distinguish them from the standard operations on `Lists`. To be concrete, the three functions all take a `List` as parameter, but the translation scheme would work just as well with other collection types.

```
object Demo {  
    def map[A, B](xs: List[A], f: A => B): List[B] =  
        for (x <- xs) yield f(x)  
  
    def flatMap[A, B](xs: List[A], f: A => List[B]): List[B] =  
        for (x <- xs; y <- f(x)) yield y  
  
    def filter[A](xs: List[A], p: A => Boolean): List[A] =  
        for (x <- xs if p(x)) yield x  
}
```

Not surprisingly, the translation of the for-expression in the body of `Demo.map` will produce a call to `map` in class `List`. Similarly, `Demo.flatMap` and `Demo.filter` translate to `flatMap` and `filter` in class `List`.

So this little demonstration has shown that for-expressions really are equivalent in their expressiveness to applications of the three functions `map`, `flatMap`, and `filter`.

23.6 Generalizing For

Because the translation of for-expressions only relies on the presence of methods `map`, `flatMap`, and `filter`, it is possible to apply the for-notation to a large class of datatypes.

You have already seen for-expressions over lists and arrays. These are supported because lists, as well as arrays, define operations `map`, `flatMap`, and `filter`. For-loops over these data types are also possible because they define a `foreach` method as well.

Besides lists and arrays, there are also many other types in the Scala standard library that support the same four methods and therefore allow for-expressions. Examples are ranges, iterators, streams, and all implementations of sets. It's also perfectly possible for your own datatypes to support for-expressions by defining the necessary methods. To support the full range of for-expressions and for-loops, you need to define `map`, `flatMap`, `filter`, and `foreach` as methods of your datatype. But it's also possible to define a subset of these methods, and thereby support a subset of all possible for-expressions or loops. Here are the precise rules:

- If your type defines just `map`, it allows for-expressions consisting of a single generator.
- If it defines `flatMap` as well as `map`, it allows for-expressions consisting of several generators.
- If it defines `foreach`, it allows for-loops (both with single and multiple generators).
- If it defines `filter`, it allows for filter expressions starting with an `if` in the for-expression.

The translation of for-expression happens before type checking. This allows for maximal flexibility, because it is only required that the result of expanding a for-expression type-checks. Scala defines no typing rules for the for-expressions themselves, and does not require `map`, `flatMap` and friends to have any particular type.

Nevertheless, there is a typical setup which captures the most common intention of the higher order methods to which for-expressions translate. Say you have a parameterized class `C` which typically would stand for some sort of collection. Then it's quite natural to pick the following type signatures for `map`, `flatMap`, `filter`, and `foreach`:

```
class C[A] {  
    def map[B](f: A => B): C[B]  
    def flatMap[B](f: A => C[B]): C[B]  
    def filter(p: A => Boolean): C[A]  
    def foreach(b: A => Unit): Unit  
}
```

That is, the `map` function takes a function from the collection's element type `A` to some other type `B`. It produces a new collection of the same kind `C`, but with `B` as the element type. The `flatMap` method takes a function `f` from `A` to some `C`-collection of `B`'s and produces a `C`-collection of `B`'s. The `filter` method takes a predicate function from the collection's element type `A` to `Boolean`. It produces a collection of the same type as the one on which it is invoked. Finally, the `foreach` method takes a function from `A` to `Unit`, and produces a `Unit` result.

Concentrating on just the first three functions, the following facts are noteworthy. In functional programming, there's a general concept called a *monad* which can explain a large number of types with computations, ranging from collections, computations with state and I/O, backtracking computations, to transactions, to name but a few. One can formulate functions `map`, `flatMap` and `filter` on a monad, and, if one does, they end up having exactly the types given above. Furthermore, one can characterize every monad by `map`, `flatMap` and `filter`, plus a "unit" constructor that produces a monad from an element value. In an object-oriented language, this "unit" constructor is simply an instance creation or a factory method. Therefore, `map`, `flatMap` and `filter` can be seen as an object-oriented version of the functional concept of monad. Because for-expressions are equivalent to applications of these three methods, they can be seen as syntax for monads.

All this suggests that the concept of for-expression is something more general than just iteration over a collection, and indeed it is. For instance, for-expressions also play an important role in asynchronous I/O, or as an alternative notation for optional values. Watch out in the Scala libraries for occurrences of `map`, `flatMap`, and `filter` – wherever they are present, for-expressions suggest themselves as a concise way of manipulating elements of the type.

23.7 Conclusion

In this chapter, you have been given a peek under the hood of for-expressions and for-loops. You have learned that they translate into applications of a standard set of higher-order methods. As a consequence of this, you have seen that for-expressions are really much more general than mere iterations over collections, and that you can design your own classes to support them.

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 `Some` 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's 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 15](#) 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(str: String): Option[(String, String)] = {
        val parts = str 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 object that can be applied to arguments in parentheses in the same way a method is applied. 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 `str` 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 = ...
x 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

The `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): String = s + s
  def unapply(s: String): Option[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.

```
objectUpperCase {  
    def unapply(s: String): Boolean = 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 whose user part consists of two times the same string in uppercase letters. For instance:

```
scala> test2("DIDI@hotmail.com")  
res0: java.lang.String = match: DI in domain hotmail.com  
  
scala> test2("DIDO@hotmail.com")  
res1: java.lang.String = no match  
  
scala> test2("didi@hotmail.com")  
res2: 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 15.2](#): The form `x @UpperCase()` associates the variable `x` with the pattern matched by `UpperCase()`. For

instance, in the first test above, `x` was bound to "DI", because that was the value against which the `UpperCase()` pattern got matched.

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 15.2](#) 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.

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 result type of an `unapplySeq` must conform to some type `Option[Seq[T]]`, where the element type `T` is arbitrary. As you have seen in [Chapter 17](#), `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.

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:

```
def isTomInDotCom(s: String): Boolean = s match {
    case EMail("tom", Domain("com", _*)) => true
    case _ => false
}
```

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, topdom, subdoms @ _) = s
name: String = tom
topdom: String = ch
subdoms: Seq[String] = List(epfl, support)

```

24.5 Extractors and sequence patterns

You have seen in [Section 15.2](#) 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 Regular Expressions

One particularly useful application area of extractors are regular expressions. Like Java, Scala treats regular expressions in the library. But extractors make it much nicer to interact with them.

Forming regular expressions

Scala inherits its regular expression syntax from Java, which in turn inherits most of the features of Pearl. We assume you know that syntax already; if

not there are many accessible tutorials, starting with the JavaDoc documentation of class `java.util.regex.Pattern`. Here are just some examples that should be enough as refreshers:

`ab?`

An ‘a’, possibly followed by a ‘b’.

`[a-dA-D]\w*`

A word starting with a letter between a and d in lower or upper case, followed by a sequence of zero or more “word characters” denoted by `\w`. (A word character is a letter, digit, or underscore.)

`\d+`

A number consisting of one or more digits represented by `\d`.

`(-)?(\d+)(\.\d*)`

A number consisting of an optional minus sign, followed by one or more digits, optionally followed by a period and zero or more digits. The number contains three *groups*, i.e. the minus sign, the part before the decimal point, and the fractional part including the decimal point. Groups are enclosed in parentheses.

Scala’s regular expression class is contained in package `scala.util.matching`.

```
scala> import scala.util.matching.Regex
```

A new regular expression value is created by passing a string to the `Regex` constructor. For instance:

```
scala> val Decimal = new Regex("(-)?(\\d+)(\\.\\d*)?")
Decimal: scala.util.matching.Regex = (-)?(\\d+)(\\.\\d*)?
```

Note that, compared to the regular expression for decimal numbers given previously, every backslash appears twice in the string above. This is because in Java and Scala a single backslash is an escape character in a string literal, not a regular character that shows up in the string. So instead of ‘\’ you need to write ‘\\’ to get a single backslash in the string.

If a regular expression contains many backslashes this might be a bit painful to write and to read. Scala’s raw strings provide an alternative. As

you have seen in [Chapter 5](#), a raw string is a sequence of characters between triple quotes. The difference between a raw and a normal string is that all characters in a raw string appear exactly how they are typed. This includes backslashes, which are not treated as escape characters. So you could write equivalently and somewhat more legibly:

```
scala> val Decimal = new Regex("""(-)?(\d+)(\.\d*)?""")
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?
```

As you can see from the interpreter's output, the generated result value for `Decimal` is exactly the same as before.

Another, even shorter way to write a regular expression in Scala is this:

```
val Decimal = """(-)?(\d+)(\.\d*)?""".r
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?
```

In other words, simply append a `.r` to a string to obtain a regular expression. This is possible because there is a method named `r` in class `java.runtime.RichString` which converts a string to a regular expression. The method is defined as follows.

```
package scala.runtime
class RichString(self: String) ... {
  ...
  def r = new Regex(self)
}
```

Searching for regular expressions

You can search for occurrences of regular expression in a string using several different operators:

<code>str findFirst regex</code>	Finds first occurrence of regular expression <code>regex</code> in string <code>str</code> , returning the result in an <code>Option</code> type.
<code>str findAll regex</code>	Finds all occurrences of regular expression <code>regex</code> in string <code>str</code> , returning the results in an <code>Iterator</code> .
<code>str findPrefix regex</code>	Finds an occurrence of regular expression <code>regex</code> at the start of string <code>str</code> , returning the result in an <code>Option</code> type.

For instance, you could define the input sequence below and then search decimal numbers in it:

```
scala> val input = "for -1.0 to 99 by 3"
input: java.lang.String = for -1.0 to 99 by 3
scala> for (s <- input findAll Decimal) println(s)
-1.0
99
3
scala> input findFirst Decimal
res4: Option[String] = Some(1.0)
scala> input findPrefix Decimal
res5: Option[String] = None
```

Extracting with regular expressions

What's more, every regular expression in Scala defines an extractor. The extractor is used to identify substrings that are matched by the groups of the regular expression. For instance, you could decompose a decimal number string as follows:

```
val Decimal(sign, integerpart, decimalpart) = "-1.23"
sign: String = -
integerpart: String = 1
decimalpart: String = .23
```

What happens here is that the `Decimal` regular expression value defines an `unapplySeq` method. That method matches every string that corresponds to the regular expression syntax for decimal numbers. If the string matches, the parts that correspond to the three groups in the regular expression `(-)?(\d+)(\.\d*)?` are returned as elements of the pattern and are then matched by the three pattern variables `sign`, `integerpart`, and `decimalpart`. If a group is missing, the element value is set to `null`, as can be seen in the following example:

```
scala> val Decimal(sign, integerpart, decimalpart) = "1.0"
sign: String = null
integerpart: String = 1
decimalpart: String = .0
```

It's also possible to mix extractors with regular expression searches in a `for`-expression. For instance, the following expression decomposes all decimal numbers it finds in the input string:

```
scala> for (val Decimal(s, i, d) <- input findAll Decimal)
|   println("sign: "+ s +", integer: "+ i +", decimal: "+ d)
sign: -, integer: 1, decimal: .0
sign: null, integer: 99, decimal: null
sign: null, integer: 3, decimal: null
```

24.8 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.

Extractors are one more element in your tool box that let you define flexible library abstractions. They are used heavily in Scala's libraries, for instance to enable convenient regular expression matching.

Chapter 25

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 29](#).

25.1 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 15](#)), and implicits ([Chapter 21](#)) 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—[meta-programming](#) tools—to pay attention to these annotations and do something useful.

25.2 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 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 15](#)). 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 of 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 annotations that have optional arguments. These arguments are optional, and they can be specified in any order. To keep things simple, the part to the right-hand side of the equals sign must be a constant.

25.3 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 heed 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 30](#) 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. For example, on Java, you get the same behavior as if you wrote

the field in Java code and marked it with the Java `volatile` modifier. Specific behavior aside, the general rule is that 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 26](#)), 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 29](#)).

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 add this annotation to a field, the compiler will automatically generate getters and setters for you. If you annotate a field 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, because in Scala code you can access the fields directly. This feature is intended to support frameworks that expect regular getters and setters, and typically you do not compile the framework and the code that uses it at the same time.

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 15](#) for details.

25.4 Conclusion

This chapter has described how to use annotations, and how to use several standard annotations.

Chapter 26

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.

26.1 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. Plain XML is less structured than the objects of a programming language, though, as it lacks a type system to organize the data in a more, well, structured way.¹

Semi-structured data is very helpful any time you need to serialize program data for saving in a file or shipping across a network. Instead of converting structured data all the way down to bytes, you convert it to and from semi-structured data. You then use pre-existing library routines to convert between semi-structured data and binary data, saving your time for more important problems.

There are many forms of semi-structured data, but XML is the most widely used on the Internet. There are XML tools on most operating systems, and most programming languages have XML libraries available. Its popularity is self-reinforcing. The more tools and libraries are developed

¹There are type systems for XML, such as XML Schemas, but they are beyond the scope of this book.

in response to XML's popularity, the more likely software engineers are to choose XML as part of their formats. If you write software that communicates over the Internet, then sooner or later you will need to interact with some service that speaks XML.

For all of these reasons, Scala includes special support for processing XML. This chapter shows you Scala's support for: constructing XML, processing it with regular methods, and processing it with Scala's pattern matching. In addition to these nuts and bolts, the chapter shows along the way several common idioms for using XML in Scala.

26.2 XML overview

XML is built out of two basic elements, text and tags.² Text is as usual any sequence of characters. Tags, written like `<pod>`, consist of a less-than sign, an alphanumeric label, and a greater than sign. Tags can be *open* or *close* tags. A close tag looks just like an open tag except that it has a slash just before the tag's label, like this: `</pod>`.

Open and close tags must match each other, just like parentheses. Any open tag must eventually be followed by a close tag with the same label. Thus the following is illegal:

```
// Illegal XML
One <pod>, two <pod>, three <pod> zoo
```

Further, the contents of any two matching tags must itself be valid XML. You cannot have two pairs of matching tags overlap each other:

```
// Also illegal
<pod>Three <peas> in the </pod> </peas>
```

You have to write it like this:

```
<pod>Three <peas></peas> in here</pod>
```

Since tags are required to match in this way, XML is structured as nested *elements*. Each pair of a start tag and an end tag gives an element, and elements are nested within each other. In the above example, the entirety

²The full story is more complicated, but this is enough to be effective with XML.

of `<pod>Three...here</pod>` is an element, and `<peas></peas>` is an element nested within it.

Those are the basics. Two other things you should know are that, first, there is a shorthand notation for an open tag followed immediately by its matching close tag. Simply write one tag with a slash put after the tag's label. Such a tag is called an *empty tag*. Using an empty tag, the previous example could just as well be written as follows:

```
<pod>Three <peas/> in here</pod>
```

Second, open tags can have *attributes* attached to them. An attribute is a name-value pair written with an equals sign in the middle. The attribute itself is plain, unstructured text surrounded by either double quotes ("") or single quotes (''). Attributes look like this:

```
<pod peas="3" strings="true"/>
```

26.3 XML Literals

Scala lets you type in XML as a **literal** anywhere that an expression is valid. Simply type an open tag and then continue writing XML content. The compiler will go into an XML-input mode and will read content as XML until it sees the end tag matching the open tag you started with.

```
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 element with 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 are not restricted to writing out the exact XML you want, character for character. You can evaluate Scala code in the middle of an XML literal by using curly braces {} as an escape. Here is a simple example:

```
scala> <a> {"hello" + ", world"} </a>
res1: scala.xml.Elem = <a> hello, world </a>
```

A braces escape can include arbitrary Scala content, including further XML literals. Thus, as the nesting level increases, your code can switch back and forth between XML and ordinary Scala code.

```
scala> val yearMade = 1955
yearMade: Int = 1955
scala> <a> { if (yearMade < 2000) <old>{yearMade}</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. Note in the above example that “nothing” as an XML node is denoted with `xml.NodeSeq.Empty`.

An expression inside a brace escape does not have 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>
```

To contrast, if you create XML with low-level string operations, you will run into traps such as the following:

```
scala> "<a>" + "</a>potential security hole<a>" + "</a>"
res5: java.lang.String = <a></a>potential security
hole<a></a>
```

26.4 Serialization

You have now seen enough of Scala's XML support to write the first part of a serializer: conversion from internal data structures to XML. All you need for this are XML literals and their brace escapes.

As an example, suppose you are implementing a database to keep track of your extensive collection of vintage Coca-Cola thermometers. You might make the following internal class to hold entries 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
}
```

This is a straightforward, data-heavy class that holds various pieces of information such as when the thermometer was made, when you got it, and how much you paid for it.

To convert instances of this class to XML, simply add a `toXML` method that uses XML literals and brace escapes, like this:

```
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 #5"  
|   val yearMade = 1952  
|   val dateObtained = "March 14, 2006"  
|   val bookPrice = 2199  
|   val purchasePrice = 500    // sucker!  
|   val condition = 9  
| }  
therm: CCTherm = hot dog #5  
scala> therm.toXML  
res6: scala.xml.Elem =  
<cctherm>  
            <description>hot dog #5</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>
res7: scala.xml.Elem = <a> }}brace yourself!{{ </a>
```

26.5 Taking XML apart

Among the many methods available for the XML classes, there are three 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. By calling the `text` method on any XML node you retrieve all of the text within that node, minus any element tags.

```
scala> <a>Sounds <tag/> good</a>.text
res8: String = Sounds good
```

Any encoded characters are decoded automatically.

```
scala> <a> input ---&gt; output </a>.text
res9: 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"
res10: 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"
res11: scala.xml.NodeSeq =
scala> <a><b><c>hello</c></b></a> \\ "c"
res12: 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"
res13: scala.xml.NodeSeq = Joe
scala> joe \ "@serial"
res14: scala.xml.NodeSeq = 123
```

26.6 Deserialization

Using the previous methods for taking XML apart, you can now write the dual of a serializer, a parser from XML back into your internal data structures. For example, you can parse back a CCTherm instance by using the following code:

```
def fromXML(node: xml.Node): CCTherm =
  new CCTherm {
    val description  = (node \ "description").text
    val yearMade    = (node \ "yearMade").text.toInt
    val dateObtained = (node \ "dateObtained").text
    val bookPrice   = (node \ "bookPrice").text.toInt
    val purchasePrice = (node \ "purchasePrice").text.toInt
    val condition    = (node \ "condition").text.toInt
  }
```

This code searches through an input XML node, node, to find each of the six pieces of data needed to specify a CCTherm. The data that is text is extracted with .text and left as is. Here is this method in action:

```
scala> val node = therm.toXML
node: scala.xml.Elem =
```

```
<cctherm>
    <description>hot dog #5</description>
    <yearMade>1952</yearMade>
    <dateObtained>March 14, 2006</dateObtained>
    <bookPrice>2199</bookPrice>
    <purchasePrice>500</purchasePrice>
    <condition>9</condition>
</cctherm>

scala> fromXML(node)
res15: CCTherm = hot dog #5
```

26.7 Loading and saving

There is one last part needed to write a data serializer: conversion between XML and streams of bytes. This last part is the easiest, because there are library routines that will do it all for you. You simply have to call the right routine on the right data.

To convert XML to a string, all you need is `toString()`. The presence of a workable `toString` is why you can experiment with XML in the Scala shell. However, it is better to use a library routine and convert all the way to bytes. That way, the resulting XML can include a directive that specifies which character encoding was used. If you encode the string to bytes yourself, then the onus is on you to keep track of the character encoding.

To convert from XML to a file of bytes, you can use the `XML.saveFull` command. The important pieces you must choose are a file name, a node to be saved, and a character encoding. The fourth argument is whether to write an XML declaration at the top that includes the character encoding. The fifth argument is the “document type” of this XML, a subject beyond the scope of this chapter. 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 #5</description>
```

```
<yearMade>1952</yearMade>
<dateObtained>March 14, 2006</dateObtained>
<bookPrice>2199</bookPrice>
<purchasePrice>500</purchasePrice>
<condition>9</condition>
</cctherm>
```

Loading is simpler than saving, because the file includes everything the loader needs to know. Simply call `XML.loadFile` on a file name:

```
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.

26.8 Pattern matching on XML

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! " + blahblah
|   case _ => "no! "
| }
res16: 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! "
| }
res17: 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 cont:

```
scala> <a></a> match {
|   case <a>{contents @ _*}</a> => "yes! " + contents
|   case _ => "no! "
| }
res18: 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 #5</description>  
      <yearMade>1952</yearMade>  
      <dateObtained>March 14, 2006</dateObtained>  
      <bookPrice>2199</bookPrice>  
      <purchasePrice>500</purchasePrice>  
      <condition>9</condition>  
    </cctherm>  
    <cctherm>  
      <description>Sprite Boy</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:

```
catalog match {  
  case <catalog>{therms @ _*}</catalog> =>  
    for (therm <- therms)  
      println("processing: " +  
             (therm \ "description").text)  
}  
  
processing:  
processing: hot dog #5  
processing:  
processing: Sprite Boy
```

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>` tag. You can describe this subset using the pattern `<cctherm>{_*}</cctherm>`, and you can restrict the for expression to iterating over items that match that pattern.

```
catalog match {
    case <catalog>{therms @ _*}</catalog> =>
        for (therm @ <cctherm>{_*}</cctherm> <- therms)
            println("processing: " +
                    (therm \ "description").text)
}
```



```
processing: hot dog #5
processing: Sprite Boy
```

26.9 Conclusion

This chapter has only scratched the surface of what you can do with XML. There are many other extensions, libraries, and tools you could learn about, some customized for Scala, some made for Java but usable in Scala, and some language-neutral. What you should walk away from this chapter with is how to use semi-structured data for interchange, and how to access semi-structured data via Scala's XML support.

Chapter 27

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 it.

27.1 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 27.1: 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 27.2: A simple Recipe class and an example recipe.

```
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 27.1](#) and [Figure 27.2](#).

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): Option[String] =  
        allFoods.find(_.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").get  
apple: Food = Apple  
  
scala> SimpleBrowser.recipesUsing(apple)  
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 is no big news, but it becomes very useful when you consider abstraction.

27.2 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 a 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) {
    //...
}
```

the 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, allRecipes, and allCategories, 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").get
apple: Food = Apple
```

```
scala> SimpleBrowser.recipesUsing(apple)
res1: List[Recipe] = List(fruit salad)
```

Now you can create a second database, and use the same browser class 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 StudentBrowser extends Browser {
    val database = StudentDatabase
}
```

27.3 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])
    def allCategories: List[FoodCategory]
}
```

Now the `Database` class can mix in this trait instead of defining `FoodCategory` and `allCategories` itself:

```
abstract class Database extends FoodCategories {
    // ...
}
```

Continuing in this way, you might try and divide SimpleDatabase into two traits, one for foods and one for recipes. An easily fixed problem arises when you do.

```
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`.

There is a way you can tell this to the compiler, however. 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 specifies the requirements on any concrete 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, it is enough to specify a self type of `SimpleFoods`.

```
trait SimpleRecipes {
    this: SimpleFoods =>
```

```
object FruitSalad extends Recipe{
    "fruit salad",
    List(Apple, Pear), // now Pear is in scope
    "Mix it all together.")

def allRecipes = List(FruitSalad)
}
```

Given the new self type, `Pear` is now available. Implicitly, the reference to `Pear` is thought of as `this.Pear`. This is safe, because any *concrete* class that mixes in `SimpleRecipes` must also be a subtype of `SimpleFoods`, which means that `Pear` will be a member. Abstract subclasses and traits do not have to follow this restriction, but since they cannot be instantiated with `new`, there is no risk that the `this.Pear` will fail.

27.4 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  
$
```

27.5 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.

27.6 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 28

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.

28.1 Equality in Scala

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` have exactly the same characters in the same order.

Scala also has an equality like ‘==’ in Java, but it is not used much. That kind of equality, written “`x eq y`,” is true if `x` and `y` reference the same object. The ‘==’ equality is reserved in Scala for the “natural” equality of each type. For value types, ‘==’ is value comparison, just like in Java. 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 == (that: Any): Boolean =  
  if (null eq this) (null eq that) else (this equals that)
```

28.2 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. Mandana 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.mutable.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@62d74e
p2: Point = Point@254de0

scala> val q = new Point(2, 3)
q: Point = Point@349f8a

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.Set[Point] =
Set(Point@62d74e)

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@254de0
```

Now, repeating the first comparison but with the alias p2a instead of p2 you get:

```
scala> p1 equals p2a
res3: 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

¹If you write a lot of Java then you might expect the argument to this method to be type `Object` instead of type `Any`. Don't worry about it. It is the same `equals` method. The compiler simply makes it appear to have type `Any`.

pattern match. It first tests whether the other object is also of type Point. 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 method 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.

```
scala> val p1, p2 = new Point(1, 2)
p1: Point = Point@670f2b
p2: Point = Point@14f7c0

scala> HashSet(p1) contains p2
res4: 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 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 trouble spots 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:

```
scala> val p1 = new Point(1, 2)
p1: Point = Point@2b
scala> val coll = HashSet(p1)
coll: scala.collection.mutable.Set[Point] = Set(Point@2b)
scala> 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
res7: 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
```

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 when `equals` and `hashCode` depend on mutable state, it causes problems for potential users. If they put such objects into collections, they have to be careful never to modify the depended-on state, and this is tricky. If you need a comparison that takes the current state of an object into account, you should usually 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.*

The definition of equals developed so far for class Point 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 type Color.

```
object Colors {  
    sealed abstract class Color  
    case object red extends Color  
    case object green extends Color  
    case object blue extends Color  
}
```

Class ColoredPoint reimplements equals to also take the new color field into account:

```
class ColoredPoint(x: Int, y: Int, val color: Colors.Color)
  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, Colors.red)
cp: ColoredPoint = ColoredPoint@2b

scala> p equals cp
res8: Boolean = true

scala> cp equals p
res9: 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
res10: Boolean = true

scala> HashSet[Point](cp) contains p
res11: 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: Colors.Color)
  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, Colors.red)
redp: ColoredPoint = ColoredPoint@2b

scala> val bluep = new ColoredPoint(1, 2, Colors.blue)
bluep: ColoredPoint = ColoredPoint@2b
```

Taken individually, `redp` is equal to `p` and `p` is equal to `bluep`:

```
scala> redp == p
res12: Boolean = true

scala> p == bluep
res13: Boolean = true
```

However, comparing `redp` and `bluep` yields `false`:

```
scala> redp == bluep
res14: Boolean = false

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: Colors.Color)
    extends Point(x, y) {
    override def equals(other: Any) = other match {
        case that: ColoredPoint =>
            (this.color == that.color) && super.equals(that)
        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):

```
scala> val p1a = new Point(1, 1) { override val y = 2 }
p1a: Point = $anon$1@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 =>
            (that.isComparable(this)) &&
            (this.x == that.x) && (this.y == that.y)
        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: Colors.Color)
  extends Point(x, y) {
  override def hashCode = super.hashCode * 41 + color.hashCode
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (that.isComparable(this)) &&
        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 any point 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 equality 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 `that` 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.

28.3 Conclusion

In retrospect, defining a correct implementation of `equals` has been surprisingly subtle. You must be careful about the type signature, you must override `hashCode`, you should avoid dependencies on mutable state, and you should implement and use an `isComparable` method if your class is non-final. Given how difficult it is to implement a correct quality method, you might prefer to define your classes of comparable objects as case classes. That way, the Scala compiler will add an `equals` method with the right properties automatically.

Chapter 29

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.

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.

29.1 Using Scala from Java

Most of the time you can 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 run time, 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 11.2](#).

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.

For every Scala singleton object, the compiler creates a Java 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 all the methods and fields of the Scala singleton object. The Java class also has a single static field named MODULE\$ to hold the one instance of the class that is created at run time.

As a full 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();  
}
```

That's the translation for the general case. An important special case is if you have a “standalone” singleton object, one 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.

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. In the general case it is not practical. 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.

29.2 Annotations

Scala's general annotations system is discussed in [Chapter 25](#). 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 first 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 deprecated Scala methods.

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 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 runtime 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

¹The reason it all works is that Java bytecode verifier does not check the declarations, anyway! The Java compiler checks, but not the verifier.

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;  
    public int $tag();  
}  
$
```

Note that the `read()` method is annotated as throwing an `IOException`.

Java annotations

Existing annotations from Java frameworks can be used directly in Scala code. Any Java framework will see the annotations you write just as if you were writing in Java.

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.assertEquals

class SetTest {
    @Test
    def testMultiAdd {
        val set = Set() + 1 + 2 + 3 + 1 + 2 + 3
        assertEquals(3, set.size)
    }
}
```

The `testMultiAdd` method is a test. This test adds multiple items to a set and makes sure that each one is only added one time. The `assertEquals` method comes from the Java code of JUnit and checks that its two arguments are equal. If they are different, then the test fails. In this case, the test verifies that repeatedly adding the same numbers does not increase the size of a set.

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's all there is to it. The test can be run using any JUnit test runner. Here it is being run with the command-line test runner from Sun.

```
$ 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`. For this use case, writing the annotation in Scala does not seem helpful, so the standard compiler does not support it. The reasoning is that the Scala support would inevitably fall short of the full possibilities of Java annotations, and further, Scala might one day have its own reflection, in which case you would want to access Scala annotations with Scala reflection.

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 APIs. 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()
```

As an aside, notice that the methods are in class `Tests$` instead of class `Tests` when viewed with Java reflection. As described at the beginning of the chapter, the implementation of a Scala singleton object is placed in a Java class with a dollar sign added to the end of its name. In this case, the implementation of `Tests` is in the Java class `Tests$`.

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.

29.3 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 when your Scala code accesses 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 declared 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 an `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 still unknown, but now it is sure to be subtype of `Component`.

By the way, there is a shorter way to write these examples. If you write `Iterator[_]`, it means the same thing as `Iterator[T] forSome { type T }`. This is *placeholder syntax* for existential types, and is similar in spirit to the placeholder syntax for function literals that was described in [Section 8.5](#). If you use a `_` in place of an expression, then Scala treats this as a placeholder and makes a function literal for you. For types it works similarly. If you use a `_` in place of a type, Scala makes an existential type for you. Each `_` becomes one type parameter in a `forSome` clause, so if you use two `_`'s in the same type, you will get the effect of a `forSome` clause with two types in it.

You can also insert upper and lower bounds when using this placeholder syntax. Simply add them to the `_` instead of in the `forSome` clause. The type `Iterator[_ <: Component]` is the same as this one that you just saw:

```
Iterator[T] forSome { type T <: Component }
```

Enough about the existential types themselves. How do you actually use them? Well, 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 on line 3. There is no way to name the type of elements in the Java collection, so you cannot write down a satisfactory type for `set`. To work around this kind of problem, here 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 20](#) 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.

29.4 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 three aspects of Scala’s implementation on the Java platform: what the translation looks like, how Scala and Java annotations work together, and how Scala’s existential types let you access Java wildcard types. These topics are important whenever you use Scala and Java together.

Chapter 30

Actors and Concurrency

30.1 Overview

Sometimes it helps in designing a program to specify that things happen independently, in parallel, *concurrently*. Java includes support for concurrency, 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.

30.2 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 too 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 will not deadlock while trying to obtain them. 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 race conditions nor deadlocks, so you cannot play it safe by overdoing it in either direction.

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.

30.3 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!
```

For creating simple actors like these, there is also a utility method named `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.
```

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. You can create actors and they run independently. How do they work together, though? How do they communicate without using shared memory and locks?

Actors communicate by sending each other *messages*. You send a message by using the “!” method, like this:

```
scala> sillyActor ! "hi there"
```

Nothing happens in this case, because `sillyActor` is too busy acting to process its messages, and so the “hi there” message sits in its mailbox unread. Here is a new, more sociable, actor that waits for a message in its mailbox and prints out whatever it receives. It receives a message by calling `receive` with a pattern-match expression.

```
val echoActor = actor {
  while (true) {
    receive {
      case msg =>
        println("received message: " + msg)
    }
  }
}

scala> echoActor ! "hi there"
received message: hi there
```

```
scala> echoActor ! 15
scala> received message: 15
```

As these examples suggest, when an actor sends a message, it does not block, and when an actor receives a message, it is not interrupted. The sent message waits in the receiving actor's mailbox until the actor calls `receive`.

Even then, only messages matching the supplied pattern-match expression will be processed. As discussed in [Section 15.7](#), a pattern-match expression is a partial function, not a full function. The `receive` method will not pass just any message from the mail box to the pattern-match expression. It will choose the first message in the mail box matching the pattern-match expression. If there is no such message present already, the actor will block until a matching message arrives.

```
scala> val intActor = actor { react {
    |   case x: Int =>
    |     // I only want Ints
    |     println("Got an Int: "+x)
    |   } }
intActor: scala.actors.Actor =
scala.actors.Actor$$anon$1@34ba6b
scala> intActor ! "hello"
scala> intActor ! Math.Pi
scala> intActor ! 12
Got an Int: 12
```

30.4 Better performance through thread reuse

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, despite their light-sounding name, threads are not all that cheap in Java. Threads use enough memory that typical Java virtual machines can have millions of objects but only 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 after it finds and processes a message. Its result type is `Nothing`. It evaluates the message handler and that's it.

Because the `react` method does not need to return, the implementation does not need to preserve the call stack of the current thread. Instead, the library can reuse the current thread for the next actor that wakes up. This approach is so effective that if every actor in a program uses `react` instead of `receive`, then only a single thread is necessary in principle to host all of the program's actors (to be sure, if your computer has several processor cores, the Scala runtime will use enough threads to utilize all cores when it can). In practice, programs will need at least a few `receive`'s, but you should use `react` whenever possible so as to conserve threads.

Because `react` does not return, the message handler you pass it must now both process that message and arrange to do all of the actor's remaining work. A common way to do this is to have a top-level work method—such as `act()` itself—that the message handler calls when it finishes. Here is an example of using this approach.

```
object NameResolver extends Actor {  
    import java.net.{InetAddress, UnknownHostException}  
  
    def act() {  
        receive {  
            case (name: String, actor: Actor) =>  
                actor ! getip(name)  
                act()  
            case "EXIT" =>  
                println("Name resolver exiting.")  
                // quit  
            case msg =>  
                println("Unhandled message: " + msg)  
                act()  
        }  
    }  
  
    def getip(name: String): Option[InetAddress] = {  
        try {  
            Some(InetAddress.getByName(name))  
        } catch {  
            case e: UnknownHostException =>  
                None  
        }  
    }  
}
```

```

    } 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()
res0: scala.actors.Actor = NameResolver@90d6c5
scala> NameResolver ! ("www.scala-lang.org", self)
scala> self.receive { case x => x }
res2: Any = Some(www.scala-lang.org/128.178.154.102)
scala> NameResolver ! ("wwwwww.scala-lang.org", self)
scala> self.receive { case x => x }
res4: 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)
    }
  }
}

```

30.5 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 use `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.Actor._  
import scala.actors.Actor._  
  
scala> self ! "hello"  
  
scala> self.receive { case x => x }  
res6: Any = hello  
  
scala>
```

Notice that `receive` returns the value computed by the pattern-match expression passed to it. In this case, the expression returns the message itself, and so the received message ends up being printed out by the interpreter shell.

If you use this technique, it is better to use a variant of `receive` called `receiveWithin`. You can then specify a timeout in milliseconds. If you use `receive` in the interpreter shell, then the `receive` will block the shell until a message arrives. In the case of `self.receive`, this could mean waiting forever! Instead, use `receiveWithin` with a timeout of 0.

```
scala> self.receiveWithin(0) { case x => x }  
res7: Any = TIMEOUT  
  
scala>
```

30.6 Good actors style

At this point you have seen everything you need to write your own actors. Simply using these methods takes you only so far, however. The point of

them is that they support an actors *style* of concurrent programming. To the extent you can write in this style, your code will be easier to debug and will have fewer deadlocks and race conditions. Take a look at the main guidelines for programming in an actors style.

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, instead of calling `Thread.sleep` directly and blocking the current actor, you could create a helper actor that sleeps and then sends a message back when enough time has elapsed.

```
actor {  
    Thread.sleep(time)  
    mainActor ! "WAKEUP"  
}
```

This helper actor does indeed block, but since it will never receive a message, it is OK in this case. The main actor remains available to answer new requests. Figure 30.1 shows this idiom in use. The `emoteLater()` method uses the idiom described above. It creates a new actor that will do the `sleep` so that the main actor does not block. So that it sends the "Emote" message to the correct actor, it is careful to evaluate `self` in the scope of the main actor instead of the scope of the helper actor.

Because this actor does not block in `sleep`—its helper actor does—it can continue to do other work while waiting for its next time to emote. Unlike the earlier silly actor, this one will continue to print out messages while it waits for its next input.

```
scala> sillyActor2 ! "hi there"  
scala> Received: hi there  
I'm acting!
```

Figure 30.1: This actor uses a helper actor to avoid blocking itself.

```
val sillyActor2 = actor {
    def emoteLater() {
        val mainActor = self
        actor {
            Thread.sleep(1000)
            mainActor ! "Emote"
        }
    }

    var emoted = 0
    emoteLater()

    loop {
        receive {
            case "Emote" =>
                println("I'm acting!")
                emoted += 1
                if (emoted < 5)
                    emoteLater()
            case msg =>
                println("Received: " + msg)
        }
    }
}
```

I'm acting!

I'm acting!

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. The “immutable” collection types live up to their name, including both the immutable sets and maps and the `List` class. You can go a long way with immutable data structures in Scala.

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 18](#), 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 an-

other, 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. [Figure 30.2](#) shows an updated name-resolving actor that uses case classes instead of tuples for its messages.

Figure 30.2: An actor that uses case classes for messages.

```
import scala.actors.Actor._  
import java.net.InetAddress  
  
case class LookupIP(name: String, respondto: Actor)  
  
case class LookupResult(name: String,  
                        address: Option[InetAddress])  
  
def getip(name: String) = // as before  
  
val nameResolver2 = actor {  
    loop {  
        react {  
            case LookupIP(name, actor) =>  
                actor ! LookupResult(name, getip(name))  
            }  
        }  
    }  
}
```

30.7 A longer example: Parallel discrete event simulation

As a longer example, suppose you wanted to parallelize the discrete event simulation of [Chapter 18](#). Each participant in the simulation can run as its own actor, thus allowing you to speed up a simulation by using more processors.

This section walks through the design and implementation of a parallel circuit simulation. The example is based on a parallel circuit simulator developed for Scala by Philipp Haller.

Overall Design

To start with, let's make every simulated object be an actor, thus allowing a large amount of parallelism. To hold the common behavior of simulated objects, define a trait `Simulant`, because “simulated object” is a mouthful. Wires, gates, and any other simulated objects can then mix in this trait.

```
trait Simulant
class Wire extends Simulant
def orGate(o1: Wire, o2: Wire, output: Wire) {
    // creates a Simulant
}
```

`Simulant` is made a trait because it is intended to be mixed in to many diverse classes. It would also be possible to make it an abstract class, but then any class that inherits from it would not be able to subclass anything else. Making it a trait keeps your options open, so that you can make things that are primarily a dessert topping but also a simulant. To contrast, `Wire` is concrete, and it is not expected to be mixed into anything. You are unlikely to have a class that happens to be a wire but is primarily something else, so the framework does not support such usage. `Wire` is made a class.

A core design question is how the simulation participants are to stay synchronized. Participant A should not race ahead and process an event at time tick 100 until all other actors have finished with time tick 99. To see why this is essential, imagine for a moment that simulant A is working at time 90 while simulant B is working at time 100. It might be that participant A is about to send a message that changes B's state at time 91. B will not learn this until too late, because it has already processed times 92-99. To avoid this problem, no simulant should process events for time n until all other simulants are finished with time $n - 1$.

There are multiple approaches to keeping all of the simulants working on one time tick at a time. To keep things simple, suppose that there is a single clock actor that knows the current time and tells actors when it is safe to move forward. To keep the clock from moving forward before all simulants are ready, the clock can ping actors at carefully chosen times to make sure they have received and processed all messages for the current time tick. Thus there will need to be Ping messages from the clock to the simulants, and Pong messages for their replies.

```
case class Ping(time: Int)
case class Pong(time: Int, from: Actor)
```

These messages could be defined as having no fields. However, the `time` and `from` fields add a little bit of redundancy to the system. The `time` field holds the time of a ping, and it can be used to connect a Pong with its associated Ping. The `from` field is the sender of a Pong. The sender of a Ping is

always the clock, so it does not have a `from` field. All of this information is unnecessary if the program is behaving perfectly, but it can simplify the logic in some places, and it can greatly help in debugging if the program has any errors.

One subtlety that arises is how a simulant knows it has finished with the current time tick. Simulants should not respond to a Ping until they have finished all the work for that tick, but how do they know? If another actor has sent them a message, maybe the message has not yet arrived.

To keep things simple, let's say that simulants do not send each other messages directly, and that they do not post work items for the current time tick. Instead, whenever a simulant needs something from another simulant, it posts a work item on the simulation's agenda, and that work item must be scheduled for a time strictly in the future.

If the agenda is held by the clock, then the synchronization becomes particularly simple. Whenever an actor receives a ping, it must have received all the other messages the clock is going to send it for that tick. If an actor has processed all of its messages except for a Ping, then it is safe to respond to the Ping. No more work will be arriving during the current time tick. Taking this approach, a Clock has the following state:

```
class Clock extends Actor {  
    private var running = false  
    private var currentTime = 0  
    private var agenda: List[WorkItem] = Nil  
}
```

Other arrangements are possible. In fact, it's a good exercise for the reader to extend the framework so that actors can send messages to each other. If you try it, you will need a way for simulants to synchronize with those simulants they have made requests to. Each simulant should delay responding to a Ping until any other simulants it has made requests to are finished processing those requests. To ensure this property, you will need the simulants to pass each other some extra information. For now, just suppose that all simulants communicate only via the clock.

Another subtlety of the design has to do with how you set up a simulation to begin with. It is natural enough to create the simulation with the clock stopped, add all the simulants and connect them together, and then start the clock. The subtlety is that you need to be absolutely sure that everything is

connected before you start the clock running. Otherwise, some parts of the simulation will start running before they are fully formed.

How do you know when the simulation is fully assembled and ready to start? There are again multiple ways to approach this problem. One simple way is to avoid actors messages while setting the simulation up. That way, once the last method call returns, you know that the simulation is entirely constructed. The resulting coding pattern is that you prefer regular method calls to set the simulation up, but actors messages while the simulation is running.

Given the preceding decisions, the rest of the design is less tricky. A `WorkItem` can be defined much like in [Chapter 18](#), in that it holds a time and an action. For the parallel simulation, however, the action itself has a different encoding. In [Chapter 18](#), actions are represented as zero-argument functions. For parallel simulation, it is more natural to use a target actor and a message to be sent to that actor.

```
case class WorkItem(time: Int, msg: Any, target: Actor)
```

Likewise, the `afterDelay` method for scheduling a new work item becomes an `AfterDelay` message that can be sent to the clock. Just as with the `WorkItem` class, the zero-argument action function is replaced by a message and a target actor.

```
case class AfterDelay(delay: Int, msg: Any, target: Actor)
```

Finally, it will prove useful to have messages requesting the simulation to start and stop.

```
case object Start  
case object Stop
```

That's it for the overall design. There is a `Clock` class holding a current time and an agenda, and a clock only advances the clock after it has pinged all of its simulants to be sure they are ready. There is a `Simulant` trait for simulation participants, and these communicate with their fellow simulants by sending work items to the clock to add to its agenda.

Take a look now at how to implement these core classes.

Implementing the simulation framework

There are two things that need implementing for the core framework: the Clock class and the Simulant trait. Consider the Clock class, first. The necessary state of a clock is as follows:

```
class Clock extends Actor {  
    private var running = false  
    private var currentTime = 0  
    private var agenda: List[WorkItem] = Nil  
    private var allSimulants: List[Actor] = Nil  
    private var busySimulants: Set[Actor] = Set.empty
```

A clock starts out with running set to false. Once the simulation is fully initialized, the clock will be sent the Start message and running will become true. This way, the simulation stays frozen until all of its pieces have been connected together as desired. It also means that, since all of the simulants are also frozen, it is safe to use regular method calls to set things up instead of needing to use actors messages.

A clock may as well go ahead and start running as an actor once it is created. This is safe, because it will not actually do anything until it receives a Start message.

```
start()
```

A clock also keeps track of the current time (currentTime), the list of participants managed by this clock (allSimulants), and the list of participants that are still working on the current time tick (busySimulants). A list is used to hold allSimulants, because it is only iterated through, but a set is used for busySimulants because items will be removed from it in an unpredictable order. Once the simulator starts running, it will only advance to a new time when busySimulants is empty, and whenever it advances the clock, it will set busySimulants to allSimulants.

To set up a simulation, there is going to be a need for a method to add new simulants to a clock. It may as well be added right now:

```
def add(sim: Simulant) {  
    allSimulants = sim :: allSimulants  
}
```

That's the state of a clock. Now look at its activity. Its main loop alternates between two responsibilities: advancing the clock, and responding to messages. Once the clock advances, it can only advance again when at least one message has been received, so it is safe to define the main loop as an alternation between these two activities.

```
def act() {
    loop {
        if (running && busySimulants.isEmpty)
            advance()

        reactToOneMessage()
    }
}
```

The advancement of time has a few parts beyond simply incrementing `currentTime`. First, if the agenda is empty, and the simulation is not just starting, then the simulation should exit. Second, assuming the agenda is non-empty, all work items for time `currentTime` should now take place. Third, all simulants should be put on the `busySimulants` list and sent Pings. The clock will not advance again until all Pings have been responded to.

```
def advance() {
    if (agenda.isEmpty && currentTime > 0) {
        println("** Agenda empty. Clock exiting at time "+
               currentTime+".")
        self ! Stop
        return
    }

    currentTime += 1
    println("Advancing to time "+currentTime)

    processCurrentEvents()
    for (sim <- allSimulants)
        sim ! Ping(currentTime)

    busySimulants = Set.empty ++ allSimulants
}
```

Processing the current events is simply a matter of processing all events at the top of the agenda whose time is `currentTime`.

```
private def processCurrentEvents() {  
    val todoNow = agenda.takeWhile(_.time <= currentTime)  
    agenda = agenda.drop(todoNow.length)  
    for (WorkItem(time, msg, target) <- todoNow) {  
        assert (time == currentTime)  
        target ! msg  
    }  
}
```

There are three steps in this method. First, the items that need to occur at the current time are selected using `takeWhile` and saved into the `val todoNow`. Second, those items are dropped from the agenda by using `drop`. Finally, the items to do now are looped through and sent the target message. The `assert` is included just to guarantee that the scheduler's logic is sound.

Given this ground work, handling the messages that a clock can receive is straightforward. An `AfterDelay` message causes a new item to be added to the work queue. A `Pong` causes a simulant to be removed from the list of busy simulants. `Start` causes the simulation to begin, and `Stop` causes the clock to stop.

```
def reactToOneMessage() {  
    react {  
        case AfterDelay(delay, msg, target) =>  
            val item = WorkItem(currentTime + delay, msg, target)  
            agenda = insert(agenda, item)  
  
        case Pong(time, sim) =>  
            assert (time == currentTime)  
            assert (busySimulants contains sim)  
            busySimulants -= sim  
  
        case Start =>  
            running = true  
  
        case Stop =>  
            for (sim <- allSimulants)  
                sim ! Stop  
            exit()  
    }  
}
```

The `insert` method, not shown, is exactly like that of [Chapter 18](#). It inserts its argument into the agenda while being careful to keep the agenda sorted.

That's the complete implementation of `Clock`. Now consider how to implement `Simulant`. Boiled down to its essence, a `Simulant` is any actor that understands and cooperates with the simulation messages `Stop` and `Ping`. Its `act` method can therefore be as simple as this:

```
def act() {
    loop {
        react {
            case Stop => exit()
            case Ping(time) =>
                if (time == 1) simStarting()
                clock ! Pong(time, self)
            case msg => handleSimMessage(msg)
        }
    }
}
```

Whenever a simulant receives the `Stop` message, it exits. If it receives a `Ping`, it responds with a `Pong`. If the `Ping` is for time 1, then `simStarting()` is called before the `Pong` is sent back, thus allowing a subclass to define behavior that should happen when the simulation starts running. Any other message must be interpreted by a subclass, so it defers to an abstract `handleSimMessage` method.

There are two abstract members of a simulant. First is the `handleSimMessage` just described. Second, a simulant must know its clock so that it can reply to `Ping` messages and schedule new work items. Putting it all together, the `Simulant` trait is as follows.

```
trait Simulant extends Actor {
    val clock: Clock
    def handleSimMessage(msg: Any)
    def simStarting() { }
    def act() {
        loop {
            react {
                case Stop => exit()
                case Ping(time) =>
```

```
    if (time == 1) simStarting()
        clock ! Pong(time, self)
    case msg => handleSimMessage(msg)
}
}
}
start()
}
```

Note that a simulant goes ahead and starts running the moment it is created. This is safe and convenient, because it will not actually do anything until its clock sends it a message, and that should not happen until the simulation starts and the clock receives a `Start` message.

That completes the framework for parallel event simulation. Like its sequential cousin in [Chapter 18](#), it takes surprisingly little code for how much power it gives you.

Implementing a circuit simulation

Now that the simulation framework is complete, it's time to work on the implementation of circuits. A circuit has a number of wires and gates, which will be simulants, and a clock for managing the simulation. A wire holds a boolean signal—either high (`true`) or low (`false`). Gates are connected to a number of wires, some of which are inputs and some of which are outputs. Gates compute a signal for their output wires based on the state of their input wires.

Since the wire, gates, *etc.*, of a circuit are only used for that particular circuit, their classes can be defined as members of a `Circuit` class, just as with the currency objects of [Section 20.4](#). The overall `Circuit` class will therefore have a number of members:

```
class Circuit {
    val clock = new Clock
    // simulation messages
    // delay constants
    // Wire and Gate classes and methods
    // misc. utility methods
}
```

Let's look at each of these members, one group at a time. First, there are the simulation messages. Once the simulation starts running, wires and gates can only communicate via message sends, so they will need a message type for each kind of information they want to send each other. There are only two such kinds of information. Gates need to tell their output wires to change state, and wires need to inform the gates they are inputs to whenever their state changes.

```
case class SetSignal(sig: Boolean)
case class SignalChanged(wire: Wire, sig: Boolean)
```

Next, there are several delays that must be chosen. Any work item scheduled with the simulation framework—including propagation of a signal to or from a wire—must be scheduled at some time in the future. It is unclear what the precise delays should be, so those delays are worth putting into vals. This way, they can be easily adjusted in the future.

```
val WireDelay = 1
val InverterDelay = 2
val OrGateDelay = 3
val AndGateDelay = 3
```

At this point it is time to look at the `Wire` and `Gate` classes. Consider wires, first. A wire is a simulant that has a current signal state (high or low) and a list of gates that are observing that state. It mixes in the `Simulant` trait, so it also needs to specify a clock to use.

```
class Wire(name: String, init: Boolean) extends Simulant {
    def this(name: String) { this(name, false) }
    def this() { this("unnamed") }

    val clock = Circuit.this.clock
    clock.add(this)

    private var sigVal = init
    private var observers: List[Actor] = List()
```

The class also needs a `handleSimMessage` method to specify how it should respond to simulation messages. The only message a wire should receive is `SetSignal`, the message for changing a wire's signal. The response should be that if the signal is different from the current signal, the current state changes, and the new signal is propagated.

```

def handleSimMessage(msg: Any) {
    msg match {
        case SetSignal(s) =>
            if (s != sigVal) {
                sigVal = s
                signal0bservers()
            }
    }
}

def signal0bservers() {
    for (obs <- observers)
        clock ! AfterDelay(
            WireDelay,
            SignalChanged(this, sigVal),
            obs)
}

```

The above code shows how *changes* in a wire's signal are propagated to any gates watching it. It's also important to pass the initial state of a wire to any observing gates. This only needs to be done once, when the simulation starts up. After that, gates can simply store the result of the most recent `SignalChanged` they have received. Sending out the initial signal when the simulation starts is as simple as providing a `simStarting()` method:

```
override def simStarting() { signal0bservers() }
```

There are now just a few more odds and ends about wires. They need a method for connecting new gates, and they could use a nice `toString` method.

```

def add0bserver(obs: Actor) {
    observers = obs :: observers
}
override def toString = "Wire(\"+name+\""

```

That is everything you need for wires. Now consider gates, the other major class of objects in a circuit. There are three fundamental gates it would be nice to define: And, Or, and Not. All of these share a lot of behavior, so it is worth defining an abstract `Gate` class to hold the commonality.

A difficulty in defining this Gate class is that some gates have two inputs wires (And, Or) while others have just one (Not). It would be possible to model this difference explicitly. However, it simplifies the code to think of all gates as having two inputs, where Not gates simply ignore their second input. The ignored second input can be set to some dummy wire that never changes state from false.

```
private object DummyWire extends Wire("dummy")
```

Given this trick, the gate class wills together straightforwardly. It mixes in the Simulant trait, and its one constructor accepts two input wires and one output wire.

```
abstract class Gate(in1: Wire, in2: Wire, out: Wire)  
  extends Simulant {
```

There are two abstract members of a gate that specific subclasses have to fill in. The most obvious is that different kinds of gates compute a different function of their inputs. Thus, there should be an abstract method for computing inputs:

```
def computeOutput(s1: Boolean, s2: Boolean): Boolean
```

Second, different kinds of gates have different propagation delays. Thus, the delay of the gate should be an abstract val.

```
val delay: Int
```

The delay could be a def, but making it a val encodes the fact that a particular gate's delay should never change.

Because Gate mixes in Simulant, it is required to specify which clock it is using. As with Wire, Gate should specify the clock of the enclosing Circuit. For convenience, the Gate can go ahead and add itself to the clock when it is constructed.

```
val clock = Circuit.this.clock  
clock.add(this)
```

Similarly, it makes sense to go ahead and connect the gate to the two input wires, using regular method calls.

```
in1.addObserver(this)
in2.addObserver(this)
```

The only local state of a gate is the most recent signal on each of its input wires. This state needs to be stored, because wires only send a signal when the state changes. If one input wire changes, only that one wire's state will be sent to the gate, but the new output will need to be computed from both wires' states.

```
var s1, s2 = false
```

Now look at how gates respond to simulation messages. There is only one message they need to handle, and that's the `SignalChanged` message indicating that one of the input wires has changed. When a `SignalChanged` arrives, two things need to be done. First, the local notion of the wire states need to be updated according to the change. Second, the new output needs to be computed and then sent out to the output wire with a `SetSignal` message.

```
def handleSimMessage(msg: Any) {
    msg match {
        case SignalChanged(w, sig) =>
            if (w == in1)
                s1 = sig
            if (w == in2)
                s2 = sig
            clock ! AfterDelay(delay,
                SetSignal(computeOutput(s1, s2)),
                out)
    }
}
```

That is all of the abstract `Gate` class. Given that class, it is now easy to define specific kinds of gates. As with the sequential simulation in [Chapter 18](#), the gates can be created as side effects of calling some utility methods. All the methods need to do is create a `Gate` and fill in the appropriate delay and output computation. Everything else is common to all gates and is handled in the `Gate` class.

```
def orGate(in1: Wire, in2: Wire, output: Wire) =
    new Gate(in1, in2, output) {
```

```

    val delay = OrGateDelay
    def computeOutput(s1: Boolean, s2: Boolean) = s1 || s2
}

def andGate(in1: Wire, in2: Wire, output: Wire) =
new Gate(in1, in2, output) {
    val delay = AndGateDelay
    def computeOutput(s1: Boolean, s2: Boolean) = s1 && s2
}

```

In the case of Not gates, a dummy wire will be specified as the second input. This is an implementation detail from the point of view of a caller creating a Not gate, so the inverter method only takes one input wire instead of two.

```

def inverter(input: Wire, output: Wire) =
new Gate(input, DummyWire, output) {
    val delay = InverterDelay
    def computeOutput(s1: Boolean, ignored: Boolean) = !s1
}

```

One final utility is worth defining. When you create a circuit simulation, you would like to watch it evolve, but you would not want to have to see every change to any wires in the entire circuit. The wire-probing approach of [Chapter 18](#) is a way to log just the few wires that are of interest.

Define a probe method that takes a `Wire` as an argument and then prints out line of text whenever that wire's signal changes. The method can be implemented by simply making a new simulant that connects itself to a specified wire. This simulant can respond to `SignalChanged` messages by printing out the new signal.

```

def probe(wire: Wire) = new Simulant {
    val clock = Circuit.this.clock
    clock.add(this)
    wire.addObserver(this)
    def handleSimMessage(msg: Any) {
        msg match {
            case SignalChanged(w, s) =>
                println("signal "+w+" changed to "+s)
        }
    }
}

```

```
}
```

That is the bulk of the `Circuit` class. Callers should create an instance of `Circuit`, create a bunch of wires and gates, call `probe` on a few wires of interest, and then start the simulation running. The one piece missing is how the simulation is started, and that can be as simple as sending the clock a `Start` message.

```
def start() { clock ! Start }
```

Libraries of circuit components

The code now gives everything you absolutely need to assemble a circuit and start it in motion. It provides And gates, Or gates, and Not gates. That is one more gate than is needed to build any circuit, so what more could a user want?

Well, to make things nicer on users, you could also supply a library of standard components. For example, the half adders and full adders of [Chapter ??](#) might be nice to predefine:

```
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)
}

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)
}
```

Those are the methods, but where should they be put? It would be nice to keep them separate from the core `Circuit` class, because they are really at a different level of abstraction. The standard components are written entirely in terms of the simple components already present in `Circuit`. Keeping the

levels of abstraction separated into separate modules makes for nicer code to understand and work with.

One way is to wrap the methods in a class that takes a circuit as a constructor parameter. That way works, but instead let's wrap them in a trait that is intended to be mixed into class `Circuit`. The trait itself looks like this:

```
trait Adders extends Circuit {  
    def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) {  
        // ...  
    }  
  
    def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire) {  
        // ...  
    }  
}
```

The trait is marked as extending `Circuit`, so it can directly access members of `Circuit` such as `Wire` and `orGate`. Using the trait looks like this:

```
val circuit = new Circuit with Adders
```

This `circuit` variable holds a circuit that has all of the methods of `Circuit` and all of the methods of `Adders`. Note that with this coding pattern, based on a trait instead of a class, you set the stage to provide multiple component sets. Users mix in whichever component sets they plan to use, like this:

```
val circuit =  
    new Circuit  
        with Adders  
        with Multiplexers  
        with FlipFlops  
        with MultiCoreProcessors
```

Testing it all out

That's the whole framework. It includes a simulation framework, a circuit simulation class, and a small library of standard adder components. Here is a simple test object that uses it:

```
object Test {  
    def main(args: Array[String]) {  
        val circuit = new Circuit with Adders  
        import circuit._  
  
        val ain = new Wire("ain", true)  
        val bin = new Wire("bin", false)  
        val cin = new Wire("cin", true)  
        val sout = new Wire("sout")  
        val cout = new Wire("cout")  
  
        probe(ain)  
        probe(bin)  
        probe(cin)  
        probe(sout)  
        probe(cout)  
  
        fullAdder(ain, bin, cin, sout, cout)  
        circuit.start()  
    }  
}
```

This example creates a circuit that includes the `Adders` trait. It immediately imports all of the circuit's members, thus allowing easy accesses to methods like `probe` and `fullAdder`. Without the import, it would be necessary to write `circuit.probe(ain)` instead of just `probe(ain)`.

The example then creates five wires. Three will be used as inputs (`ain`, `bin`, and `cin`), and two will be used as outputs (`sout`, `cout`). The three input wires are given arbitrary initial signals of `true`, `false`, and `true`. These inputs correspond to adding 1 to 0, with a carry in of 1.

The `probe` method gets applied to all five externally visible wires, so any changes in their state can be observed as the simulation runs. Finally the wires are plugged into a full adder, and the simulation is started. The output of the simulation is as follows:

```
Advancing to time 1  
Advancing to time 2  
signal Wire(cout) changed to false  
signal Wire(cin) changed to true  
signal Wire(ain) changed to true
```

```
signal Wire(sout) changed to false
signal Wire(bin) changed to false
Advancing to time 3
Advancing to time 4
Advancing to time 5
Advancing to time 6
Advancing to time 7
Advancing to time 8
Advancing to time 9
Advancing to time 10
signal Wire(cout) changed to true
Advancing to time 11
Advancing to time 12
Advancing to time 13
Advancing to time 14
Advancing to time 15
Advancing to time 16
Advancing to time 17
Advancing to time 18
signal Wire(sout) changed to true
Advancing to time 19
Advancing to time 20
Advancing to time 21
signal Wire(sout) changed to false
** Agenda empty. Clock exiting at time 21.
```

Does the final output look correct? The inputs were true, false and true, which correspond to the numbers 1, 0, and 1. The sum of these is 10 in binary, so the carry output (cout) should be 1, and the sum output (sout) should be 0. Scanning the output, we can see that this is the case.

Do the timings look correct? The probes are part of the simulation, so they should print their message one clock tick after the wire actually changes. Thus the print outs of the initial wire states are at time 2 rather than time 1. The final value of cout is computed at time 9, while that of sout is not computed until time 20.

30.8 Conclusion

Concurrent programming gives you great power. It lets you simplify your code, and it lets you take advantage of multiple processors. It is therefore unfortunate that the most widely used concurrency primitives, threads, locks, and monitors, are such a minefield of deadlocks and race conditions.

The actors style provides a way out of the minefield, letting you write concurrent programs without having such a great risk of deadlocks and race conditions. This chapter has introduced several fundamental constructs for working with actors in Scala, including how to create actors, how to send and receive messages, and how to conserve threads with `react`, among other nuts and bolts. It then showed you how to use these constructs as part of a general actors style.

Chapter 31

Combinator Parsing

There are many situations where someone has defined a small, special-purpose language and now you need to process it. For example, you need to read configuration files for your software, and you want to make them easier to modify by hand than XML. Alternatively, maybe you want to support an input language in your program, such as search terms with boolean operators (computer, find me a movie “with ‘space ships’ and without ‘love stories’”). Whatever the reason, you are going to need a *parser*. You need a way to convert the input language into some data structure your software can 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.

An alternative choice is to use a parser generator. There exist quite a few of these generators. Some of the better known are Yacc and 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 standalone 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 opera-

tors 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 31.6](#). 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. If you don't, the material in this chapter can also safely be skipped.

31.1 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   ::=  floatingPointNumber | '(' 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._  
class Arith extends JavaTokenParsers {  
    def expr: Parser[Any] = term ~ rep(("+" ~ term) | ("-" ~ term))  
    def term: Parser[Any] = factor ~ rep(("*" ~ factor) | ("/" ~ factor))  
    def factor: Parser[Any] = floatingPointNumber | "(" ~ expr ~ ")"  
}
```

The parsers for arithmetic expressions are contained in a class that inherits from the class `JavaTokenParsers`. This class provides the basic machinery for writing a parser and also provides some primitive parsers that recognize some word classes: identifiers, string literals and numbers. In the example above you need only the primitive `floatingPointNumber` parser, which is inherited from this class.

The three definitions in class `Arith` 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. In the example above we have chosen not to write any spaces around the “`~`” operator. That way, the parser code keeps closely to the visual appearance of the grammar—it just replaces spaces by `~`-characters.

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 all there is to it. The resulting class `Arith` defines three parsers `expr`, `term` and `factor` that can be used to parse expressions and their parts.

31.2 Running your parser

You can test your parsers with the following a small program:

```
object ArithTest extends Arith {
  def main(args: Array[String]) {
    println("input : "+args(0))
    println(parseAll(expr, args(0)))
  }
}
```

The `ArithTest` object defines a `main` method which parses the first command-line argument that's passed to it. It prints the original input argument, and then prints its parsed version. Parsing is done by the expression

```
parseAll(expr, input)
```

This expression applies the parser `expr` to the given `input`. It expects that all of the `input` matches, i.e. that there are no characters trailing a parsed expression. There's also a method `parse` which allows to parse an input prefix, leaving some remainder unread.

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: `-' 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 `parseAll` method then issued an error message which said that it expected a ‘-’ operator at the point of the closing parenthesis. You’ll find out below why it produced this particular error message, and how you can improve it.

31.3 Basic Regular Expression Parsers

The parser for arithmetic expressions made use of another parser named `floatingPointNumber` which was inherited from `JavaTokenParsers`. This parser recognizes a floating point number in the format of Java. But what do you do if you need to parse numbers in a format that’s a bit different from Java’s? Here, regular expression parsers come in.

The idea is that you can use any regular expression as a parser. The regular expression parses all strings that it can match. Its result is the parsed string. For instance, the following parser describes Java’s identifiers:

```
object MyParsers extends RegexParsers {  
    val ident: Parser[String] = """[a-zA-Z_]\w*""".r  
}
```

The `MyParsers` object inherits from class `RegexParsers` whereas `Arith` inherited from `JavaTokenParsers`. Scala’s parsing combinators are arranged in a hierarchy of classes, which are all contained in package `scala.util.parsing.combinator`. The top-level class is `Parsers`, which defines a very general parsing framework for all sorts of input. One level

```

value = obj | arr |
    stringLiteral | floatingPointNumber |
    "null" | "true" | "false"
obj = "{" [members] "}"
arr = "[" [values] "]"
members = member {"," member}
member = stringLit ":" value
values = value {"," value}

```

Figure 31.1: JSON grammar

```

{ "address book": {
    "name": "John Smith",
    "address": {
        "street": "10 Market Street",
        "city" : "San Francisco, CA",
        "zip" : 94111
    },
    "phone numbers": [
        "408 338-4238",
        "408 111-6892"
    ]
}
}

```

Figure 31.2: Data in JSON format

below is class `RegexParsers` which requires that the input is a sequence of characters and provides for regular expression parsing as shown above. Even more specialized is class `@JavaTokenParsers@`, which implements parsers for basic classes of words (or: *tokens*) as they are defined in Java.

```
class JSON extends JavaTokenParsers {  
    def value : Parser[Any] = obj | arr |  
        stringLiteral |  
        floatingPointNumber |  
        "null" | "true" | "false"  
    def obj   : Parser[Any] = "{$~repsep(member, ",")~$}"  
    def arr   : Parser[Any] = "[{$~repsep(value, ",")~$}]"  
    def member: Parser[Any] = stringLiteral~":"~value  
}
```

Figure 31.3: A simple JSON parser

31.4 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. The syntax of JSON is shown in [Figure 31.1](#). 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.

As an example, [Figure 31.2](#) contains an address-book formatted as a JSON object. Parsing such data is straightforward when using Scala's parser combinators. The complete parser is shown in [Figure 31.3](#).

This parser follows the same structure as the arithmetic expression parser. It is again a straightforward mapping 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 java.io.FileReader

object JSONTest extends JSON {
    def main(args: Array[String]) {
        val reader = new FileReader(args(0))
        println(parseAll(value, reader))
    }
}
```

The `main` method in this program first creates a `FileReader` object. It then parses the characters returned by that reader according to the value production of the JSON grammar. Note that `parseAll` and `parse` exist in overloaded variants: both can take a character sequence or alternatively an input reader as second argument.

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:

```
java JSONTest address-book1.json
[13.4] 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"))~]))))~}})))~})
```

31.5 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 output is not very useful. It is 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. Regular expression parsers such as `"""\d+""".r` also return the parsed string itself. The same holds for regular expression parsers such as `stringLiteral` or `floatingPointNumber` which are inherited from class `JavaTokenParsers`.
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 `“~”`. So if `P` returns `“true”` and `Q` returns `“?”`, then the sequential composition `P~Q` returns `~(“true”, “?”)`, which prints as `(true~?)`.
4. An alternative composition `P | Q` returns the result of either `P` or `Q`, whichever one succeeds.
5. A repetition `rep(P)` or `repsep(P, separator)` returns a list of the results of all runs of `P`.
6. An option `opt(P)` returns an instance of Scala’s `Option` type. It returns `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 one more combination form for parsers: `“^”`.

The “`^~`” operator *transforms* the result of a parser. Expressions using this operator have the form `P ^~ f` where `P` is a parser and `f` is a function. `P ^~ f` parses the same sentences as just `P`. Whenever `P` returns with some result `R`, the result of `P ^~ f` is `f(R)`.

As an example, here is a parser that parses a numeric literal and converts it to a Scala integer:

```
numericLit ^~ (_.toInt)
```

And here is a parser that parses the string "true" and returns Scala's `true` value:

```
"true" ^~ (x => 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, ",") ~ "}" ^~
    { case "{" ~ ms ~ "}" => Map() ++ ms }
```

Remember that the “`~`” operator produces as result an instance of a case class with the same name, “`~`”. Here's a definition of that class—it's an inner class of class `Parsers`:

```
case class ~[+A, +B](x: A, y: B) {
  override def toString = "(" + x + " ~ " + y + ")"
}
```

The name of the class is intentionally the same as the name of the sequence combinator method “`~`”. That way, you can match parser results with patterns that follow the same structure as the parsers themselves. For instance, the pattern `"{" ~ ms ~ "}"` 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 “`^~`”. In its desugared versions where the “`~`” operator comes first, the same pattern reads `~(~("{", 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 that avoids producing unnecessary parser results that are immediately

```

class JSON1 extends JavaTokenParsers {
    def obj: Parser[Map[String, Any]] =
        "{~":" repsep(member, ",") <~":" }" ^^ (Map() ++ _)
    def arr: Parser[List[Any]] =
        "["~":" repsep(value, ",") <~":" ]"
    def member: Parser[(String, Any)] =
        stringLiteral~":" value ^^
            { case name~":" value => (name, value) }
    def value: Parser[Any] = (
        obj
        | arr
        | stringLiteral
        | floatingPointNumber ^^ (_.toInt)
        | "null" ^^ (x => null)
        | "true" ^^ (x => true)
        | "false" ^^ (x => false)
    )
}

```

Figure 31.4: A full JSON parser that returns meaningful results

discarded by the pattern match. The alternative makes use of the “`~>`” and “`<~`” parser combinators. Both express sequential composition just like “`~`”, but “`~>`” keeps only the result of its right operand, whereas “`<~`” keeps only the result of its left operand. Using these combinators, the JSON object parser can be expressed more succinctly:

```

def obj: Parser[Map[String, Any]] =
    "{~":" repsep(member, ",") <~":" }" ^^ (Map() ++ _)

```

Figure 31.4 shows a full JSON parser that returns meaningful results. 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(

```

Table 31.1: Summary of parser combinators

<code>" ... "</code>	literal
<code>" ... ".r</code>	regular expression
<code>P~Q</code>	sequential composition
<code>P <~ Q, P ~> Q</code>	sequential composition; keep left/right only
<code>P Q</code>	alternative
<code>opt(P)</code>	option
<code>rep(P)</code>	repetition
<code>repsep(P, Q)</code>	interleaved repetition
<code>P ^^ f</code>	result conversion

```
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)  
)  
)
```

Aside: Note that the body of each of the value parser in Figure 31.4 is enclosed in parentheses. This is a little trick to disable semicolon inference in parser expressions. You have seen in Section 4.2 that Scala assumes there's a semicolon 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 each alternative instead of at the beginning of the following one, like this:

```
def value: Parser[Any] =  
  obj |  
  arr |  
  stringLiteral |  
  ...
```

In that case, no parentheses around the body of function value 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:

```
obj; // semicolon implicitly inserted  
| arr
```

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.

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 31.1](#) lists the parser combinators that were discussed so far.

Symbolic vs alphanumeric names

Many of the parser combinators in [Table 31.1](#) use symbolic names. This has both advantages and disadvantages. On the minus side, symbolic names take time to learn. Users who are unfamiliar with Scala’s combinator parsing libraries are probably mystified what “~” or “~>” or “^” mean. On the plus side, symbolic names are short, and can be chosen to have the “right” precedences and associativities. For instance, the parser combinators “~”, “^”, and “|” are chosen intentionally in decreasing order of precedence. A typical grammar production is composed of alternatives which have a parsing part and a transformation part. The parsing part usually contains several sequential items which are separated by “~” operators. With the chosen precedences of “~”, “^”, and “|” you can write such a grammar production without any parentheses.

Furthermore, symbolic operators take less visual estate than alphabetic ones. That’s important for a parser because it lets you concentrate on the grammar at hand, instead of the combinators themselves. To see the difference, imagine for a moment that sequential composition was called `andThen` and alternative was called `orElse`. The arithmetic expression parsers in [Section 31.1](#) would look as follows:

```
class ArithHypothetical extends JavaTokenParsers {
```

```
def expr: Parser[Any]    =
  term andThen rep(("+" andThen term) orElse
                    ("-" andThen term))
def term: Parser[Any]    =
  factor andThen rep(("*" andThen factor) orElse
                    ("/" andthen factor))
def factor: Parser[Any] =
  floatingPointNumber orElse
  ("(" andThen expr andThen ")")
}
```

You notice that the code becomes much longer, and that it's hard to "see" the grammar among all those operators and parentheses. On the other hand, somebody new to combinator parsing could probably figure out better what the code is supposed to do.

As guidelines to choose between symbolic and alphabetic names we recommend the following:

- Use symbolic names in cases where they already have a universally established meaning. For instance, nobody would recommend to write `add` instead of `+` for numeric addition.
- Otherwise, give preference to alphabetic names if you want your code to be understandable to casual readers.
- You can still choose symbolic names for domain-specific libraries, if this gives clear advantages in legibility and you do not expect anyway that a casual reader without a firm grounding in the domain would be able understand the code immediately.

In the case of parser combinators we are looking at a highly domain-specific language, which casual readers would have trouble to understand. Furthermore, symbolic names give clear advantages in legibility for the expert. So we believe their use is warranted in this application.

31.6 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 to do 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 the 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 subsections 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
}
```

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

Sometimes, a parser reads a stream of tokens instead of a raw sequence of characters. A separate lexical analyzer is then used to convert a stream of raw characters into a stream of tokens. The type of parser inputs is defined as follows:

```
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, `RegexParsers` and `JavaTokenParsers` fix `Elem` to be equal to `Char`. But it would also be possible to set `Elem` to some other type, such as the type of tokens returned from a separate lexer.

Parser results

A parser might either succeed or fail on some given input. Consequently class `ParseResult` has two subclasses for representing success and failure:

```
sealed 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`. The type parameter 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 `AnyRefs`.

The Parser class

The previous characterization of parsers as functions from inputs to parse result was a bit oversimplified. The examples above have shown that parsers also implement *methods* such as “ \sim ” 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.

You saw syntax like this in [Section 27.3](#), where it was used to give a `self type` to a trait. Aliasing can also 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 568](#) and [569](#).

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 $\text{elem}(\text{kind}, p)$ 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 the result of p (*i.e.* x) and the result of q (*i.e.* 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 result type of “ \sim ” is a parser that returns an instance of the case class “ \sim ” with elements of types T and U . The type expression $T \sim U$ is just a more legible shorthand for the parameterized type $\sim[T, U]$. Generally, Scala always interprets a binary type operation such as $A \text{ op } B$, as the parameterized type $\text{op}[A, B]$. This is analogous to the situation for patterns, where an binary pattern $P \text{ op } Q$ is also interpreted as an application, *i.e.* $\text{op}(P, Q)$.

The other two sequential composition operators “ $<\sim$ ” and “ $\sim>$ ” could be defined just like “ \sim ”, only with some small adjustment in how the result is computed. A more elegant technique, though, 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 “ $|$ ” 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)
  }
}
```

Note that if P and Q both fail, then the failure message is determined by Q . This subtle choice is discussed later, in [Section 31.9](#).

Dealing with recursion

Note that the `q` parameter in methods “`~`” and “`|`” is by-name—its type is preceded by “`=>`”. This means that the actual parser argument will be evaluated only when `q` is needed, which should only be the case 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 “`|`” and “`~`” took 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 method of class `Parser` converts a parser’s result. The parser `P ^^ f` succeeds exactly when `P` succeeds. In that case it returns `P`’s result converted using the function `f`. Here is the implementation of this method:

```
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
    }
}
} // 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
```

31.7 String literals and regular expressions

The parsers you have written so far made use of string literals and regular expressions to parse single words. The support for these comes from the subclass `RegexParsers` of `Parsers`.

```
class RegexParsers extends Parsers {
```

This class is more specialized than class `Parsers` in that it only works for inputs that are sequences of characters:

```
type Elem = Char
```

It defines two methods, `literal` and `regex`, with the following signatures:

```
implicit def literal(s: String): Parser[String] = ...
implicit def regex(r: Regex): Parser[String] = ...
```

Note that both methods have an `implicit` modifier, so they are automatically applied whenever a `String` or a `Regex` is given but a `Parser` is expected. That's why you can write string literals and regular expressions directly in a grammar, without having to wrap them with one of these methods. For instance, the parser `"(~expr~)"` expands to `literal("(")~expr~literal(")")`.

The `RegexParsers` class also takes care of handling whitespace between symbols. To do this, it calls a method named `handleWhiteSpace` before running a `literal` or `regex` parser. The `handleWhiteSpace` method skips the longest input sequence that conforms to the `whiteSpace` regular expression, which is defined by default as follows:

```
protected val whiteSpace = """\s+""".r
} // end RegexParsers
```

If you want a different treatment of whitespace you can override the `whiteSpace` value. For instance, if you want whitespace not to be skipped at all, you can achieve this by overriding `whiteSpace` with the empty regular expression:

```
object MyParsers extends RegexParsers {
  override val whiteSpace = "".r
  ...
}
```

31.8 Lexing and parsing

The task of syntax analysis is often 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
```

If you want to split your parser into a separate lexer and syntactical analyzer, you should consult the ScalaDocs for these packages. But for simple parsers, the regular expression based approach you have seen before is usually sufficient.

31.9 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. Each alternative parse must have failed, and recursively so at 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. What it does is 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 by the `phrase` method itself. A functional version of the same scheme would be possible, but it would require threading the `lastFailure` value though every parser result, no matter whether this result is a `Success` or a `Failure`.

Putting it all together

The last two methods in class `Parsers` run a given parser on an input. They are implemented as follows:

```
def parse[T](p: Parser[T], in: Input): ParseResult[T] =
    p(in)

def parseAll[T](p: Parser[T], in: Input): ParseResult[T] =
    parse(phrase(p), in)
```

The `parse` method parses some prefix of the given input `in` with the given parser `p`. To do that, it simply applies the parser to the input. The `parseAll` method parses all of the given input `in` with the given parser `p`. To do that it runs `phrase(p)` on the input.

31.10 Backtracking versus LL(1)

The parser combinators employ *backtracking* to choose between different parsers in an alternative. In an expression $P \mid Q$, 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[Aho86] When a combinator parser is formed from such a grammar, it will never backtrack, *i.e.* the input position will never be reset to an earlier value.

For instance, the grammars for arithmetic expressions and JSON terms earlier in this chapter are both LL(1), so the backtracking capabilities of

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.

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 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.

31.11 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 using 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`, `floatingPointNumber`, “`~`”, “`|`” 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) [Aho86]. 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.

Chapter 32

GUI Programming

32.1 Introduction

In this chapter you'll learn how to develop in Scala applications that use a graphical user interface, or GUI for short. The applications you will develop are based on a Scala library which provides access to Java's Swing framework of GUI classes. Conceptually, the Scala library resembles the underlying Swing classes, but it hides much of their complexity. You'll find out that developing simple GUI applications using the framework is actually quite easy.

Even with Scala's simplifications, a framework like Swing is quite rich, with many different classes and many methods in each class. To find your way in such a rich library, it helps to use an IDE such as Scala's Eclipse plugin. The advantage is that the IDE can show you interactively with its command completion which classes are available in a package and which methods are available for objects you reference. This speeds up your learning considerably when you first explore an unknown library space.

32.2 A first Swing Application

As a first swing application, let's start with a Window containing a single button. To program in with Swing, you need to import various classes from Scala's Swing API package.

```
import scala.swing._
```

Here's the code of your first Swing application in Scala:



Figure 32.1: A simple Swing application: initial (left) and resized (right).

```
object FirstSwingApp extends SimpleGUIApplication {  
    def top = new Frame {  
        title = "First Swing App"  
        contents += new Button {  
            text = "Click me"  
        }  
    }  
}
```

If you compile and run that file, you should see a window as shown on the left of [Figure 32.1](#). The window can be resized to a larger size as shown on the right of [Figure 32.1](#).

If you analyze the code above line by line, you'll notice the following elements.

Line 1: The `FirstSwingApp` object inherits from class `scala.swing.SimpleGUIApplication`. This is different to traditional command-line application, which often inherits from class `scala.Application`. The `SimpleGUIApplication` class already defines a `main` method which contains some setup code for Java's Swing framework. The `main` method then proceeds to call the `top` method, which you supply.

Line 2 The `top` method contains the code which defines your top-level GUI component. This is usually some kind of `Frame`—i.e. a window that can contain arbitrary data.

Line 3 Frames have a number of attributes. Two of the most important are the frame's `title` (which will be written in the title bar)@, and its `contents` (which will be displayed in the window itself. In Scala's Swing API, such attributes are modelled as properties. You know from

Section 18.2 that properties are encoded in Scala as pairs of a getter and a setter methods. For instance, the `title` property of a `Frame` object is modelled as a pair a getter method

```
def title: String
```

and a setter method

```
def title_=(s: String)
```

It is this setter method that gets invoked by the assignment

```
title = "First Swing App"
```

The effect of the assignment is that the chosen title is shown in the header of the window. If you leave it out, the Window will display the default title “Untitled Frame”.

Line 4 The `contents` property of the frame is a collection of components, which is initially empty. Components can be added using the `+=` operator. If you need to remove a component later, there’s the `-=` operator for that. In the example above, a single `Button` gets added the `contents` of the `top` frame.

Line 5 The button also gets a title, in this case “Click Me”.

32.3 Panels and layouts

As next step, let’s add some text as a second content element to the `top` frame of our application. The left part of Figure 32.2 shows what the application should look like. To display a text with the button, you could try to simply add the following code at the end of the frame which defines `top`.

```
contents += new Label("No button clicks registered")
```

This adds a new `label` (i.e. a displayed text field) to the `contents` of `top`. However, if you run the application you’ll find that only the label displays whereas the button has vanished. The reason is that even though `top` contains now two components, only the component that got added last is displayed.

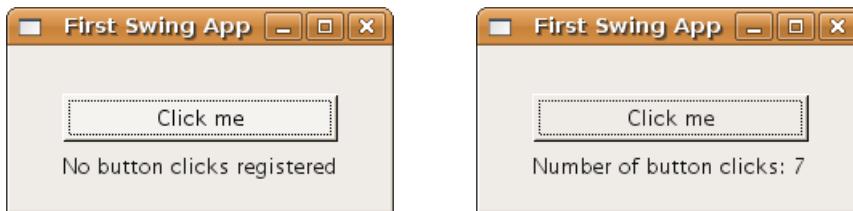


Figure 32.2: A reactive Swing application: initial (left) after user interaction (right).

```
import scala.swing._  
import scala.swing.event._  
  
object FirstSwingApp extends SimpleGUIApplication {  
    def top = new Frame {  
        title = "First Swing App"  
        val button = new Button {  
            text = "Click me"  
        }  
        val label = new Label {  
            text = "No button clicks registered"  
        }  
        contents += new Panel(button, label) {  
            layout = Layout.column  
            border = new EmptyBorder(30, 30, 10, 30)  
        }  
    }  
}
```

Figure 32.3: Component assembly on a panel.

To change that, you need to put all components in a `Panel`. A panel is a container class which displays all the components it contains according to some layout rules which can be changed. Figure 32.3 shows a complete implementation. Here, the two sub-components of the top frame are named `button` and `label`. They are assembled by passing them as arguments to

a new Panel. When doing that you can specify in what way the two sub-components should be laid out. There are a number of possible layouts in the `scala.swing.Layout` object, ranging from simple to quite intricate. In fact, one of the hardest parts of a complex GUI application can be getting the layouts right—it's not easy to come up with something that displays reasonably well on all sorts of devices and for all window sizes. The code in [Figure 32.3](#) picks a simple column layout where components are stacked on top of each other. It also adds a border around the two objects by assigning to the `border` property. Like all other GUI components, borders are represented as objects. `EmptyBorder` is a constructor that takes four parameters indicating the width of the borders on the top, right, bottom, and left sides of the objects to be drawn.

Simple as it is, the example has already shown the principles how to structure a GUI application. It is built from components, which are instances of `scala.swing` classes such as `Frame`, `Panel`, `Label` or `Button`. Components have properties which can be customized by the application. Components also can contain other components in their `contents` property, so that in the end a GUI application consists of a tree of components.

32.4 Handling events

On the other hand, the application still misses an essential property. If you run in [Figure 32.3](#) and click on the displayed button, nothing happens. In fact, the application is completely static; it does not react in any way to user events except for the close button of the top frame, which terminates the application. So as a next step, let's refine the application so that it displays together with the button a label which indicates how often the button was clicked. The right part of [Figure 32.2](#) contains a snapshot of what the application should look like after a few button clicks.

To achieve this behavior, you need to connect a user-input event (the button was clicked) with an action (the displayed label is updated). Java and Scala have fundamentally the same "publish/subscribe" approach to event handling: Components may be publishers and/or subscribers. A publisher publishes events. A subscriber subscribes with a publisher to be notified of any published events. Publishers are also called "event sources" and subscribers are also called "event listeners". For instance a `Button` is an event source which publishes an event `ButtonPressed` indicating that the button

was pressed.

In Scala, subscribing to an event source `source` is done by the call `listenTo(source)`. There's also a way to unsubscribe from an event source using `deafTo(source)`. In the current example application, the first thing to do is to get the top frame to listen to its button, so that it gets notified of any events which the button issues. To do that you need to add the following call to the body of the top frame:

```
listenTo(button)
```

Being notified of events is only half of the story; the other half is handling them. It is here that the Scala Swing framework is most different from (and radically simpler than) the Java Swing API's. In Java an event does not exist as such, instead it is simulated by calling a `notify` method in a component which has to implement some Listener interfaces. Usually, this involves a fair amount of indirection and boiler-plate code which makes event-handling applications hard to write and to read. By contrast, in Scala, an event is a real object that gets sent to subscribing components much like messages are sent to actors. For instance, pressing a button will create an event which is an instance of the following case class

```
case class ButtonPressed(button: Button)
```

The parameter of the case class refers to the button that was pressed. Like for all other Scala events, this event class is contained in a package named `scala.swing.event`.

To have your component react to incoming events you need to add a handler to a property called `reactions`. Here's the code that does this:

```
var nclicks = 0
reactions += {
    case ButtonPressed(b) =>
        nclicks += 1
        label.text = "Number of button clicks: "+nclicks
}
```

The first line above defines a variable `nclicks` that holds the number of times a button was clicked. The rest of the code adds the code between braces as a *handler* to the `reactions` property of the top frame. Handlers are functions defined by pattern matching on events, much like an actor's

receive blocks are defined by pattern matching on messages. The handler above matches events of the form `ButtonPressed(b)`, i.e. any event which is an instance of class `ButtonPressed`. The pattern variable `b` refers to the actual button that was pressed. The action that corresponds to this event in the code above increments the `nclicks` variable and updates the text of the label.

Generally, a handler is always a `PartialFunction` that matches on events and performs some actions. It is also possible to match on more than one kind of event in a single handler by using multiple cases.

The `reactions` property implements a collection, just like the `contents` property does. Some components come already with predefined reactions. For instance, a `Frame` has a predefined reaction that it will close if the user presses the close button on the upper right. If you install your own reactions by adding them with `+=` to the `reactions` property, the reactions you define will be considered in addition to the standard ones. Conceptually, the handlers installed in `reactions` form a stack. In the current example, if the top frame receives an event, the first handler that gets tried is the one which matches on `ButtonPressed`, because it was the last handler that got installed for the frame. If the received event is of type `ButtonPressed`, the code associated with the pattern will be invoked. After that code has completed, the system will search for further handlers in the event stack that might also be applicable. If the received event is not of type `ButtonPressed`, the event is immediately propagated to the rest of the installed handler stack. It's also possible to remove handlers from the `reaction` property. Just as for `contents`, you use the `--=` operator for that.

Figure 32.4 shows the completed application, including reactions. The code illustrates the essential elements of a GUI application in Scala's swing framework: The application consists of a tree of components, starting with the top frame. The components shown in the code are `Frame`, `Panel`, `Button`, and `Label`, but there are many other kinds of components defined in the Swing libraries. Each component is customized by setting attributes. Two important attributes are `contents`, which fixes the children of a component in the tree, and `reactions`, which determines how the component reacts to events.

```
import scala.swing._  
import scala.swing.event._  
  
object ReactiveSwingApp extends SimpleGUIApplication {  
    def top = new Frame {  
        title = "First Swing App"  
        val button = new Button {  
            text = "Click me"  
        }  
        val label = new Label {  
            text = "No button clicks registered"  
        }  
        contents += new Panel(button, label) {  
            layout = Layout.column  
            border = new EmptyBorder(30, 30, 10, 30)  
        }  
        listenTo(button)  
        var nclicks = 0  
        reactions += {  
            case ButtonPressed(b) =>  
                nclicks += 1  
                label.text = "Number of button clicks: "+nclicks  
        }  
    }  
}
```

Figure 32.4: Implementing a reactive Swing application.

32.5 Example: Celsius/Fahrenheit converter

As another example, let's write a GUI program which converts between temperature degrees in Celsius and Fahrenheit. The user interface of the program is shown in [Figure 32.5](#). It consists of two text fields (shown in white) with a label in front of each. One text field shows temperatures in degrees Celsius, the other in degrees Fahrenheit. Each of the two fields can be edited by the user of the application. Once the user has changed the temperature in either field, the temperature in the other field should be automatically updated.



Figure 32.5: A converter between degrees Celsius and Fahrenheit.

Figure 32.6 shows the complete code that implements this application. Here are some explications:

The imports at the top of the code use a short-hand:

```
import swing._  
import event._
```

This is in fact equivalent to the imports used before

```
import scala.swing._  
import scala.swing.event._
```

The reason you can use the shorthand is that packages nest in Scala. Because package `scala.swing` is contained in package `scala`, and everything in that package imported automatically, you can write just `swing` to refer to the package. Likewise, package `scala.swing.event`, is contained as subpackage `event` in package `scala.swing`. Because you have imported everything in `scala.swing` in the first import, you can refer to the `event` package with just `event` thereafter.

The `TempConverter` components has two children, `celsius` and `fahrenheit`. Both are objects of class `TextField`. A `TextField` in Swing is a field of fixed width which you can edit. The width is given in the `columns` property (set to 5 for both fields).

The contents of `TempConverter` are assembled into a `Panel` which includes the two textfields and two labels that explain what the fields are. The `layout` property of the `Panel` is not set. This means that a default layout will be used for the panel. The default layout is a so-called "flow-layout" which displays all elements in one or more rows, depending on the width of the frame.

The reactions of `TempConverter` are defined by a handler which contains two cases. Each case matches a `TextModified` event for one of the

```
import swing._  
import event._  
  
object TempConverter extends SimpleGUIApplication {  
    def top = new Frame {  
        title = "Celsius/Fahrenheit Converter"  
        object celsius extends TextField { columns = 5 };  
        object fahrenheit extends TextField { columns = 5 }  
        contents += new Panel(  
            celsius, new Label(" Celsius = "),  
            fahrenheit, new Label(" Fahrenheit")) {  
            border = new EmptyBorder(15, 10, 10, 10)  
        }  
        listenTo(celsius, fahrenheit)  
        reactions += {  
            case TextModified(`fahrenheit`) =>  
                val f = fahrenheit.text.toInt  
                val c = (f - 32) * 5 / 9  
                celsius.text = c.toString  
            case TextModified(`celsius`) =>  
                val c = celsius.text.toInt  
                val f = c * 9 / 5 + 32  
                fahrenheit.text = f.toString  
        }  
    }  
}
```

Figure 32.6: An implementation of the temperature converter.

two text fields. Such an event gets issued when a textfield has been edited. Note the form of the patterns with back ticks around the element names:

```
case TextModified(`celsius`)
```

As was explained in Chapter 15, the back ticks around `celsius` ensure that the pattern matches only if the `celsius` object was not modified. If you had omitted them and has written just `case TextModified(celsius)`,

the pattern would have matched every event of class `TextModified`. The changed field would then be stored in the pattern variable `celsius`. Obviously, this is not what you want. Alternatively, you could have defined the two `TextField` objects in upper case, i.e. `Celsius`, `Fahrenheit`. In that case you could have matched them directly without backslashes, as in `case TextModified(Celsius)`.

The two actions of the `TextModified` events convert one quantity to another. Each starts by reading out the contents of the modified field and converting it to an `Int`. It then applies the formula for converting one temperature degree to the other, and stores the result back as a string in the other text field.

32.6 Conclusion

This chapter has given you a first taste of GUI programming, using Scala's wrappers for the Swing framework. It has shown how to assemble GUI components, how to customize their properties, and how to handle events. For space reasons, we could discuss only a small number of simple components. There are many more kinds of components. You can find out about them by consulting the Scala documentation of the package `scala.swing`. The next section will develop an example of a more complicated Swing application.

There are also many tutorials on the original Java Swing framework, on which the Scala wrapper is based. See for instance [J]. The Scala wrappers resemble the underlying Swing classes, but try to simplify concepts where possible and make them more uniform. The simplification makes extensive use of the properties of the Scala language. For instance, Scala's emulation of properties and its operator overloading allow convenient property definitions using assignments and “`+≡`” operations. Its “everything is an object” philosophy makes it possible to inherit the main method of a GUI application. The method can thus be hidden from user applications, including the boilerplate code for setting things up that comes with it. Finally, and most importantly, Scala's first-class functions and pattern matching allow to formulate event handling as the `reactions` component property, which greatly simplifies life for the application developer.

Chapter 33

The SCells Spreadsheet

33.1 Introduction

In the previous chapters you have seen many different constructs of the Scala programming language. In this chapter you'll see how these constructs play together in the implementation of a sizable application. The task is to write a spreadsheet application, which will be named SCells. There are several reasons why this task is interesting. First, everybody knows spreadsheets, so it is easy to understand what the application should do. Second, spreadsheets are programs that exercise a large range of different computing tasks. There's the visual aspect, where a spreadsheet is seen as a rich GUI application. There's the symbolic aspect, having to do with formulas and how to parse and interpret them. There's the calculational aspect, dealing with how to update possibly large tables incrementally. There's the reactive aspect, where spreadsheets are seen as programs that react in intricate ways to events. Finally, there's the component aspect where the application is constructed as a set of re-usable components. All these aspects will be treated in depth below.

33.2 The visual framework

Let's start by writing the basic visual framework of the application. [Figure 33.1](#) shows the first iteration of the user interface. You can see that a spreadsheet is a scrollable table. It has rows going from 0 to 99 and columns going from A to Z. You express this in Swing by defining a spreadsheet as a `ScrollPane` containing a `Table`. [Figure 33.2](#) shows the code which pro-

The screenshot shows a window titled "ScalaSheet". The window contains a grid of cells with the following data:

	A	B	C	D	E	F
0						
1	Low price:	0.99				
2	High price:	1.21				
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

The table has 16 rows and 6 columns. Rows 1 and 2 contain text entries: "Low price:" and "High price:". Row 2 also contains numerical values: 0.99 and 1.21. The cell containing 1.21 is highlighted with a black border. The window has scroll bars on the right and bottom.

Figure 33.1: A simple spreadsheet table.

duces this table.

The spreadsheet component is defined in a new package `scells`, which will contain all source files needed for the application. It imports from package `scala.swing` essential elements of Scala's Swing wrapper. `SpreadSheet` itself is a class which takes the height and width (in numbers of cells) as parameters. The class extends `ScrollPane`, which gives it the scroll-bars at the bottom on the right in [Figure 33.1](#). It contains two sub-components named `table` and `rowHeader`.

The `table` component has class `scala.swing.Table`. The four lines in its body set some of its attributes: `rowHeight` for the height of a table row in points, `autoResizeMode` to turn auto-sizing the table off, `showGrid` to show a grid of lines between cells, and `gridColor` to set the color of the grid to a dark grey.

```
package scells
import swing._

class SpreadSheet(val height: Int, val width: Int)
extends ScrollPane {

    val table = new Table(height, width) {
        rowHeight = 25
        autoResizeMode = Table.AutoResizeMode.Off
        showGrid = true
        gridColor = new java.awt.Color(150, 150, 150)
    }

    val rowHeader =
        new ComponentList((0 until height) map (_.toString)) {
            fixedCellWidth = 30
            fixedCellHeight = table.rowHeight
        }

    viewportView = table
    rowHeaderView = rowHeader
}
```

Figure 33.2: Code for spreadsheet in [Figure 33.1](#).

The `rowHeader` component contains the headers on the left of the spreadsheet in [Figure 33.1](#). It's a `ComponentList` which displays the strings from 0 through 99 in its elements. The two lines in its body fix the width of a cell in this list to be 30 points and its height to be the same as the `table`'s `rowHeight`.

The whole spreadsheet is assembled by fixing two fields in `ScrollPane`. The field `viewportView` is set to the `table`, and the field `rowHeaderView` is set to the `rowHeader` list. The difference between the two views is the following: A view port of a scroll pane is the area that is scrolls with the two bars, whereas the row header on the left stays fixed when you move the horizontal scroll bar. By some quirk, Swing already supplies by default a column header at the top of the table, so there's no need to define one explicitly.

```
package scells
import swing._

object Main extends SimpleGUIApplication {
    def top = new Frame {
        title = "ScalaSheet"
        contents = new SpreadSheet(100, 26)
    }
}
```

Figure 33.3: The main program for the spreadsheet application.

Now that you have defined a rudimentary spreadsheet, you can try it out immediately. You just need to define a main program that contains a `SpreadSheet` component. Such a program is shown in [Figure 33.3](#).

The `Main` program inherits from `SimpleGUIApplication`, which takes care of all the low-level details that need to be set up before a Swing application can be run. You only need to define the top-level window of the application in the `top` method. In our example, `top` is a `Frame` that has two elements defined: Its `title`, set to “`ScalaSheet`,” and its `contents`, set to an instance of class `SpreadSheet` with 100 rows and 26 columns. That’s all. If you launch this application with `@scala scells.Main@` you should see the spreadsheet in [Figure 33.1](#).

33.3 Disconnecting data entry and display

If you play a bit with the spreadsheet written so far, you’ll quickly notice that the output that’s displayed in a cell is always exactly what you entered in the cell. A real spreadsheet does not behave like that. In a real spreadsheet, you would enter a formula and you’d see its value. So what is entered into a cell is different from what is displayed.

As a first step to a real spreadsheet application, you should concentrate on disentangling data entry and display. The basic mechanism for display is contained in the `render` method of class `Table`. By default, `render` always displays what’s entered. If you want to change that, you need to override `render` to do something different. [Figure 33.4](#) shows a new version of

```
class SpreadSheet(val height: Int, val width: Int)
extends ScrollPane {

    val cellModel = new Model(height, width)
    import cellModel._

    val table = new Table(height, width) {
        ... // settings as before

        override def render(isSelected: Boolean, hasFocus: Boolean,
                            row: Int, column: Int): Component =
            if (hasFocus) new TextField(userData(row, column))
            else new Label(cells(row, column).toString) {
                xAlignment = Alignment.RIGHT
            }

        def userData(row: Int, column: Int): String = {
            val v = this(row, column)
            if (v == null) "" else v.toString
        }
    }
    ... // rest as before
}
```

Figure 33.4: A spreadsheet component with a `render` method.

`SpreadSheet` with a `render` method.

The `render` method overrides a default method of the same name in class `Table`. It takes four parameters: `isSelected` and `hasFocus` are Booleans which tell whether the cell has been selected and whether it has focus, meaning that keyboard events go into the cell. The remaining two parameters `row` and `column` give the cell's coordinates.

The new `render` method distinguishes whether the cell has input focus or not. If `hasFocus` is true, the cell is used for editing. In this case you want to display an editable `TextField` which contains the data that the user has entered so far. This data is returned by the auxiliary method `userData`, which displays the contents of the table at a given `row` and `column`. The contents are retrieved by the call `this(row,`

```
package scells
class Model(val height: Int, val width: int) {
    case class Cell(row: Int, column: Int)
    val cells = new Array[Array[Cell]](height, width)
    for (i <- 0 until height; j <- 0 until width)
        cells(i)(j) = new Cell(i, j)
}
```

Figure 33.5: First version of the Model class.

column). The `userData` method also takes care to display a null table element as the empty string instead of printing “null”.

So far so good. But what should be displayed if the cell does not have focus? In a real spreadsheet this would be the value of a cell. So you see that there are really two tables at work: The first table, named `table` contains what the user entered. A second “shadow” table contains the internal representation of cells and what should be displayed. In the spreadsheet example, this table is a two-dimensional array called `cells`; If a cell at a given `row` and `column` does not have editing focus, the `render` method will display the element `cells(row)(column)`. The element cannot be edited, so it should be displayed in a `Label`, instead of in an editable `TextField`.

It remains to define the internal array of cells. You could do this directly in the `SpreadSheet` class, but it’s generally preferable to separate the view of a GUI component from its internal model. That’s why in the example above the `cells` array is defined in a separate class named `Model`. The model is integrated into the `SpreadSheet` by defining a value `cellModel` of type `Model`. The `import` clause which follows this value definition makes the contents of `cellModel` available inside `SpreadSheet` without having to prefix them. [Figure 33.5](#) shows a first simplified version of a `Model` class. The class defines an inner class `Cell` and a two-dimensional array `cells` of `Cell` elements. Each element is initialized to be a fresh `Cell`.

That’s it. If you compile the modified `SpreadSheet` class with the `Model` trait and run the `Main` application you should see a window as in [Figure 33.6](#).

The objective of this section was to arrive at a design where the displayed value of a cell is different from the string that was entered into it. This objec-

	A	B	C	D	E	F
0	Cell(0,0)	Cell(0,1)	Cell(0,2)	Cell(0,3)	Cell(0,4)	Cell(0,5)
1	Cell(1,0)	Cell(1,1)	Cell(1,2)	Cell(1,3)	Cell(1,4)	Cell(1,5)
2	Cell(2,0)	Cell(2,1)	Cell(2,2)	Cell(2,3)	Cell(2,4)	Cell(2,5)
3	Cell(3,0)	Cell(3,1)	Cell(3,2)	Cell(3,3)	Cell(3,4)	Cell(3,5)
4	Cell(4,0)	Cell(4,1)	Cell(4,2)	Cell(4,3)	Cell(4,4)	Cell(4,5)
5	Cell(5,0)	Cell(5,1)	Cell(5,2)	Cell(5,3)	Cell(5,4)	Cell(5,5)
6	Cell(6,0)	Cell(6,1)	Cell(6,2)	Cell(6,3)	Cell(6,4)	Cell(6,5)
7	Cell(7,0)	Cell(7,1)	Cell(7,2)	Cell(7,3)	Cell(7,4)	Cell(7,5)
8	Cell(8,0)	Cell(8,1)	Cell(8,2)	Cell(8,3)	Cell(8,4)	Cell(8,5)
9	Cell(9,0)	Cell(9,1)	Cell(9,2)	Cell(9,3)	Cell(9,4)	Cell(9,5)
10	Cell(10,0)	Cell(10,1)		Cell(10,3)	Cell(10,4)	Cell(10,5)
11	Cell(11,0)	Cell(11,1)	Cell(11,2)	Cell(11,3)	Cell(11,4)	Cell(11,5)
12	Cell(12,0)	Cell(12,1)	Cell(12,2)	Cell(12,3)	Cell(12,4)	Cell(12,5)
13	Cell(13,0)	Cell(13,1)	Cell(13,2)	Cell(13,3)	Cell(13,4)	Cell(13,5)
14	Cell(14,0)	Cell(14,1)	Cell(14,2)	Cell(14,3)	Cell(14,4)	Cell(14,5)
15	Cell(15,0)	Cell(15,1)	Cell(15,2)	Cell(15,3)	Cell(15,4)	Cell(15,5)

Figure 33.6: Cells displaying themselves.

tive has clearly been met, albeit in a very crude way. In the new spreadsheet you can enter anything you want into a cell, but it will always display just its coordinates once it loses focus. Clearly, you are not done yet.

33.4 Formulas

In reality, a spreadsheet cell holds two things: An actual *value* and a *formula* to compute this value. There are three types of formulas in a spreadsheet:

1. Numeric values such as 1.22, -3, or 0.
2. Textual labels such as Annual sales, Deprecation, total.

3. Formulas that compute a new value from the contents of cells, such as
`=add(A1,B2)`, or `=sum(mul(2, A2), C1:D16)`

A formula that computes a value always starts with an equals sign and is followed by an arithmetic expression. The SCells spreadsheet has a particularly simple and uniform convention for arithmetic expressions: every expression is an application of some function to a list of arguments. The function name is an identifier such as `add` for binary addition, or `sum` for summation of an arbitrary number of operands. A function argument can be a number, a reference to a cell, a reference to a range of cells such as `C1:D16`, or another function application. You'll see later that SCells has an open architecture that makes it easy to install your own functions via mixin composition.

The first step to handling formulas is writing down the types that represent them. As you would expect, the different kinds of formulas are represented by case classes. [Figure 33.7](#) shows the contents of a file `Formulas.scala`, where these case classes are defined.

The root of the class hierarchy is a trait `Formula`. This trait has five case classes as children:

<code>Coord</code>	for cell coordinates such as <code>A3</code> ,
<code>Range</code>	for cell ranges such as <code>A3:B17</code> ,
<code>Number</code>	for floating-point numbers such as <code>3.1415</code> ,
<code>Textual</code>	for textual labels such as <code>Deprecation</code> ,
<code>Application</code>	for function applications such as <code>sum(A1,A2)</code> .

Each case class overrides the `toString` method so that it displays its kind of formulas in the standard way shown above. For convenience there's also an `Empty` object which represents the contents of an empty cell. As you would expect, the `Empty` object is an instance of the `Textual` class with an empty string argument.

33.5 Parsing formulas

In the previous section you have seen the different kinds of formulas and how they display as strings. In this section you'll see how to reverse the process; that is how to transform a user input string into a `Formula` tree. The rest of this section explains one by one the different elements of a class `FormulaParsers`, which contains the parsers that do the transformation.

```

package scells

trait Formula

case class Coord(row: Int, column: Int) extends Formula {
    override def toString = ('A' + column).toChar.toString + row
}

case class Range(c1: Coord, c2: Coord) extends Formula {
    override def toString = c1.toString + ":" + c2.toString
}

case class Number(value: Double) extends Formula {
    override def toString = value.toString
}

case class Textual(value: String) extends Formula {
    override def toString = value.toString
}

case class Application(function: String,
                      arguments: List[Formula]) extends Formula {
    override def toString =
        function + arguments.mkString("(", ", ", ", ", ")
}

object Empty extends Textual("")

```

Figure 33.7: Classes representing formulas.

The class builds on the combinator framework given in [Chapter 31](#). Specifically, formula parsers are an instance of the `RegexParsers` class explained in that chapter.

```

package scells
import scala.util.parsing.combinator._

object FormulaParsers extends RegexParsers {

```

The first two elements of the `FormulaParser` class are auxiliary parsers for identifiers and decimal numbers:

```

def ident: Parser[String] = """[a-zA-Z_]\w*""".r
def decimal: Parser[String] = """-?\d+(\.\d*)?""".r

```

As you can see from the regular expressions above, an identifier starts with a letter or underscore, which is followed by an arbitrary number of “word” characters represented by the regular expression code ‘\w’—these are letters, digits or underscores. A whole number consists of one or more digits.

The next element of class `FormulaParsers` is the cell parser, which recognizes the coordinates of a cell. It first calls a regular expression parser which determines the form of a coordinate: A single letter followed by one or more digits. The string returned from that parser is then converted to a cell coordinate by separating the letter from the numerical part and converting the two parts to indices for the cell’s column and row.

```
def cell: Parser[Coord] =
  """[A-Za-z]\d+""".r ^^ { s =>
  val column = s.charAt(0) - 'A'
  val row = s.substring(1).toInt
  Coord(row, column)
}
```

Note that the `cell` parser is a bit restrictive in that it allows only column coordinates consisting of a single letter. Hence the number of spreadsheet columns is in effect restricted to be at most 26, because further columns cannot be parsed. It’s a good idea to generalize the parser so that it accepts cells with several leading letters. This is left as an exercise to you.

The `range` parser recognizes a range of cells. Such a range is composed of two cell coordinates with a colon between them.

```
def range: Parser[Range] =
  cell~":"~cell ^^ {
    case c1~":"~c2 => Range(c1, c2)
}
```

The `number` parser recognizes a decimal number, which is converted to a `Double` and wrapped in an instance of the `Number` class.

```
def number: Parser[Number] =
  decimal ^^ Number(_.toDouble)
```

The `application` parser recognizes a function application. Such an application starts with an identifier, which is followed by a list of argument expressions in parentheses.

```
def application: Parser[Application] =
  ident ~ ("~repsep(expr, ",")~") ^^ {
    case f ~ ("~ps~") => Application(f, ps)
  }
```

The `expr` parser recognizes a formula expression—either a top-level formula following a ‘=’, or an argument to a function. Such a formula expression is defined to be a cell, a range of cells, a number, or an application.

```
def expr: Parser[Formula] =
  cell | range | number | application
```

This definition of the `expr` parser contains a slight oversimplification because ranges of cells should only appear as function arguments; they are should not to be allowed as top-level formulas. You could change the formula grammar so that the two uses of expressions are separated, and ranges are excluded syntactically from top-level formulas. In the spreadsheet presented here such an error is instead detected once an expression is evaluated.

The `textual` parser recognizes an arbitrary input string, as long as it does not start with an equals sign (recall that strings which start with “=” are considered to be formulas).

```
def textual: Parser[Textual] =
  """[^=].*""".r ^^ Textual
```

The `formula` parser recognizes all kinds of legal inputs into a cell. A formula is either a number, or a `textual` entry, or a formula starting with an equals sign.

```
def formula: Parser[Formula] =
  number | textual | "=" ~> expr
```

This concludes the grammar for spreadsheet cells. The final method `parse` uses this grammar in a method that converts an input string into a `Formula` tree:

```
def parse(input: String): Formula =
  parseAll(formula)(input) match {
    case Success(e, _) => e
    case f: NoSuccess => Textual("[ "+f.msg+"]")
  }
```

```
    } //end FormulaParsers
```

The `parse` method parses all of the input with the `formula` parser. If that succeeds, the resulting formula is returned. If it fails, a `Textual` object with an error message is returned instead.

That's everything there is to parsing formulas. It only remains to integrate the parser into the spreadsheet. To do this, you can enrich the `Cell` class in class `Model` by a `formula` field:

```
case class Cell(row: Int, column: Int) {
    var formula: Formula = Empty
    override def toString = formula.toString
}
```

In the new version of the `Cell` class, the `toString` method is defined to display the cell's formula. That way you can check whether formulas have been correctly parsed.

The last step in this section is to integrate the parser into the spreadsheet. Parsing a formula happens as a reaction to the user's input into a cell. A completed cell input is modelled in the Swing library by a `TableChanged` event. The `TableChanged` class is contained in package `swing.event`. The event is of the form

```
TableChanged(table, firstRow, lastRow, column)
```

It contains the `table` that was changed and an interval of cell coordinates between rows `firstRow` and `lastRow` on a given `column` that were affected.

Once a table is changed, the affected cells need to be re-parsed. To react to a `TableChange` event, you add a case to the `reactions` value of the `table` component in the way it is shown below:

```
package scells
import swing._, swing.event._
class SpreadSheet(val height: Int, val width: Int) ... {
    val table = new Table(height, width) {
        ...
        reactions += {
            case TableChanged(table, firstRow, lastRow, column) =>
                for (row <- firstRow to lastRow)
                    cells(row)(column).formula =
```

	A	B	C	D	E	F
0						
1	test data		10.0			
2			11.0			
3			12.0			
4			13.0			
5			14.0			
6			15.0			
7		prod(B1:B6)				
8		=add(1,X)				
9						
10						
11						
12						
13						
14						
15						

Figure 33.8: Cells displaying their formulas.

```
    FormulaParsers.parse(userData(row, column))  
}  
}  
}  
}
```

Now, whenever the table is edited the formulas of all affected cells will be updated by parsing the corresponding user data. When compiling the classes discussed so far and launching the `scells.Main` application you should see a spreadsheet application like the one shown in [Figure 33.8](#). You can edit cells by typing into them. After editing is done, a cell displays the formula it contains. You can also try to type some illegal input such as the one reading `=add(1, X)` in the field that has the editing focus in [Figure 33.8](#). Illegal input will show up as an error message. For instance, once you'd leave the edited

field in Figure 33.8 you should see the error message [`(` expected] in the cell (to see all of the error message you might need to widen the column by dragging the separation between the column headers to the right).

33.6 Evaluation

Of course, in the end a spreadsheet should evaluate formulas, not just display them. In this section, you'll add the necessary components to achieve this.

What's needed is a method `evaluate` that takes a formula and returns the value of that formula in the current spreadsheet, represented as a `Double`. Let's place this method in a new trait, `Evaluator`. The method needs to access the `cells` field in class `Model` to find out about the current values of cells that are referenced in a formula. On the other hand, the `Model` class needs to call `evaluate`. Hence, there's a mutual dependency between the `Model` and the `Evaluator`. A good way to express such mutual dependencies between classes was shown in Chapter 27: You use inheritance in one direction and self types in the other.

In the spreadsheet example, class `Model` inherits from `Evaluator` and thus gains access to its `evaluate` method. To go the other way, class `Evaluator` defines its self type to be `Model`, like this:

```
package scells
trait Evaluator { this: Model => ...
```

That way, the `this` value inside class `Evaluator` is assumed to be `Model` and the `cells` array is accessible writing either `cells` or `this.cells`.

Now that the wiring is done you can concentrate on defining the contents of class `Evaluator`. Here's the implementation of the `evaluate` method:

```
def evaluate(e: Formula): Double = try {
  e match {
    case Coord(row, column) =>
      cells(row)(column).value
    case Number(v) =>
      v
    case Textual(_) =>
      0
    case Application(function, arguments) =>
```

```
    val argvals = arguments flatMap evalList
    operations(function)(argvals)
  }
} catch {
  case ex: Throwable => Math.NaN_DOUBLE
}
```

As you would expect, the method contains a pattern match over the different types of formulas. For a coordinate `Coord(row, column)`, it returns the value of the `cells` array at that coordinate. For a number `Number(v)`, it returns the value `v`. For a textual label `Textual(s)`, it returns zero. Finally, for an application `Application(function, arguments)`, it computes the values of all arguments, retrieves a function object corresponding to the function name from an `operations` table and applies that function to all argument values.

The `operations` table maps function names to function objects. It is defined as follows:

```
type Op = List[Double] => Double
val operations = new collection.mutable.HashMap[String, Op]
```

As you can see from this definition, operations are modelled as functions from lists of values to values. The `Op` type introduces a convenient alias for the type of an operation.

The computation in `evaluate` is wrapped in a try-catch to guard against input errors. There are actually quite a few things that can go wrong when evaluating a cell formula: Coordinates might be out of range, function names might be undefined, functions might have the wrong number of arguments, arithmetic operations might be illegal or overflow. The reaction to any of these errors is the same: a “not-a-number” value is returned. The returned value, `Math.NaN_DOUBLE` is the IEEE representation for a computation that does not have a representable floating-point value. This might happen because of an overflow or a division by zero, for example. The `evaluate` method above chooses to return the same value also for all other kinds of errors. The advantage of this scheme is that it is simple to understand and that it does not require much code to implement. Its disadvantage is that all kinds of errors are lumped together, so a user of the spreadsheet does not get any detailed feedback what went wrong. If you wish you can experiment with more refined ways of representing errors in the `SCells` application.

The evaluation of arguments is different from the evaluation of top-level formulas. Arguments might be lists whereas top-level functions may not. For instance, the argument expression A1:A3 in `sum(A1:A3)` returns the values of cells A1, A2, A3 in a list. This list is then passed to the `sum` operation. It's also possible to mix lists and single values in argument expressions; for instance the operation `sum(A1:A3, 1.0, C7)`, which would sum up five values. To handle arguments that might evaluate to lists, there's another evaluation function, called `evalList`. This function takes a formula and returns a list of values. Here is its definition:

```
private def evalList(e: Formula): List[Double] = e match {
    case Range(_, _) => references(e) map (_.value)
    case _ => List(evaluate(e))
}
```

If the formula argument passed to `evalList` is a `Range`, the returned value is a list consisting of the values of all cells referenced by the range. For every other formula the result is a list consisting of the single result value of that formula.

The cells referenced by a formula are computed by a third function `references`. Here is its definition:

```
def references(e: Formula): List[Cell] = e match {
    case Coord(row, column) =>
        List(cells(row)(column))
    case Range(Coord(r1, c1), Coord(r2, c2)) =>
        for (row <- (r1 to r2).toList; column <- c1 to c2)
            yield cells(row)(column)
    case Application(function, arguments) =>
        arguments flatMap references
    case _ =>
        List()
```

}

} // end Evaluator

The `references` method is actually more general than needed right now in that it computes the list of cells referenced by any sort of formula, not just a `Range` formula. It will turn out later that the added functionality is needed to compute the sets of cells that need updating. The body of the method

```

package scells
trait Arithmetic { this: Evaluator =>
  operations += (
    "add"  -> { case List(x, y) => x + y },
    "sub"  -> { case List(x, y) => x - y },
    "div"  -> { case List(x, y) => x / y },
    "mul"  -> { case List(x, y) => x * y },
    "mod"  -> { case List(x, y) => x % y },
    "sum"  -> { xs => (0.0 /: xs)(_ + _) },
    "prod" -> { xs => (1.0 /: xs)(_ * _) }
  )
}

```

Figure 33.9: A library for arithmetic operations.

is a straightforward pattern match on kinds of formulas. For a coordinate `Coord(row, column)`, it returns a singleton list containing the cell at that coordinate. For a range expression `Range(coord1, coord2)` it returns all cells between the two coordinates, computed by a for-expression. For a function application `Application(function, arguments)` it returns the cells referenced by each argument expression, concatenated via `flatMap` into a single list. For the other two types of formulas (`Textual` and `Number`), it returns an empty list.

33.7 Operation Libraries

The class `Evaluator` itself defines no operations that can be performed on cells; its `operations` table is initially empty. The idea is to define such operations in other traits which are then mixed into the `Model` class. [Figure 33.9](#) shows an example trait that implements common arithmetic operations.

Interestingly, this trait has no exported members. The only thing it does is populate the `operations` table during its initialization. It gets access to that table by using a self type `Evaluator`, i.e. by the same technique which the `Arithmetic` class used to get access to the model.

Of the seven operations which are defined by the `Arithmetic` trait, five

are binary operations and two take an arbitrary number of arguments. The binary operations all follow the same schema. For instance, the addition operation add is defined by the expression:

```
{ case List(x, y) => x + y }
```

That is, it expects an argument list consisting of two elements x and y and returns the sum of x and y. If the argument list contains a number of elements different from two, a `MatchError` is thrown. This corresponds to the general “let-it-crash” philosophy of SCell’s evaluation model, where incorrect input is expected to lead to a runtime exception that then gets caught by the try-catch inside the evaluation method.

The last two operations sum and prod take a list of arguments of arbitrary length and insert a binary operation between successive elements. So they are instances of the “fold-left” schema that’s expressed in class `List` by the operation `/::`. For instance, to sum a list of numbers `List(x, y, z)`, the operation computes $0 + x + y + z$. The first operand 0 is the result that’s returned if the list is empty.

You can integrate this operation library into the spreadsheet application by mixing the `Arithmetic` trait into the `Model` class, like this:

```
package scells

class Model(val height: Int, val width: int)
extends Evaluator with Arithmetic {

  case class Cell(row: Int, column: Int) {
    var formula: Formula = Empty
    def value = evaluate(formula)

    override def toString = formula match {
      case Textual(s) => s
      case _ => value.toString
    }
  }
  ...
}
```

Another change in the `Model` class concerns the way cells display themselves. In the new version, the displayed value of a cell depends on its formula. If the formula is a `Textual` field, the contents of the field are displayed

	A	B	C	D	E	F
0						
1	Test data	10.0	20.0			
2		11.0	21.0			
3		12.0	22.0			
4		13.0	23.0			
5		46.0	=sum(C1:C4)			
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

Figure 33.10: Cells that evaluate.

literally. In all other cases, the formula is evaluated and the result value of that evaluation is displayed.

If you compile the changed traits and classes and re-launch the `scells.Main` program you get something that starts to resemble a real spreadsheet. Figure 33.10 shows an example. You can enter formulas into cells and get them to evaluate themselves. For instance, once you close the editing focus on cell C5 in Figure 33.10, you should see 86.0, the result of evaluating the formula `sum(C1:C4)`.

However, there's a crucial element still missing. If you change the value of cell C1 in Figure 33.10 from 20 to 100, the sum in cell C5 will not be automatically updated to 166. You'll have to click on C5 manually to see a change in its value. What's still missing is a way to have cells recompute their values automatically after a change.

33.8 Change Propagation

If a cell's value has changed, all cells that depend on that value should have their results recomputed and redisplayed. The simplest way to achieve this would be to recompute the value of every cell in the spreadsheet after each change. However such an approach does not scale well as the spreadsheet grows in size.

A better approach is to recompute the values of only those cells that refer to a changed cell in their formula. The idea is to use an event-based publish/subscribe framework for change propagation: Once a cell gets assigned a formula it will subscribe to be notified of all value changes in cells that are referred by the formula. A value change in one of these cells will trigger a re-evaluation of the subscriber cell. If such the re-evaluation causes a change in the value of the cell, it will in turn notify all cells that depend on it. The process continues until all cell values have stabilized, i.e. there are no more changes in the values of any cell.¹

The publish/subscribe framework is implemented in class `Model` using the standard event mechanism of Scala's Swing framework. Here's a new (and final) version of this class.

```
package scells
import swing._

class Model(val height: Int, val width: int)
  extends Evaluator with Arithmetic {
```

Compared to the previous version of `Model`, there's a new import of `swing._`, which makes the event abstractions of this library directly available.

The main modifications of class `Model` concern the nested class `Cell`. here is its new version:

```
case class Cell(row: Int, column: Int) extends Publisher {
  override def toString = formula match {
    case Textual(s) => s
    case _ => value.toString
  }
}
```

¹This assumes that there are no cyclic dependencies between cells. We discuss dropping this assumption at the end of this chapter.

Class Cell now inherits from Publisher, so that it can publish events. The event-handling logic is completely contained in the setters of two properties: value and formula. To the outside, it looks like value and formula are two variables in class Cell. Their actual implementation is in terms of two private fields which are equipped with public getters value, formula and setters value_=, formula_. Here are the definitions implementing the value property:

```
private var v: Double = 0
def value: Double = v
def value_=(w: Double) {
    if (!(v == w || v.isNaN && w.isNaN)) {
        v = w
        publish(ValueChanged(this))
    }
}
```

The value_= setter assigns a new value w to the private field v. If the new value is different from the old one, it also publishes a ValueChanged event with the cell itself as argument. Note that the test whether the value has changed is a bit tricky because it involves the value NaN. The Java spec says that NaN is different from every other value, including itself! Therefore, a test whether two values are the same has to treat NaN specially: two values v, w are the same if they are equal with respect to ==, or they are both the value NaN, i.e. v.isNaN and w.isNaN both yield true.

Whereas the value_= setter does the publishing in the publish/subscribe framework, the formula_= setter does the subscribing:

```
private var f: Formula = Empty
def formula: Formula = f
def formula_=(f: Formula) {
    for (c <- references(formula)) deafTo(c)
    this.f = f
    for (c <- references(formula)) listenTo(c)
    value = evaluate(f)
}
```

If a cell is assigned a new formula, it first unsubscribes with deafTo from all cells referenced by the previous formula value. It then stores the new

formula in the private variable `f` and subscribes with `listenTo` to all cells referenced by it. Finally, it recomputes its value using the new formula.

The last piece of code in class `Cell` specifies how to react to a `ValueChanged` event:

```
reactions += {  
    case ValueChanged(_) => value = evaluate(formula)  
}  
} // end class Cell
```

The `ValueChanged` class is also contained in class `Model`:

```
case class ValueChanged(cell: Cell) extends event.Event
```

The rest of class `Model` is as before:

```
val cells = new Array[Array[Cell]](height, width)  
for (i <- 0 until height; j <- 0 until width)  
    cells(i)(j) = new Cell(i, j)  
} // end class Model
```

The spreadsheet code is now almost complete. The final piece missing is the re-display of modified cells. So far, all value propagation concerned the internal `Cell` values only; the visible table was not affected. One way to change this would be to add a `redraw` command to the `value_=` setter. However, this would undermine the strict separation between model and view that you have seen so far. A more modular solution is to notify the table of all `ValueChanged` events and let it do the redrawing itself. [Figure 33.11](#) shows the final spreadsheet component which implements this scheme.

There are only two new elements in [Figure 33.11](#) compared to what was discussed before. First, the `table` component now subscribes with `listenTo` to all cells in the model. Second, there's a new case in the `table`'s `reactions`: if it is notified of a `ValueChanged(cell)` event, it demands a `redraw` of the corresponding cell with a call of `markUpdated(cell)`.

33.9 Conclusion

That's all. The resulting spreadsheet is fully functional, even though at some points it adopts the simplest solution rather than the most convenient one.

```
package scells
import swing._, swing.event._

class SpreadSheet(val height: Int, val width: Int)
extends ScrollPane {

    val cellModel = new Model(height, width)
    import cellModel._

    val table = new Table(height, width) {
        ... // settings as in Figure 33.2

        override def render(isSelected: Boolean, hasFocus: Boolean,
                            row: Int, column: Int) =
            ... // as in Figure 33.4

        def userData(row: Int, column: Int): String =
            ... // as in Figure 33.4

        reactions += {
            case TableChanged(table, firstRow, lastRow, column) =>
                for (row <- firstRow to lastRow)
                    cells(row)(column).formula =
                        FormulaParsers.parse(userData(row, column))
            case ValueChanged(cell) =>
                markUpdated(cell.row, cell.column)
        }
        for (row <- cells; cell <- row) listenTo(cell)
    }

    val rowHeader = new ListView(0 until height) {
        fixedCellWidth = 30
        fixedCellHeight = table.rowHeight
    }

    viewportView = table
    rowHeaderView = rowHeader
}
```

Figure 33.11: The finished spreadsheet component

That way, it could be written in just under 200 lines of code. Nevertheless, the architecture of the spreadsheet makes modifications and extensions easy. In case you would like to experiment with the code a bit further, here are some suggestions of what you could change or add.

1. You could make the spreadsheet resizable, so that the number of rows and columns can be changed interactively.
2. You could add new kinds of formulas, for instance binary operations, or other functions.
3. You might think about what to do when cells refer recursively to themselves. For instance, if cell A1 holds the formula `add(B1, 1)` and cell B1 holds the formula `mul(A1, 2)`, a re-evaluation of either cell will trigger a stack-overflow. Clearly, that's not a very good solution. As alternatives, you could either disallow such a situation, or just compute one iteration each time one of the cells is touched.
4. You could error handling, giving more detailed messages what went wrong.
5. You could a formula entry field at the top of the spreadsheet, so that long formulas can be entered more conveniently.

At the beginning of this book we stressed the scalability aspect of Scala. We claimed that the combination of Scala's object-oriented and functional constructs makes it suitable for programs ranging from small scripts to very large systems. The spreadsheet presented here is clearly still a small system, even though it would probably take up much more than 200 lines in most other languages. Nevertheless, you can see many of the details that make Scala scalable at play in this application.

The spreadsheet uses Scala's classes and traits with their mixin composition to combine its components in flexible ways. Recursive dependencies between components are expressed using self types. The need for static state is completely eliminated—the only top-level components that are not classes are formula trees and formula parsers, and both of these are purely functional. The application also uses higher-order functions and pattern matching extensively, both for accessing formulas and for event handling. So it is a good show-case how functional and object-oriented programming can be combined smoothly.

One important reason why the spreadsheet application is so concise is that it can base itself on powerful libraries. The parser combinator library provides in effect an embedded domain-specific language for writing parsers. Without it, reading formulas would have been much more difficult. The event handling in the Swing libraries is a good example of the power of control abstractions. If you know Java's Swing libraries, you probably appreciate the conciseness of Scala's reactions concept, particularly when compared to the tedium of writing notify methods and implementing listener interfaces in the classical publish/subscribe design pattern. So the spreadsheet demonstrates the benefits of extensibility, where high-level libraries can be made to look just like language extensions.

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 *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.

contravariant A *contravariant* annotation can be applied to a type parameter of a class or a trait by putting a minus sign (-) before the type parameter. The class or trait then subtypes contravariantly with—in the opposite direction as—the type annotated parameter. For example, `Function1` is contravariant in its first type parameter, and so `Function1[Any, String]` is a subtype of `Function1[String, String]`.

covariant A *covariant* annotation can be applied to a type parameter of a class or a trait by putting a plus sign (+) before the type parameter. The class or trait then subtypes covariantly with—in the same direction as—the type annotated parameter. For example, `List` is covariant in its type parameter, so `List[String]` is a subtype of `List[Any]`.

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 An existential type includes references to type variables that are unknown. For example, Array[T] forSome { type T } is an existential type. It is an array of T, where T is some completely unknown type. All that is assumed about T is that it exists at all. This assumption is weak, but it means at least that an Array[T] forSome { type T } is an array and not a banana.

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*.

for comprehension A *for comprehension* is another name for *for expression*.

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.

invariant Invariant is used in two ways. It can mean a property that always holds true when a data structure is well-formed. For example, it is an invariant of a sorted binary tree that each node is ordered before its right subnode, if it has a right subnode. Invariant is also sometimes used as a synonym for nonvariant: “class `Array` is invariant in its type parameter.”

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 function A *local function* is a `def` defined inside a block. To contrast, a `def` defined inside a class, trait, or singleton object is called a *method*.

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.

message Actors communicate with each other by sending each other *messages*. Sending a message does not interrupt what the receiver is doing. The receiver can wait until it has finished its current activity and its invariants have been reestablished.

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.`

nonvariant A type parameter of a class or trait is by default *nonvariant*. The class or trait then does not subtype when that parameter changes. For example, `Array` is nonvariant in its type parameter, so `Array[String]` is neither a subtype nor a supertype of `Array[Any]`.

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.

parameterless method A *parameterless method* is a parameterless function that is a member of a class, trait, or singleton object.

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.

path-dependent type A *path-dependent type* is a type like `swiss.cow.Food`. The `swiss.cow` part is a *path*, and the meaning of the type is sensitive to the path you use to access it. The types `swiss.cow.Food` and `fish.Food` are different types.

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 function 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 27](#).

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.

type parameter A *type parameter* is a parameter to a generic class or generic method that must be filled in by a type. For example, class `List` is defined as “`class List[T] { ... }`,” and method `identity` is defined as “`def @identity[T](x:T)@ = x.`” The `T` in both cases is a type parameter.

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.

variance A type parameter of a class or trait can be marked with a *variance* annotation, either *covariant* (+) or *contravariant* (-). Such variance annotations indicate how subtyping works for a generic class or trait. For example, the generic class `List` is covariant in its type parameter, and thus `List[String]` is a subtype of `List[Any]`.

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.
- [Aho86] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.
- [Blo01] Bloch, Joshua. *Effective Java*. Addison-Wesley, 2001.
- [Eck98] Eckel, Bruce. *Thinking in Java*. Prentice Hall, 1998.
- [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.
- [Mey91] Meyers, Scott. *Effective C++*. Addison-Wesley, 1991.
- [Mey00] Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall, 2000.

- [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.
- [Ode08] Odersky, Martin. *The Scala Language Specification, Version 2.7*. EPFL, February 2008. [Http://www.scala-lang.org/docu/manuals.html](http://www.scala-lang.org/docu/manuals.html).
- [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 is a software engineer at Google, Inc. He worked on Scala for two years as a post-doc at EPFL. He has a Ph.D. in computer science from Georgia Tech, where he worked on static analysis of dynamic languages. In addition to Scala, he has worked on a wide variety of programming 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 Machine*, 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.

Index

- *
 - repeated parameter, 171
 - /:
 - method on List, 317
 - ::
 - method on List, 67, 299
 - pattern constructor class, 302
 - :::
 - method on List, 67
 - :\
 - method on List, 318
 - *:ul style="list-style-type: none;"> - repeated parameter, 172
- actors, 33
- alternative
 - catch-all, 272
 - in match expression, 153, 270
- annotations
 - applied to expression, 464
 - arguments to, 465
 - Java, 510
 - standard, 465
 - syntax of, 465
 - type, 30, 44, 50
 - type, for debugging, 327
 - variance, 381
 - with reflection, 513
- application
 - function to arguments, 165
 - partial function, 164, 313
- apply, 64
 - as List() factory, 67, 321
 - as Map() factory, 76
 - as Set() factory, 74
 - method on List, 308
 - the FunctionN method, 160
- args
 - in scripts, 55
- arguments
 - command line, in scripts, 55
 - variable length, 171
- assignment
 - result is unit value, 144
- BeanProperty, 468
- BigInt, 31
- BigInteger, 32
- break
 - living without, 154
 - not in match expressions, 154
- Cathedral and the Bazaar, The, 31
- companion objects, 95
 - visibility, 255
- concat
 - method on List object, 323

cons
 List's :: method, 67, 299
constructors, 62, 123
continue
 living without, 154
contravariance, 381
copyToArray
 method on List, 311
covariance, 381
deprecated, 464, 510
drop
 method on List, 307
dropWhile
 method on List, 316
Eclipse, 48
elements
 method on List, 311
exists
 method on List, 316
exists2
 method on List object, 323
filter
 method on List, 315
find
 method on List, 315
flatMap
 method on List, 314
flatten
 method on List object, 322
fold
 left, 317
 right, 318
forall
 method on List, 316
forall2
 method on List object, 323
foreach
 method on List, 315
formatted, 399
functional programming, 29
functions
 partially applied, 164, 313
Haller, Philipp, 531
head
 method on List, 300
indices
 method on List, 308
inference
 semicolon, 89, 90
 type, 324
init
 method on List, 305
insertion sort, 301
isEmpty
 method on List, 300
last
 method on List, 305
length
 method on List, 305
linearization
 of traits, 238
make
 method on List, 322
map
 method on List, 313
map2
 method on List object, 323
merge sort, 311
mkString
 method on List, 309

Nil, 299
object-oriented programming, 29
parameters
 repeated, 171
partially applied function, 164,
 313
partition
 method on List, 315
placeholder syntax
 for existentials, 516
 for function literals, 163

range
 as List factory, 321
Raymond, Eric, 31
result type, 53
return type, 53
reverse
 method on List, 306

semicolon
 inference, 89
serializable, 467, 511
serialization, 467
sort
 insertion, 301
 merge, 311
span
 method on List, 316
splitAt
 method on List, 307
Steele, Guy, 31
switch, 153

tail
 method on List, 300
take
 method on List, 307
 method on List, 308
takeWhile
 method on List, 316
throws, 511
toArray
 method on List, 310
toList
 method on Array, 310
traits, 226
transient, 467
type
 inference, 324
 result, 53
 return, 53
type annotation, 30, 44, 50
 for debugging, 327

unary_+, 134
unchecked
 selector annotation, 284
unzip
 method on List object, 322

varargs, 171
view bound, 418
volatile, 466, 511

zip
 method on List, 309