

SCALA

for the Impatient



Cay S. Horstmann

Foreword by Martin Odersky

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

Operators

- Infix notation $x \text{ op } y$ is $x.op(y)$, postfix notation $x \text{ op}$ is $x.op()$
- Only $+ - ! \sim$ can be prefix—define method `unary_op`
- Assignment $x \text{ op}= y$ is $x = x \text{ op } y$ unless defined separately
- Precedence depends on *first* character, except for assignments

Highest: Other operator char	<code>*</code>	<code>/</code>	<code>+</code>	<code>:</code>	<code><</code>	<code>!</code>	<code>&</code>	<code>^</code>	<code> </code>	Not operator char	Lowest: Assignments
	<code>%</code>		<code>-</code>		<code>></code>	<code>=</code>					

- Right associative if *last* character is a colon :
- `x(i) = x(j)` is `x.update(i, x.apply(j))`
- There is no `++` or `--` for numbers. Use `x += 1`; `y -= 1`
- Use `x == y` to compare objects—it calls `equals`

Functions

```
def triple(x: Int) = 3 * x // Parameter name: Type
val f = (x: Int) => 3 * x // Anonymous function
(1 to 10).map(3 * _) // Function with anonymous parameter
def greet(x: Int) { // Without =, return type is Unit
  println("Hello, " + x) }
def greet(x: Int, salutation: String = "Hello") { // Default argument
  println(salutation + ", " + x) }
// Call as greet(42), greet(42, "Hi"), greet(salutation = "Hi", x = 42)
def sum(xs: Int*) = { // * denotes varargs
  var r = 0; for (x <- xs) r += x // Semicolon separates statements on same line
  r // No return. Last expression is value of block
}
def sum(xs: Int*): Int = // Return type required for recursive functions
  if (xs.length == 0) 0 else xs.head + sum(xs.tail : _*) // Sequence as varargs
```

for Loops

```
for (i <- 1 to n) println(i) // i iterates through all values in 1 to n
for (i <- 1 to 9; j <- 1 to 9) println(i * 10 + j) // Multiple iterates
for (i <- 1 to 9 if i != 5; j <- 1 to 9 if i != j) println(i * 10 + j) // Guards
for (i <- 1 to 3; from = 4 - i; j <- from to 3) println(i * 10 + j) // Variable
val r = for (i <- 1 to n) yield i * i // r is a sequence 1, 4, 9, ...
for ((x, y) <- pairs) println(x + " " + y) // Destructures pairs and other values with extractors
```

Pattern Matching

```
val x = r match {
  case '0' => ... // Match value
  case ch if someProperty(ch) => ... // Guard
  case e: Employee => ... // Match runtime type
  case (x, y) => ... // Destructures pairs and other values with extractors
  case Some(v) => ... // Case classes have extractors
  case 0 :: tail => ... // Infix notation for extractors yielding a pair
  case _ => ... // Default case
}

try { ... } catch { // Use the same syntax for catch clauses
  case _: MalformedURLException => println("Bad URL")
  case ex: IOException => ex.printStackTrace()
}
```

Scala for the Impatient

This page intentionally left blank

Scala for the Impatient

Cay S. Horstmann

◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Horstmann, Cay S., 1959-

Scala for the impatient / Cay S. Horstmann.

p. cm.

Includes index.

ISBN 978-0-321-77409-5 (pbk. : alk. paper)—ISBN 0-321-77409-4 (pbk. : alk. paper) 1. Scala (Computer program language) 2. Programming languages (Electronic computers) 3. Computer programming. I. Title.

QA76.73.S28H67 2012

005.13'3—dc23

2011052136

Copyright © 2012 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-77409-5

ISBN-10: 0-321-77409-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
Third printing, February 2014

*To my wife, who made writing this book possible,
and to my children, who made it necessary.*

This page intentionally left blank

Contents

Foreword	xvii
Preface	xix
About the Author	xxi

1	THE BASICS A1	1
1.1	The Scala Interpreter	1
1.2	Declaring Values and Variables	3
1.3	Commonly Used Types	4
1.4	Arithmetic and Operator Overloading	5
1.5	Calling Functions and Methods	7
1.6	The <code>apply</code> Method	8
1.7	Scaladoc	8
	Exercises	11
2	CONTROL STRUCTURES AND FUNCTIONS A1	13
2.1	Conditional Expressions	14
2.2	Statement Termination	15
2.3	Block Expressions and Assignments	16

2.4	Input and Output	17
2.5	Loops	18
2.6	Advanced for Loops and for Comprehensions	19
2.7	Functions	20
2.8	Default and Named Arguments L1	21
2.9	Variable Arguments L1	22
2.10	Procedures	23
2.11	Lazy Values L1	23
2.12	Exceptions	24
	Exercises	26

3 WORKING WITH ARRAYS **A1** 29

3.1	Fixed-Length Arrays	29
3.2	Variable-Length Arrays: Array Buffers	30
3.3	Traversing Arrays and Array Buffers	31
3.4	Transforming Arrays	32
3.5	Common Algorithms	34
3.6	Deciphering Scaladoc	35
3.7	Multidimensional Arrays	37
3.8	Interoperating with Java	37
	Exercises	38

4 MAPS AND TUPLES **A1** 41

4.1	Constructing a Map	41
4.2	Accessing Map Values	42
4.3	Updating Map Values	43
4.4	Iterating over Maps	43
4.5	Sorted Maps	44
4.6	Interoperating with Java	44
4.7	Tuples	45
4.8	Zippping	46
	Exercises	46

5	CLASSES A1	49
5.1	Simple Classes and Parameterless Methods	49
5.2	Properties with Getters and Setters	50
5.3	Properties with Only Getters	53
5.4	Object-Private Fields	54
5.5	Bean Properties L1	55
5.6	Auxiliary Constructors	56
5.7	The Primary Constructor	57
5.8	Nested Classes L1	60
	Exercises	63
6	OBJECTS A1	65
6.1	Singletons	65
6.2	Companion Objects	66
6.3	Objects Extending a Class or Trait	67
6.4	The apply Method	67
6.5	Application Objects	68
6.6	Enumerations	69
	Exercises	71
7	PACKAGES AND IMPORTS A1	73
7.1	Packages	74
7.2	Scope Rules	75
7.3	Chained Package Clauses	77
7.4	Top-of-File Notation	77
7.5	Package Objects	78
7.6	Package Visibility	78
7.7	Imports	79
7.8	Imports Can Be Anywhere	80
7.9	Renaming and Hiding Members	80
7.10	Implicit Imports	80
	Exercises	81

- 8 INHERITANCE **A1** 85
 - 8.1 Extending a Class 85
 - 8.2 Overriding Methods 86
 - 8.3 Type Checks and Casts 87
 - 8.4 Protected Fields and Methods 88
 - 8.5 Superclass Construction 88
 - 8.6 Overriding Fields 89
 - 8.7 Anonymous Subclasses 91
 - 8.8 Abstract Classes 91
 - 8.9 Abstract Fields 91
 - 8.10 Construction Order and Early Definitions **L3** 92
 - 8.11 The Scala Inheritance Hierarchy 94
 - 8.12 Object Equality **L1** 95
 - Exercises 96

- 9 FILES AND REGULAR EXPRESSIONS **A1** 99
 - 9.1 Reading Lines 100
 - 9.2 Reading Characters 100
 - 9.3 Reading Tokens and Numbers 101
 - 9.4 Reading from URLs and Other Sources 102
 - 9.5 Reading Binary Files 102
 - 9.6 Writing Text Files 102
 - 9.7 Visiting Directories 103
 - 9.8 Serialization 104
 - 9.9 Process Control **A2** 105
 - 9.10 Regular Expressions 106
 - 9.11 Regular Expression Groups 107
 - Exercises 107

- 10 TRAITS **L1** 111
 - 10.1 Why No Multiple Inheritance? 111
 - 10.2 Traits as Interfaces 113
 - 10.3 Traits with Concrete Implementations 114
 - 10.4 Objects with Traits 115

10.5	Layered Traits	116
10.6	Overriding Abstract Methods in Traits	117
10.7	Traits for Rich Interfaces	118
10.8	Concrete Fields in Traits	118
10.9	Abstract Fields in Traits	119
10.10	Trait Construction Order	120
10.11	Initializing Trait Fields	122
10.12	Traits Extending Classes	123
10.13	Self Types L2	124
10.14	What Happens under the Hood	125
	Exercises	127

11 OPERATORS **L1** 131

11.1	Identifiers	131
11.2	Infix Operators	132
11.3	Unary Operators	133
11.4	Assignment Operators	133
11.5	Precedence	134
11.6	Associativity	135
11.7	The <code>apply</code> and <code>update</code> Methods	135
11.8	Extractors L2	136
11.9	Extractors with One or No Arguments L2	138
11.10	The <code>unapplySeq</code> Method L2	138
	Exercises	139

12 HIGHER-ORDER FUNCTIONS **L1** 143

12.1	Functions as Values	143
12.2	Anonymous Functions	144
12.3	Functions with Function Parameters	145
12.4	Parameter Inference	146
12.5	Useful Higher-Order Functions	146
12.6	Closures	148
12.7	SAM Conversions	149
12.8	Currying	149

- 12.9 Control Abstractions 150
- 12.10 The return Expression 152
- Exercises 152

13 COLLECTIONS **A2** 155

- 13.1 The Main Collections Traits 156
- 13.2 Mutable and Immutable Collections 157
- 13.3 Sequences 158
- 13.4 Lists 159
- 13.5 Mutable Lists 160
- 13.6 Sets 161
- 13.7 Operators for Adding or Removing Elements 162
- 13.8 Common Methods 164
- 13.9 Mapping a Function 167
- 13.10 Reducing, Folding, and Scanning **A3** 168
- 13.11 Zipping 171
- 13.12 Iterators 172
- 13.13 Streams **A3** 173
- 13.14 Lazy Views 174
- 13.15 Interoperability with Java Collections 175
- 13.16 Threadsafe Collections 177
- 13.17 Parallel Collections 178
- Exercises 179

14 PATTERN MATCHING AND CASE CLASSES **A2** 183

- 14.1 A Better Switch 184
- 14.2 Guards 185
- 14.3 Variables in Patterns 185
- 14.4 Type Patterns 186
- 14.5 Matching Arrays, Lists, and Tuples 187
- 14.6 Extractors 188
- 14.7 Patterns in Variable Declarations 188
- 14.8 Patterns in for Expressions 189
- 14.9 Case Classes 189

- 14.10 The copy Method and Named Parameters 190
- 14.11 Infix Notation in case Clauses 191
- 14.12 Matching Nested Structures 192
- 14.13 Are Case Classes Evil? 192
- 14.14 Sealed Classes 193
- 14.15 Simulating Enumerations 194
- 14.16 The Option Type 194
- 14.17 Partial Functions **L2** 195
 - Exercises 196

15 ANNOTATIONS **A2** 199

- 15.1 What Are Annotations? 200
- 15.2 What Can Be Annotated? 200
- 15.3 Annotation Arguments 201
- 15.4 Annotation Implementations 202
- 15.5 Annotations for Java Features 203
 - 15.5.1 Java Modifiers 203
 - 15.5.2 Marker Interfaces 204
 - 15.5.3 Checked Exceptions 204
 - 15.5.4 Variable Arguments 205
 - 15.5.5 JavaBeans 205
- 15.6 Annotations for Optimizations 206
 - 15.6.1 Tail Recursion 206
 - 15.6.2 Jump Table Generation and Inlining 207
 - 15.6.3 Eliding Methods 208
 - 15.6.4 Specialization for Primitive Types 209
- 15.7 Annotations for Errors and Warnings 210
 - Exercises 211

16 XML PROCESSING **A2** 213

- 16.1 XML Literals 214
- 16.2 XML Nodes 214
- 16.3 Element Attributes 216
- 16.4 Embedded Expressions 217

- 16.5 Expressions in Attributes 218
- 16.6 Uncommon Node Types 219
- 16.7 XPath-like Expressions 220
- 16.8 Pattern Matching 221
- 16.9 Modifying Elements and Attributes 222
- 16.10 Transforming XML 223
- 16.11 Loading and Saving 223
- 16.12 Namespaces 226
 - Exercises 227

17 TYPE PARAMETERS **L2** 231

- 17.1 Generic Classes 232
- 17.2 Generic Functions 232
- 17.3 Bounds for Type Variables 232
- 17.4 View Bounds 234
- 17.5 Context Bounds 234
- 17.6 The Manifest Context Bound 235
- 17.7 Multiple Bounds 235
- 17.8 Type Constraints **L3** 236
- 17.9 Variance 237
- 17.10 Co- and Contravariant Positions 238
- 17.11 Objects Can't Be Generic 240
- 17.12 Wildcards 241
 - Exercises 241

18 ADVANCED TYPES **L2** 245

- 18.1 Singleton Types 246
- 18.2 Type Projections 247
- 18.3 Paths 248
- 18.4 Type Aliases 249
- 18.5 Structural Types 250
- 18.6 Compound Types 250
- 18.7 Infix Types 251
- 18.8 Existential Types 252

18.9	The Scala Type System	253
18.10	Self Types	254
18.11	Dependency Injection	255
18.12	Abstract Types L3	257
18.13	Family Polymorphism L3	259
18.14	Higher-Kinded Types L3	263
	Exercises	265
19	PARSING A3	269
19.1	Grammars	270
19.2	Combining Parser Operations	271
19.3	Transforming Parser Results	273
19.4	Discarding Tokens	274
19.5	Generating Parse Trees	275
19.6	Avoiding Left Recursion	276
19.7	More Combinators	277
19.8	Avoiding Backtracking	280
19.9	Packrat Parsers	280
19.10	What Exactly Are Parsers?	281
19.11	Regex Parsers	282
19.12	Token-Based Parsers	283
19.13	Error Handling	285
	Exercises	286
20	ACTORS A3	289
20.1	Creating and Starting Actors	290
20.2	Sending Messages	291
20.3	Receiving Messages	292
20.4	Sending Messages to Other Actors	293
20.5	Channels	294
20.6	Synchronous Messages and Futures	295
20.7	Thread Sharing	296
20.8	The Actor Life Cycle	299
20.9	Linking Actors	300

20.10	Designing with Actors	301
	Exercises	302
21	IMPLICITS L3	305
21.1	Implicit Conversions	306
21.2	Using Implicits for Enriching Existing Libraries	306
21.3	Importing Implicits	307
21.4	Rules for Implicit Conversions	308
21.5	Implicit Parameters	309
21.6	Implicit Conversions with Implicit Parameters	310
21.7	Context Bounds	311
21.8	Evidence	312
21.9	The <code>@implicitNotFound</code> Annotation	313
21.10	<code>CanBuildFrom</code> Demystified	314
	Exercises	316
22	DELIMITED CONTINUATIONS L3	319
22.1	Capturing and Invoking a Continuation	320
22.2	The “Computation with a Hole”	321
22.3	The Control Flow of <code>reset</code> and <code>shift</code>	322
22.4	The Value of a <code>reset</code> Expression	323
22.5	The Types of <code>reset</code> and <code>shift</code> Expressions	323
22.6	CPS Annotations	325
22.7	Turning a Recursive Visit into an Iteration	326
22.8	Undoing Inversion of Control	329
22.9	The CPS Transformation	332
22.10	Transforming Nested Control Contexts	334
	Exercises	336
	Index	339

Foreword

When I met Cay Horstmann some years ago he told me that Scala needed a better introductory book. My own book had come out a little bit earlier, so of course I had to ask him what he thought was wrong with it. He responded that it was great but too long; his students would not have the patience to read through the eight hundred pages of *Programming in Scala*. I conceded that he had a point. And he set out to correct the situation by writing *Scala for the Impatient*.

I am very happy that his book has finally arrived because it really delivers on what the title says. It gives an eminently practical introduction to Scala, explains what's particular about it, how it differs from Java, how to overcome some common hurdles to learning it, and how to write good Scala code.

Scala is a highly expressive and flexible language. It lets library writers use highly sophisticated abstractions, so that library users can express themselves simply and intuitively. Therefore, depending on what kind of code you look at, it might seem very simple or very complex.

A year ago, I tried to provide some clarification by defining a set of levels for Scala and its standard library. There were three levels each for application programmers and for library designers. The junior levels could be learned quickly and would be sufficient to program productively. Intermediate levels would make programs more concise and more functional and would make libraries

more flexible to use. The highest levels were for experts solving specialized tasks. At the time I wrote:

I hope this will help newcomers to the language decide in what order to pick subjects to learn, and that it will give some advice to teachers and book authors in what order to present the material.

Cay's book is the first to have systematically applied this idea. Every chapter is tagged with a level that tells you how easy or hard it is and whether it's oriented towards library writers or application programmers.

As you would expect, the first chapters give a fast-paced introduction to the basic Scala capabilities. But the book does not stop there. It also covers many of the more "senior" concepts and finally progresses to very advanced material which is not commonly covered in a language introduction, such as how to write parser combinators or make use of delimited continuations. The level tags serve as a guideline for what to pick up when. And Cay manages admirably to make even the most advanced concepts simple to understand.

I liked the concept of *Scala for the Impatient* so much that I asked Cay and his editor, Greg Doench, whether we could get the first part of the book as a free download on the Typesafe web site. They have gracefully agreed to my request, and I would like to thank them for that. That way, everybody can quickly access what I believe is currently the best compact introduction to Scala.

Martin Odersky

January 2012

Preface

The evolution of Java and C++ has slowed down considerably, and programmers who are eager to use more modern language features are looking elsewhere. Scala is an attractive choice; in fact, I think it is by far the most attractive choice for programmers who want to move beyond Java or C++. Scala has a concise syntax that is refreshing after the Java boilerplate. It runs on the Java virtual machine, providing access to a huge set of libraries and tools. It embraces the functional programming style without abandoning object orientation, giving you an incremental learning path to a new paradigm. The Scala interpreter lets you run quick experiments, which makes learning Scala very enjoyable. Last but not least, Scala is statically typed, enabling the compiler to find errors, so that you don't waste time finding them—or not—later in the running program.

I wrote this book for *impatient* readers who want to start programming in Scala right away. I assume you know Java, C#, or C++, and I don't bore you with explaining variables, loops, or classes. I don't exhaustively list all the features of the language, I don't lecture you about the superiority of one paradigm over another, and I don't make you suffer through long and contrived examples. Instead, you will get the information that you need in compact chunks that you can read and review as needed.

Scala is a big language, but you can use it effectively without knowing all of its details intimately. Martin Odersky, the creator of Scala, has identified levels of

expertise for application programmers and library designers—as shown in the following table.

Application Programmer	Library Designer	Overall Scala Level
Beginning A1		Beginning
Intermediate A2	Junior L1	Intermediate
Expert A3	Senior L2	Advanced
	Expert L3	Expert

For each chapter (and occasionally for individual sections), I indicate the experience level required. The chapters progress through levels **A1**, **L1**, **A2**, **L2**, **A3**, **L3**. Even if you don't want to design your own libraries, knowing about the tools that Scala provides for library designers can make you a more effective library user.

I hope you enjoy learning Scala with this book. If you find errors or have suggestions for improvement, please visit <http://horstmann.com/scala> and leave a comment. On that page, you will also find a link to an archive file containing all code examples from the book.

I am very grateful to Dmitry Kirsanov and Alina Kirsanova who turned my manuscript from XHTML into a beautiful book, allowing me to concentrate on the content instead of fussing with the format. Every author should have it so good!

Reviewers include Adrian Cumiskey, Mike Davis, Rob Dickens, Daniel Sobral, Craig Tataryn, David Walend, and William Wheeler. Thanks so much for your comments and suggestions!

Finally, as always, my gratitude goes to my editor, Greg Doench, for encouraging me to write this book, and for his insights during the development process.

Cay Horstmann

San Francisco, 2012

About the Author

Cay S. Horstmann is principal author of *Core Java™, Volumes I & II, Eighth Edition* (Sun Microsystems Press, 2008), as well as a dozen other books for professional programmers and computer science students. He is a professor of computer science at San Jose State University and a Java Champion.

The Basics

Topics in This Chapter **A1**

- 1.1 The Scala Interpreter — page 1
- 1.2 Declaring Values and Variables — page 3
- 1.3 Commonly Used Types — page 4
- 1.4 Arithmetic and Operator Overloading — page 5
- 1.5 Calling Functions and Methods — page 7
- 1.6 The `apply` Method — page 8
- 1.7 Scaladoc — page 8
- Exercises — page 11

Chapter

1

In this chapter, you will learn how to use Scala as an industrial-strength pocket calculator, working interactively with numbers and arithmetic operations. We introduce a number of important Scala concepts and idioms along the way. You will also learn how to browse the Scaladoc documentation at a beginner's level.

Highlights of this introduction are:

- Using the Scala interpreter
- Defining variables with `var` and `val`
- Numeric types
- Using operators and functions
- Navigating Scaladoc

1.1 The Scala Interpreter

To start the Scala interpreter:

- Install Scala.
- Make sure that the `scala/bin` directory is on the `PATH`.
- Open a command shell in your operating system.
- Type `scala` followed by the Enter key.



TIP: Don't like the command shell? There are other ways of running the interpreter—see <http://horstmann.com/scala/install>.

Now type commands followed by Enter. Each time, the interpreter displays the answer. For example, if you type `8 * 5 + 2` (as shown in boldface below), you get 42.

```
scala> 8 * 5 + 2  
res0: Int = 42
```

The answer is given the name `res0`. You can use that name in subsequent computations:

```
scala> 0.5 * res0  
res1: Double = 21.0  
scala> "Hello, " + res0  
res2: java.lang.String = Hello, 42
```

As you can see, the interpreter also displays the type of the result—in our examples, `Int`, `Double`, and `java.lang.String`.

You can call methods. Depending on how you launched the interpreter, you may be able to use *tab completion* for method names. Try typing `res2.to` and then hit the Tab key. If the interpreter offers choices such as

```
toCharArray toLowerCase toString toUpperCase
```

this means tab completion works. Type a `U` and hit the Tab key again. You now get a single completion:

```
res2.toUpperCase
```

Hit the Enter key, and the answer is displayed. (If you can't use tab completion in your environment, you'll have to type the complete method name yourself.)

Also try hitting the `↑` and `↓` arrow keys. In most implementations, you will see the previously issued commands, and you can edit them. Use the `←`, `→`, and `Del` keys to change the last command to

```
res2.toLowerCase
```

As you can see, the Scala interpreter reads an expression, evaluates it, prints it, and reads the next expression. This is called the *read-eval-print loop*, or REPL.

Technically speaking, the `scala` program is *not* an interpreter. Behind the scenes, your input is quickly compiled into bytecode, and the bytecode is executed by

the Java virtual machine. For that reason, most Scala programmers prefer to call it “the REPL”.



TIP: The REPL is your friend. Instant feedback encourages experimenting, and you will feel good whenever something works.

It is a good idea to keep an editor window open at the same time, so you can copy and paste successful code snippets for later use. Also, as you try more complex examples, you may want to compose them in the editor and then paste them into the REPL.

1.2 Declaring Values and Variables

Instead of using the names `res0`, `res1`, and so on, you can define your own names:

```
scala> val answer = 8 * 5 + 2
answer: Int = 42
```

You can use these names in subsequent expressions:

```
scala> 0.5 * answer
res3: Double = 21.0
```

A value declared with `val` is actually a constant—you can’t change its contents:

```
scala> answer = 0
<console>:6: error: reassignment to val
```

To declare a variable whose contents can vary, use a `var`:

```
var counter = 0
counter = 1 // OK, can change a var
```

In Scala, you are encouraged to use a `val` unless you really need to change the contents. Perhaps surprisingly for Java or C++ programmers, most programs don’t need many `var` variables.

Note that you need not specify the type of a value or variable. It is inferred from the type of the expression with which you initialize it. (It is an error to declare a value or variable without initializing it.)

However, you can specify the type if necessary. For example,

```
val greeting: String = null
val greeting: Any = "Hello"
```



NOTE: In Scala, the type of a variable or function is always written *after* the name of the variable or function. This makes it easier to read declarations with complex types.

As I move back and forth between Scala and Java, I find that my fingers write Java declarations such as `String greeting` on autopilot, so I have to rewrite them as `greeting: String`. This is a bit annoying, but when I work with complex Scala programs, I really appreciate that I don't have to decrypt C-style type declarations.



NOTE: You may have noticed that there were no semicolons after variable declarations or assignments. In Scala, semicolons are only required if you have multiple statements on the same line.

You can declare multiple values or variables together:

```
val xmax, ymax = 100 // Sets xmax and ymax to 100
var greeting, message: String = null
// greeting and message are both strings, initialized with null
```

1.3 Commonly Used Types

You have already seen some of the data types of the Scala language, such as `Int` and `Double`. Like Java, Scala has seven numeric types: `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, and `Double`, and a `Boolean` type. However, unlike Java, these types are *classes*. There is no distinction between primitive types and class types in Scala. You can invoke methods on numbers, for example:

```
1.toString() // Yields the string "1"
```

or, more excitingly,

```
1.to(10) // Yields Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

(We will discuss the `Range` class in Chapter 13. For now, just view it as a collection of numbers.)

In Scala, there is no need for wrapper types. It is the job of the Scala compiler to convert between primitive types and wrappers. For example, if you make an array of `Int`, you get an `int[]` array in the virtual machine.

As you saw in Section 1.1, “The Scala Interpreter,” on page 1, Scala relies on the underlying `java.lang.String` class for strings. However, it augments that class with well over a hundred operations in the `StringOps` class.

For example, the `intersect` method yields the characters that are common to two strings:

```
"Hello".intersect("World") // Yields "lo"
```

In this expression, the `java.lang.String` object `"Hello"` is implicitly converted to a `StringOps` object, and then the `intersect` method of the `StringOps` class is applied.

Therefore, remember to look into the `StringOps` class when you use the Scala documentation (see Section 1.7, “*Scaladoc*,” on page 8).

Similarly, there are classes `RichInt`, `RichDouble`, `RichChar`, and so on. Each of them has a small set of convenience methods for acting on their poor cousins—`Int`, `Double`, or `Char`. The `to` method that you saw above is actually a method of the `RichInt` class. In the expression

```
1.to(10)
```

the `Int` value `1` is first converted to a `RichInt`, and the `to` method is applied to that value.

Finally, there are classes `BigInt` and `BigDecimal` for computations with an arbitrary (but finite) number of digits. These are backed by the `java.math.BigInteger` and `java.math.BigDecimal` classes, but, as you will see in the next section, they are much more convenient because you can use them with the usual mathematical operators.



NOTE: In Scala, you use methods, not casts, to convert between numeric types. For example, `99.44.toInt` is `99`, and `99.toChar` is `'c'`. Of course, as in Java, the `toString` method converts any object to a string.

To convert a string containing a number into the number, use `toInt` or `toDouble`. For example, `"99.44".toDouble` is `99.44`.

1.4 Arithmetic and Operator Overloading

Arithmetic operators in Scala work just as you would expect in Java or C++:

```
val answer = 8 * 5 + 2
```

The `+` `-` `*` `/` `%` operators do their usual job, as do the bit operators `&` `|` `^` `>>` `<<`. There is just one surprising aspect: These operators are actually methods. For example,

```
a + b
```

is a shorthand for

```
a.+(b)
```

Here, `+` is the name of the method. Scala has no silly prejudice against non-alphanumeric characters in method names. You can define methods with just about any symbols for names. For example, the `BigInt` class defines a method called `/%` that returns a pair containing the quotient and remainder of a division.

In general, you can write

```
a method b
```

as a shorthand for

```
a.method(b)
```

where `method` is a method with two parameters (one implicit, one explicit). For example, instead of

```
1.to(10)
```

you can write

```
1 to 10
```

Use whatever you think is easier to read. Beginning Scala programmers tend to stick to the Java syntax, and that is just fine. Of course, even the most hardened Java programmers seem to prefer `a + b` over `a.+(b)`.

There is one notable difference between Scala and Java or C++. Scala does not have `++` or `--` operators. Instead, simply use `+=1` or `-=1`:

```
counter+=1 // Increments counter—Scala has no ++
```

Some people wonder if there is any deep reason for Scala's refusal to provide a `++` operator. (Note that you can't simply implement a method called `++`. Since the `Int` class is immutable, such a method cannot change an integer value.) The Scala designers decided it wasn't worth having yet another special rule just to save one keystroke.

You can use the usual mathematical operators with `BigInt` and `BigDecimal` objects:

```
val x: BigInt = 1234567890
x * x * x // Yields 1881676371789154860897069000
```

That's much better than Java, where you would have had to call `x.multiply(x).multiply(x)`.



NOTE: In Java, you cannot overload operators, and the Java designers claimed this is a good thing because it stops you from inventing crazy operators like `!@$&*:` that would make your program impossible to read. Of course, that's silly; you can make your programs just as hard to read by using crazy method names like `qxyz`. Scala allows you to define operators, leaving it up to you to use this feature with restraint and good taste.

1.5 Calling Functions and Methods

Scala has functions in addition to methods. It is simpler to use mathematical functions such as `min` or `pow` in Scala than in Java—you need not call static methods from a class.

```
sqrt(2) // Yields 1.4142135623730951
pow(2, 4) // Yields 16.0
min(3, Pi) // Yields 3.0
```

The mathematical functions are defined in the `scala.math` package. You can import them with the statement

```
import scala.math._ // In Scala, the _ character is a “wildcard,” like * in Java
```



NOTE: To use a package that starts with `scala.`, you can omit the `scala` prefix. For example, `import math._` is equivalent to `import scala.math._`, and `math.sqrt(2)` is the same as `scala.math.sqrt(2)`.

We discuss the `import` statement in more detail in Chapter 7. For now, just use `import packageName._` whenever you need to import a particular package.

Scala doesn't have static methods, but it has a similar feature, called *singleton objects*, which we will discuss in detail in Chapter 6. Often, a class has a *companion object* whose methods act just like static methods do in Java. For example, the `BigInt` companion object to the `BigInt` class has a method `probablePrime` that generates a random prime number with a given number of bits:

```
BigInt.probablePrime(100, scala.util.Random)
```

Try this in the REPL; you'll get a number such as `1039447980491200275486540240713`. Note that the call `BigInt.probablePrime` is similar to a static method call in Java.



NOTE: Here, `Random` is a singleton random number generator object, defined in the `scala.util` package. This is one of the few situations where a singleton object is better than a class. In Java, it is a common error to construct a new `java.util.Random` object for each random number.

Scala methods without parameters often don't use parentheses. For example, the API of the `StringOps` class shows a method `distinct`, without `()`, to get the distinct letters in a string. You call it as

```
"Hello".distinct
```

The rule of thumb is that a parameterless method that doesn't modify the object has no parentheses. We discuss this further in Chapter 5.

1.6 The apply Method

In Scala, it is common to use a syntax that looks like a function call. For example, if `s` is a string, then `s(i)` is the *i*th character of the string. (In C++, you would write `s[i]`; in Java, `s.charAt(i)`.) Try it out in the REPL:

```
"Hello"(4) // Yields 'o'
```

You can think of this as an overloaded form of the `()` operator. It is implemented as a method with the name `apply`. For example, in the documentation of the `StringOps` class, you will find a method

```
def apply(n: Int): Char
```

That is, `"Hello"(4)` is a shortcut for

```
"Hello".apply(4)
```

When you look at the documentation for the `BigInt` companion object, you will see `apply` methods that let you convert strings or numbers to `BigInt` objects. For example, the call

```
BigInt("1234567890")
```

is a shortcut for

```
BigInt.apply("1234567890")
```

It yields a new `BigInt` object, *without having to use new*. For example:

```
BigInt("1234567890") * BigInt("112358111321")
```

Using the `apply` method of a companion object is a common Scala idiom for constructing objects. For example, `Array(1, 4, 9, 16)` returns an array, thanks to the `apply` method of the `Array` companion object.

1.7 Scaladoc

Java programmers use Javadoc to navigate the Java API. Scala has its own variant, called Scaladoc (see Figure 1–1).

Navigating Scaladoc is a bit more challenging than Javadoc. Scala classes tend to have many more convenience methods than Java classes. Some methods use features that you haven't learned yet. Finally, some features are exposed as they are implemented, not as they are used. (The Scala team is working on improving the Scaladoc presentation, so that it can be more approachable to beginners in the future.)

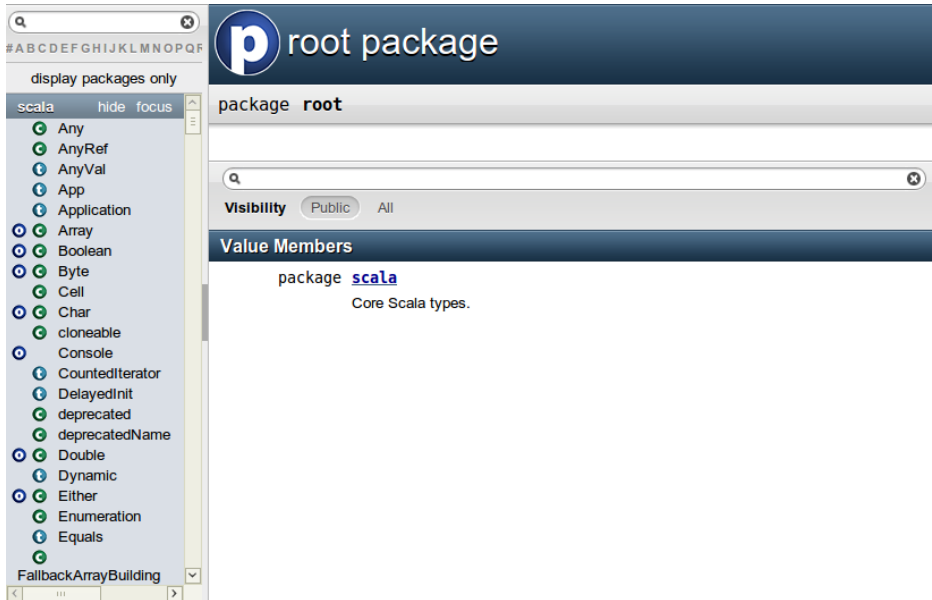


Figure 1–1 The entry page for Scaladoc

Here are some tips for navigating Scaladoc, for a newcomer to the language.

You can browse Scaladoc online at www.scala-lang.org/api, but it is a good idea to download a copy from www.scala-lang.org/downloads#api and install it locally.

Unlike Javadoc, which presents an alphabetical listing of classes, Scaladoc's class list is sorted by packages. If you know the class name but not the package name, use the filter in the top left corner (see Figure 1–2).

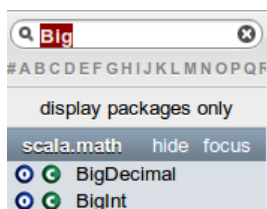


Figure 1–2 The filter box in Scaladoc

Click on the X symbol to clear the filter.

Note the O and C symbols next to each class name. They let you navigate to the class (C) or the companion object (O).

Scaladoc can be a bit overwhelming. Keep these tips in mind.

- Remember to look into `RichInt`, `RichDouble`, and so on, if you want to know how to work with numeric types. Similarly, to work with strings, look into `StringOps`.
- The mathematical functions are in the *package* `scala.math`, not in any class.
- Sometimes, you'll see functions with funny names. For example, `BigInt` has a method `unary_-`. As you will see in Chapter 11, this is how you define the prefix negation operator `-x`.
- A method tagged as `implicit` is an automatic conversion. For example, the `BigInt` object has conversions from `int` and `long` to `BigInt` that are automatically called when needed. See Chapter 21 for more information about implicit conversions.
- Methods can have functions as parameters. For example, the `count` method in `StringOps` requires a function that returns `true` or `false` for a `Char`, specifying which characters should be counted:

```
def count(p: (Char) => Boolean) : Int
```

You supply a function, often in a very compact notation, when you call the method. As an example, the call `s.count(_.isUpper)` counts the number of uppercase characters. We will discuss this style of programming in much more detail in Chapter 12.

- You'll occasionally run into classes such as `Range` or `Seq[Char]`. They mean what your intuition tells you—a range of numbers, a sequence of characters. You will learn all about these classes as you delve more deeply into Scala.
- Don't get discouraged that there are so many methods. It's the Scala way to provide lots of methods for every conceivable use case. When you need to solve a particular problem, just look for a method that is useful. More often than not, there is one that addresses your task, which means you don't have to write so much code yourself.
- Finally, don't worry if you run into the occasional indecipherable incantation, such as this one in the `StringOps` class:

```
def patch [B >: Char, That](from: Int, patch: GenSeq[B], replaced: Int)
  (implicit bf: CanBuildFrom[String, B, That]): That
```

Just ignore it. There is another version of `patch` that looks more reasonable:

```
def patch(from: Int, that: GenSeq[Char], replaced: Int): StringOps[A]
```

If you think of `GenSeq[Char]` and `StringOps[A]` as `String`, the method is pretty easy to understand from the documentation. And it's easy to try it out in the REPL:

```
"Harry".patch(1, "ung", 2) // Yields "Hungry"
```

Exercises

1. In the Scala REPL, type `3.` followed by the Tab key. What methods can be applied?
2. In the Scala REPL, compute the square root of 3, and then square that value. By how much does the result differ from 3? (Hint: The `res` variables are your friend.)
3. Are the `res` variables `val` or `var`?
4. Scala lets you multiply a string with a number—try out `"crazy" * 3` in the REPL. What does this operation do? Where can you find it in Scaladoc?
5. What does `10 max 2` mean? In which class is the `max` method defined?
6. Using `BigInt`, compute 2^{1024} .
7. What do you need to import so that you can get a random prime as `probablePrime(100, Random)`, without any qualifiers before `probablePrime` and `Random`?
8. One way to create random file or directory names is to produce a random `BigInt` and convert it to base 36, yielding a string such as `"qsnvbevtomcj38o06ku1"`. Poke around Scaladoc to find a way of doing this in Scala.
9. How do you get the first character of a string in Scala? The last character?
10. What do the `take`, `drop`, `takeRight`, and `dropRight` string functions do? What advantage or disadvantage do they have over using `substring`?

Control Structures and Functions

Topics in This Chapter **A1**

- 2.1 Conditional Expressions — page 14
- 2.2 Statement Termination — page 15
- 2.3 Block Expressions and Assignments — page 16
- 2.4 Input and Output — page 17
- 2.5 Loops — page 18
- 2.6 Advanced for Loops and for Comprehensions — page 19
- 2.7 Functions — page 20
- 2.8 Default and Named Arguments **L1** — page 21
- 2.9 Variable Arguments **L1** — page 22
- 2.10 Procedures — page 23
- 2.11 Lazy Values **L1** — page 23
- 2.12 Exceptions — page 24
- Exercises — page 26

Chapter

2

In this chapter, you will learn how to implement conditions, loops, and functions in Scala. You will encounter a fundamental difference between Scala and other programming languages. In Java or C++, we differentiate between *expressions* (such as $3 + 4$) and *statements* (for example, an if statement). An expression has a value; a statement carries out an action. In Scala, almost all constructs have values. This feature can make programs more concise and easier to read.

Here are the highlights of this chapter:

- An if expression has a value.
- A block has a value—the value of its last expression.
- The Scala for loop is like an “enhanced” Java for loop.
- Semicolons are (mostly) optional.
- The void type is `Unit`.
- Avoid using `return` in a function.
- Beware of missing `=` in a function definition.
- Exceptions work just like in Java or C++, but you use a “pattern matching” syntax for catch.
- Scala has no checked exceptions.

2.1 Conditional Expressions

Scala has an `if/else` construct with the same syntax as in Java or C++. However, in Scala, an `if/else` has a value, namely the value of the expression that follows the `if` or `else`. For example,

```
if (x > 0) 1 else -1
```

has a value of 1 or -1, depending on the value of `x`. You can put that value in a variable:

```
val s = if (x > 0) 1 else -1
```

This has the same effect as

```
if (x > 0) s = 1 else s = -1
```

However, the first form is better because it can be used to initialize a `val`. In the second form, `s` needs to be a `var`.

(As already mentioned, semicolons are mostly optional in Scala—see Section 2.2, “Statement Termination,” on page 15.)

Java and C++ have a `?:` operator for this purpose. The expression

```
x > 0 ? 1 : -1 // Java or C++
```

is equivalent to the Scala expression `if (x > 0) 1 else -1`. However, you can't put statements inside a `?:` expression. The Scala `if/else` combines the `if/else` and `?:` constructs that are separate in Java and C++.

In Scala, every expression has a type. For example, the expression `if (x > 0) 1 else -1` has the type `Int` because both branches have the type `Int`. The type of a mixed-type expression, such as

```
if (x > 0) "positive" else -1
```

is the common supertype of both branches. In this example, one branch is a `java.lang.String`, and the other an `Int`. Their common supertype is called `Any`. (See Section 8.11, “The Scala Inheritance Hierarchy,” on page 94 for details.)

If the `else` part is omitted, for example in

```
if (x > 0) 1
```

then it is possible that the `if` statement yields no value. However, in Scala, every expression is supposed to have *some* value. This is finessed by introducing a class `Unit` that has one value, written as `()`. The `if` statement without an `else` is equivalent to

```
if (x > 0) 1 else ()
```

Think of `()` as a placeholder for “no useful value,” and think of `Unit` as the analog of `void` in Java or C++.

(Technically speaking, `void` has no value whereas `Unit` has one value that signifies “no value”. If you are so inclined, you can ponder the difference between an empty wallet and a wallet with a bill labeled “no dollars”.)



NOTE: Scala has no `switch` statement, but it has a much more powerful pattern matching mechanism that we will discuss in Chapter 14. For now, just use a sequence of `if` statements.



CAUTION: The REPL is more nearsighted than the compiler—it only sees one line of code at a time. For example, when you type

```
if (x > 0) 1
else if (x == 0) 0 else -1
```

the REPL executes `if (x > 0) 1` and shows the answer. Then it gets confused about the `else` keyword.

If you want to break the line before the `else`, use braces:

```
if (x > 0) { 1
} else if (x == 0) 0 else -1
```

This is only a concern in the REPL. In a compiled program, the parser will find the `else` on the next line.



TIP: If you want to paste a block of code into the REPL without worrying about its nearsightedness, use *paste mode*. Type

```
:paste
```

Then paste in the code block and type `Ctrl+K`. The REPL will then analyze the block in its entirety.

2.2 Statement Termination

In Java and C++, every statement ends with a semicolon. In Scala—like in JavaScript and other scripting languages—a semicolon is never required if it falls just before the end of the line. A semicolon is also optional before an `}`, an `else`, and similar locations where it is clear from context that the end of a statement has been reached.

However, if you want to have more than one statement on a single line, you need to separate them with semicolons. For example,

```
if (n > 0) { r = r * n; n -= 1 }
```

A semicolon is needed to separate `r = r * n` and `n -= 1`. Because of the `}`, no semicolon is needed after the second statement.

If you want to continue a long statement over two lines, you need to make sure that the first line ends in a symbol that *cannot be* the end of a statement. An operator is often a good choice:

```
s = s0 + (v - v0) * t + // The + tells the parser that this is not the end
0.5 * (a - a0) * t * t
```

In practice, long expressions usually involve function or method calls, and then you don't need to worry much—after an opening `(`, the compiler won't infer the end of a statement until it has seen the matching `)`.

In the same spirit, Scala programmers favor the Kernighan & Ritchie brace style:

```
if (n > 0) {
  r = r * n
  n -= 1
}
```

The line ending with a `{` sends a clear signal that there is more to come.

Many programmers coming from Java or C++ are initially uncomfortable about omitting semicolons. If you prefer to have them, just put them in—they do no harm.

2.3 Block Expressions and Assignments

In Java or C++, a block statement is a sequence of statements enclosed in `{ }`. You use a block statement whenever you need to put multiple actions in the body of a branch or loop statement.

In Scala, a `{ }` block contains a sequence of *expressions*, and the result is also an expression. The value of the block is the value of the last expression.

This feature can be useful if the initialization of a `val` takes more than one step. For example,

```
val distance = { val dx = x - x0; val dy = y - y0; sqrt(dx * dx + dy * dy) }
```


The value of the `{ }` block is the last expression, shown here in bold. The variables `dx` and `dy`, which were only needed as intermediate values in the computation, are neatly hidden from the rest of the program.

In Scala, assignments have no value—or, strictly speaking, they have a value of type `Unit`. Recall that the `Unit` type is the equivalent of the `void` type in Java and C++, with a single value written as `()`.

A block that ends with an assignment, such as

```
{ r = r * n; n -= 1 }
```

has a `Unit` value. This is not a problem, just something to be aware of when defining functions—see Section 2.7, “Functions,” on page 20.

Since assignments have `Unit` value, don’t chain them together.

```
x = y = 1 // No
```

The value of `y = 1` is `()`, and it’s highly unlikely that you wanted to assign a `Unit` to `x`. (In contrast, in Java and C++, the value of an assignment is the value that is being assigned. In those languages, chained assignments are useful.)

2.4 Input and Output

To print a value, use the `print` or `println` function. The latter adds a newline character after the printout. For example,

```
print("Answer: ")
println(42)
```

yields the same output as

```
println("Answer: " + 42)
```

There is also a `printf` function with a C-style format string:

```
printf("Hello, %s! You are %d years old.\n", "Fred", 42)
```

You can read a line of input from the console with the `readLine` function. To read a numeric, Boolean, or character value, use `readInt`, `readDouble`, `readByte`, `readShort`, `readLong`, `readFloat`, `readBoolean`, or `readChar`. The `readLine` method, but not the other ones, takes a prompt string:

```
val name = readLine("Your name: ")
print("Your age: ")
val age = readInt()
printf("Hello, %s! Next year, you will be %d.\n", name, age + 1)
```

2.5 Loops

Scala has the same `while` and `do` loops as Java and C++. For example,

```
while (n > 0) {  
  r = r * n  
  n -= 1  
}
```

Scala has no direct analog of the `for` (*initialize; test; update*) loop. If you need such a loop, you have two choices. You can use a `while` loop. Or, you can use a `for` statement like this:

```
for (i <- 1 to n)  
  r = r * i
```

You saw the `to` method of the `RichInt` class in Chapter 1. The call `1 to n` returns a `Range` of the numbers from 1 to `n` (inclusive).

The construct

```
for (i <- expr)
```

makes the variable `i` traverse all values of the expression to the right of the `<-`. Exactly how that traversal works depends on the type of the expression. For a Scala collection, such as a `Range`, the loop makes `i` assume each value in turn.



NOTE: There is no `val` or `var` before the variable in the `for` loop. The type of the variable is the element type of the collection. The scope of the loop variable extends until the end of the loop.

When traversing a string or array, you often need a range from 0 to $n - 1$. In that case, use the `until` method instead of the `to` method. It returns a range that doesn't include the upper bound.

```
val s = "Hello"  
var sum = 0  
for (i <- 0 until s.length) // Last value for i is s.length - 1  
  sum += s(i)
```

In this example, there is actually no need to use indexes. You can directly loop over the characters:

```
var sum = 0  
for (ch <- "Hello") sum += ch
```

In Scala, loops are not used as often as in other languages. As you will see in Chapter 12, you can often process the values in a sequence by applying a function to all of them, which can be done with a single method call.



NOTE: Scala has no `break` or `continue` statements to break out of a loop. What to do if you need a `break`? Here are a few options:

1. Use a Boolean control variable instead.
2. Use nested functions—you can return from the middle of a function.
3. Use the `break` method in the `Breaks` object:

```
import scala.util.control.Breaks._
breakable {
  for (...) {
    if (...) break; // Exits the breakable block
    ...
  }
}
```

Here, the control transfer is done by throwing and catching an exception, so you should avoid this mechanism when time is of the essence.

2.6 Advanced for Loops and for Comprehensions

In the preceding section, you saw the basic form of the `for` loop. However, this construct is much richer in Scala than in Java or C++. This section covers the advanced features.

You can have multiple *generators* of the form `variable <- expression`. Separate them by semicolons. For example,

```
for (i <- 1 to 3; j <- 1 to 3) print((10 * i + j) + " ")
// Prints 11 12 13 21 22 23 31 32 33
```

Each generator can have a *guard*, a Boolean condition preceded by `if`:

```
for (i <- 1 to 3; j <- 1 to 3 if i != j) print((10 * i + j) + " ")
// Prints 12 13 21 23 31 32
```

Note that there is no semicolon before the `if`.

You can have any number of *definitions*, introducing variables that can be used inside the loop:

```
for (i <- 1 to 3; from = 4 - i; j <- from to 3) print((10 * i + j) + " ")
// Prints 13 22 23 31 32 33
```

When the body of the for loop starts with `yield`, then the loop constructs a collection of values, one for each iteration:

```
for (i <- 1 to 10) yield i % 3
// Yields Vector(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)
```

This type of loop is called a *for comprehension*.

The generated collection is compatible with the first generator.

```
for (c <- "Hello"; i <- 0 to 1) yield (c + i).toChar
// Yields "HIeflm1mop"
for (i <- 0 to 1; c <- "Hello") yield (c + i).toChar
// Yields Vector('H', 'e', 'l', 'l', 'o', 'I', 'f', 'm', 'm', 'p')
```



NOTE: If you prefer, you can enclose the generators, guards, and definitions of a for loop inside braces, and you can use newlines instead of semicolons to separate them:

```
for { i <- 1 to 3
    from = 4 - i
    j <- from to 3 }
```

2.7 Functions

Scala has functions in addition to methods. A method operates on an object, but a function doesn't. C++ has functions as well, but in Java, you have to imitate them with static methods.

To define a function, you specify the function's name, parameters, and body like this:

```
def abs(x: Double) = if (x >= 0) x else -x
```

You must specify the types of all parameters. However, as long as the function is not recursive, you need not specify the return type. The Scala compiler determines the return type from the type of the expression to the right of the `=` symbol.

If the body of the function requires more than one expression, use a block. The last expression of the block becomes the value that the function returns. For example, the following function returns the value of `r` after the for loop.

```
def fac(n : Int) = {
  var r = 1
  for (i <- 1 to n) r = r * i
  r
}
```

There is no need for the `return` keyword in this example. It is possible to use `return` as in Java or C++, to exit a function immediately, but that is not commonly done in Scala.



TIP: While there is nothing wrong with using `return` in a named function (except the waste of seven keystrokes), it is a good idea to get used to life without `return`. Pretty soon, you will be using lots of *anonymous functions*, and there, `return` doesn't return a value to the caller. It breaks out to the enclosing named function. Think of `return` as a kind of `break` statement for functions, and only use it when you want that breakout functionality.

With a recursive function, you must specify the return type. For example,

```
def fac(n: Int): Int = if (n <= 0) 1 else n * fac(n - 1)
```

Without the return type, the Scala compiler couldn't verify that the type of `n * fac(n - 1)` is an `Int`.



NOTE: Some programming languages (such as ML and Haskell) *can* infer the type of a recursive function, using the Hindley-Milner algorithm. However, this doesn't work well in an object-oriented language. Extending the Hindley-Milner algorithm so it can handle subtypes is still a research problem.

2.8 Default and Named Arguments

You can provide default arguments for functions that are used when you don't specify explicit values. For example,

```
def decorate(str: String, left: String = "[", right: String = "]") =
  left + str + right
```

This function has two parameters, `left` and `right`, with default arguments `"["` and `"]"`.

If you call `decorate("Hello")`, you get `"[Hello]"`. If you don't like the defaults, supply your own: `decorate("Hello", "<<<", ">>>")`.

If you supply fewer arguments than there are parameters, the defaults are applied from the end. For example, `decorate("Hello", ">>>[")` uses the default value of the `right` parameter, yielding `>>>[Hello]"`.

You can also specify the parameter names when you supply the arguments. For example,

```
decorate(left = "<<<", str = "Hello", right = ">>>")
```

The result is "<<<Hello>>>". Note that the named arguments need not be in the same order as the parameters.

Named arguments can make a function call more readable. They are also useful if a function has many default parameters.

You can mix unnamed and named arguments, provided the unnamed ones come first:

```
decorate("Hello", right = "]"<<<") // Calls decorate("Hello", "[", "]"<<<")
```

2.9 Variable Arguments

Sometimes, it is convenient to implement a function that can take a variable number of arguments. The following example shows the syntax:

```
def sum(args: Int*) = {
  var result = 0
  for (arg <- args) result += arg
  result
}
```

You can call this function with as many arguments as you like.

```
val s = sum(1, 4, 9, 16, 25)
```

The function receives a single parameter of type `Seq`, which we will discuss in Chapter 13. For now, all you need to know is that you can use a `for` loop to visit each element.

If you already have a sequence of values, you cannot pass it directly to such a function. For example, the following is not correct:

```
val s = sum(1 to 5) // Error
```

If the `sum` function is called with one argument, that must be a single integer, not a range of integers. The remedy is to tell the compiler that you want the parameter to be considered an argument sequence. Append `: _*`, like this:

```
val s = sum(1 to 5: _*) // Consider 1 to 5 as an argument sequence
```

This call syntax is needed in a recursive definition:

```
def recursiveSum(args: Int*) : Int = {
  if (args.length == 0) 0
  else args.head + recursiveSum(args.tail : _*)
}
```

Here, the `head` of a sequence is its initial element, and `tail` is a sequence of all other elements. That's again a `Seq`, and we have to use `: _*` to convert it to an argument sequence.



CAUTION: When you call a Java method with variable arguments of type Object, such as `PrintStream.printf` or `MessageFormat.format`, you need to convert any primitive types by hand. For example,

```
val str = MessageFormat.format("The answer to {0} is {1}",
    "everything", 42.asInstanceOf[AnyRef])
```

This is the case for any Object parameter, but I mention it here because it is most common with varargs methods.

2.10 Procedures

Scala has a special notation for a function that returns no value. If the function body is enclosed in braces *without a preceding = symbol*, then the return type is `Unit`. Such a function is called a *procedure*. A procedure returns no value, and you only call it for its side effect. For example, the following procedure prints a string inside a box, like

```
-----
|Hello|
-----
```

Because the procedure doesn't return any value, we omit the = symbol.

```
def box(s : String) { // Look carefully: no =
  val border = "-" * s.length + "--\n"
  println(border + "|" + s + "|\n" + border)
}
```

Some people (not me) dislike this concise syntax for procedures and suggest that you always use an explicit return type of `Unit`:

```
def box(s : String): Unit = {
  ...
}
```



CAUTION: The concise procedure syntax can be a surprise for Java and C++ programmers. It is a common error to accidentally omit the = in a function definition. You then get an error message at the point where the function is called, and you are told that `Unit` is not acceptable at that location.

2.11 Lazy Values **L1**

When a `val` is declared as lazy, its initialization is deferred until it is accessed for the first time. For example,

```
lazy val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
```

(We will discuss file operations in Chapter 9. For now, just take it for granted that this call reads all characters from a file into a string.)

If the program never accesses `words`, the file is never opened. To verify this, try it out in the REPL, but misspell the file name. There will be no error when the initialization statement is executed. However, when you access `words`, you will get an error message that the file is not found.

Lazy values are useful to delay costly initialization statements. They can also deal with other initialization issues, such as circular dependencies. Moreover, they are essential for developing lazy data structures—see Section 13.13, “Streams,” on page 173.

You can think of lazy values as halfway between `val` and `def`. Compare

```
val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
// Evaluated as soon as words is defined
lazy val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
// Evaluated the first time words is used
def words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
// Evaluated every time words is used
```



NOTE: Laziness is not cost-free. Every time a lazy value is accessed, a method is called that checks, in a threadsafe manner, whether the value has already been initialized.

2.12 Exceptions

Scala exceptions work the same way as in Java or C++. When you throw an exception, for example

```
throw new IllegalArgumentException("x should not be negative")
```

the current computation is aborted, and the runtime system looks for an exception handler that can accept an `IllegalArgumentException`. Control resumes with the innermost such handler. If no such handler exists, the program terminates.

As in Java, the objects that you throw need to belong to a subclass of `java.lang.Throwable`. However, unlike Java, Scala has no “checked” exceptions—you never have to declare that a function or method might throw an exception.



NOTE: In Java, “checked” exceptions are checked at compile time. If your method might throw an `IOException`, you must declare it. This forces programmers to think where those exceptions should be handled, which is a laudable goal. Unfortunately, it can also give rise to monstrous method signatures such as `void doSomething() throws IOException, InterruptedException, ClassNotFoundException`. Many Java programmers detest this feature and end up defeating it by either catching exceptions too early or using excessively general exception classes. The Scala designers decided against checked exceptions, recognizing that thorough compile-time checking isn’t *always* a good thing.

A throw expression has the special type `Nothing`. That is useful in `if/else` expressions. If one branch has type `Nothing`, the type of the `if/else` expression is the type of the other branch. For example, consider

```
if (x >= 0) { sqrt(x)
} else throw new IllegalArgumentException("x should not be negative")
```

The first branch has type `Double`, the second has type `Nothing`. Therefore, the `if/else` expression also has type `Double`.

The syntax for catching exceptions is modeled after the pattern matching syntax (see Chapter 14).

```
val url = new URL("http://horstmann.com/fred-tiny.gif")
try {
  process(url)
} catch {
  case _: MalformedURLException => println("Bad URL: " + url)
  case ex: IOException => ex.printStackTrace()
}
```

As in Java or C++, the more general exception types should come after the more specific ones.

Note that you can use `_` for the variable name if you don’t need it.

The `try/finally` statement lets you dispose of a resource whether or not an exception has occurred. For example:

```
val in = new URL("http://horstmann.com/fred.gif").openStream()
try {
  process(in)
} finally {
  in.close()
}
```

The finally clause is executed whether or not the process function throws an exception. The reader is always closed.

This code is a bit subtle, and it raises several issues.

- What if the URL constructor or the `openStream` method throws an exception? Then the try block is never entered, and neither is the finally clause. That's just as well—in was never initialized, so it makes no sense to invoke `close` on it.
- Why isn't `val in = new URL(...).openStream()` inside the try block? Then the scope of `in` would not extend to the finally clause.
- What if `in.close()` throws an exception? Then that exception is thrown out of the statement, superseding any earlier one. (This is just like in Java, and it isn't very nice. Ideally, the old exception would stay attached to the new one.)

Note that try/catch and try/finally have complementary goals. The try/catch statement handles exceptions, and the try/finally statement takes some action (usually cleanup) when an exception is not handled. It is possible to combine them into a single try/catch/finally statement:

```
try { ... } catch { ... } finally { ... }
```

This is the same as

```
try { try { ... } catch { ... } } finally { ... }
```

However, that combination is rarely useful.

Exercises

1. The *signum* of a number is 1 if the number is positive, -1 if it is negative, and 0 if it is zero. Write a function that computes this value.
2. What is the value of an empty block expression `{}`? What is its type?
3. Come up with one situation where the assignment `x = y = 1` is valid in Scala. (Hint: Pick a suitable type for `x`.)
4. Write a Scala equivalent for the Java loop


```
for (int i = 10; i >= 0; i--) System.out.println(i);
```
5. Write a procedure `countdown(n: Int)` that prints the numbers from `n` to 0.
6. Write a for loop for computing the product of the Unicode codes of all letters in a string. For example, the product of the characters in "Hello" is 9415087488L.
7. Solve the preceding exercise without writing a loop. (Hint: Look at the `StringOps` Scaladoc.)
8. Write a function `product(s : String)` that computes the product, as described in the preceding exercises.

9. Make the function of the preceding exercise a recursive function.
10. Write a function that computes x^n , where n is an integer. Use the following recursive definition:
- $x^n = y^2$ if n is even and positive, where $y = x^{n/2}$.
 - $x^n = x \cdot x^{n-1}$ if n is odd and positive.
 - $x^0 = 1$.
 - $x^n = 1 / x^{-n}$ if n is negative.

Don't use a return statement.

This page intentionally left blank

Index

Symbols and Numbers

- (minus sign)
 - in identifiers, 132
 - operator:
 - arithmetic, 5
 - for collections, 163–164
 - for maps, 43
 - for type parameters, 237
 - left-associative, 135
 - precedence of, 134
 - unary, 10, 133
- operator
 - arithmetic, 6
 - for collections, 163–164
 - for sets, 162–163
- _ (underscore)
 - as wildcard:
 - for XML elements, 220
 - in case clauses, 25, 184–185, 221, 292
 - in imports, 7, 70, 79–80
 - in tuples, 45
 - for function calls, 144, 254
 - for function parameters, 146
 - in identifiers, 131, 283
- _* syntax
 - for arrays, 187
 - for nested structures, 192
 - in function arguments, 22
 - in pattern matching, 221
- _=, in setter methods, 51
- _1, _2, _3 methods, 45
- ; (semicolon)
 - after statements, 4, 14–16
 - inside loops, 19–20
- : (colon)
 - followed by annotations, 201
 - in case clauses, 186–187
 - in identifiers, 132
 - in implicits, 311–312
 - in operator names, 252
 - and precedence, 134
 - right-associative, 135, 170
 - in type parameters, 234–235
- :: operator, 240
 - for lists, 159–160, 163–164
 - in case clauses, 187, 191
 - right-associative, 135, 160
- ::: operator, 163–164

- :\ operator, 170
- :+ operator, 163–164
- :+= operator, 164
- ! (exclamation mark)
 - in identifiers, 132
 - in shell scripts, 105–106
 - operator:
 - for actors, 291, 294–295
 - precedence of, 134
 - unary, 133
- !!, in shell scripts, 105
- != operator, 133
- !? operator, 295
- ? (question mark)
 - in identifiers, 132
 - in parsers, 274
- ?: operator, 14
- / (slash)
 - in identifiers, 132
 - in XPath, 220
 - operator:
 - arithmetic, 5
 - precedence of, 134
- /: operator, 170
- //
 - for comments, 283
 - in XPath, 220
- /* ... */ comments, 283
- /% operator, 6, 189
- ` (backquote)
 - as escape character, for identifiers, 132
 - in case clauses, 186
- ^ (caret)
 - in identifiers, 132
 - in Pascal, 139
 - operator:
 - arithmetic, 5
 - precedence of, 134
- ^? operator, 279
- ^^ operator, 273–275, 278
- ^^^ operator, 278
- ' (single quote)
 - in symbols, 210
 - parsing, 283
- " (double quote), 283
- """, in regular expressions, 106
- ~ (tilde)
 - in identifiers, 132
 - operator:
 - in case clauses, 191
 - in parsers, 271–277, 279–280
 - unary, 133
- ~! operator, 279–280
- ~> operator, 274–275, 278
- () (parentheses)
 - as shortcut for apply method, 8
 - as value of Unit, 14–15, 17
 - discarding, in parsers, 274
 - for annotations, 201
 - for continuations, 323, 335
 - for functions, 144–146, 151
 - for maps, 42
 - for tuples, 45, 253
 - in case clauses, 187, 190
 - in method declarations, 7, 50, 54
 - in regular expressions, 107
 - to access XML attributes, 216
- [] (square brackets)
 - for methods in traits, 117
 - for type parameters, 232, 253
- { } (braces)
 - for block expressions, 16–17
 - for existential types, 252
 - for function arguments, 145
 - for structural types, 250–251
 - in imports, 80
 - in package clauses, 77
 - in pattern matching, 195–196, 221
 - in REPL, 15
 - in XML literals, 218
 - Kernighan & Ritchie style for, 16
- @ (at), 204
 - for XML attributes, 220
 - in case clauses, 192
 - in identifiers, 132
- \ (backslash)
 - for nodes, 220–221
 - in identifiers, 132
- \\ operator, 220–221

- * (asterisk)
 - as wildcard in Java, 7, 79
 - in identifiers, 132
 - in parsers, 274
 - operator:
 - arithmetic, 5, 308–309
 - no infix notation for, 252
 - precedence of, 134
- **
 - in Fortran, 139
 - in identifiers, 132
- & (ampersand)
 - in identifiers, 132
 - operator:
 - arithmetic, 5
 - for sets, 162–164
 - precedence of, 134
- &...; (XML), 215
- &~ operator, 162–163
- &#...; (XML), 216
- # (number sign), 62
 - for type projections, 247–249, 253
 - in identifiers, 132
- :: operator, 173
- ## operator, 106
- #< operator, 105–106
- #> operator, 105
- #>> operator, 105
- #| operator, 105
- #|| operator, 106
- % (percent sign)
 - for XML attributes, 222
 - in identifiers, 132
 - operator:
 - arithmetic, 5
 - precedence of, 134
- + (plus sign)
 - in identifiers, 132
 - operator:
 - arithmetic, 5
 - for collections, 163–164
 - for maps, 43
 - for type parameters, 237
 - precedence of, 134
 - unary, 133
- +: operator
 - for collections, 163–164
 - in case clauses, 191
 - right-associative, 135, 163
- ++ operator
 - arithmetic, 6
 - for collections, 163–164
 - for sets, 162–163
- ++: operator, 163–164
- ++= operator
 - for array buffers, 30
 - for collections, 163–164
- ++=: operator, 163–164
- += operator, 315
 - assignment, 133
 - for array buffers, 30, 36
 - for collections, 163–164, 314
 - for maps, 43
- +=: operator, 163–164
- < (left angle bracket)
 - in identifiers, 132
 - in XML literals, 214
 - operator:
 - and implicits, 310–311
 - precedence of, 134
- <- operator, 18–19, 189
- <: operator, 233, 235–237, 252, 259
- <:< operator, 236, 312, 314
- <!- ... -> comments, 215
- <? ... ?> (XML), 215
- <?xml...?> (XML), 225
- <~ operator, 274–275, 278
- <% operator, 234
- <%< operator, 236, 312–313
- << operator, 5
- <= operator, 133
- > (right angle bracket)
 - in identifiers, 132
 - operator, 134
- >: operator, 233, 235
- >= operator, 133
- >> operator
 - arithmetic, 5
 - in parsers, 278

- = operator
 - for collections, 163–164
 - for maps, 43
 - === operator, 163–164
 - > operator
 - for maps, 41–42
 - precedence of, 134
 - = (equal sign)
 - in identifiers, 132
 - operator:
 - assignment, 133–134
 - precedence of, 134
 - with CPS annotations, 326
 - := operator, 236, 312–313
 - ≠ operator, 134
 - == operator, 134, 210
 - for reference types, 96
 - === operator, 134
 - => operator
 - for continuations, 321–324
 - for functions, 151, 253–254
 - for self types, 124–125, 260
 - in case clauses, 184–188, 190–192, 194–195
 - | (vertical bar)
 - in identifiers, 132
 - operator:
 - arithmetic, 5
 - for sets, 162–163
 - in parsers, 270–286
 - precedence of, 134
 - √ (square root), 132
 - 80 bit extended precision, 204
- A**
- abstract keyword, 91, 113, 117
 - accept method, 279
 - act method, 290, 297–301
 - blocking calls inside, 295
 - running concurrently, 290
 - Actor trait, 290, 299
 - Actor companion object, 290
 - actors, 289–302
 - anonymous, 291
 - blocking, 292, 295–296, 302
 - calling methods on, 302
 - creating, 290–291
 - global, 293
 - linking, 300–301
 - references to, 293–294
 - sharing threads for, 296–299
 - starting, 290, 299
 - terminating, 299–301
 - addString method, 166, 173
 - aggregate method, 165, 173, 179
 - Akka project, 289
 - aliases, 62, 157, 249, 255
 - Annotation trait, 202
 - annotations, 199–211, 253
 - arguments of, 201–202
 - deprecated, 204
 - for compiler optimizations, 206–210
 - implementing, 202–203
 - in Java, 200–206
 - meta-annotations for, 203
 - order of, 200
 - Any class, 94, 96
 - AnyRef class, 94–95, 102, 313
 - AnyVal class, 94
 - Apache Commons Resolver project, 224
 - App trait, 68
 - append method, 35
 - appendAll method, 35
 - Application trait, 69
 - apply method, 8, 67–68, 106, 135–137, 157, 190, 195, 314–315
 - args property, 69
 - array buffers, 30–31
 - adding/removing elements of, 30
 - appending collections to, 30
 - converting to arrays, 31
 - displaying contents of, 34
 - empty, 30
 - largest/smallest elements in, 34
 - parallel implementations for, 178
 - sorting, 34
 - transforming, 32–33
 - traversing, 31–32
 - Array class, 29–30, 35, 235
 - Array companion object, 8, 188

- ArrayBuffer class, 30–31, 156, 315
 - mutable, 159
 - serializing, 104
 - subclasses of, 36
 - ArrayList class (Java), 30, 37, 157
 - ArrayOps class, 35
 - arrays, 29–37
 - converting to array buffers, 31
 - displaying contents of, 34
 - fixed-length, 29–30
 - function call syntax for, 136
 - generic, 235
 - interoperating with Java, 37
 - invariance of, 238
 - largest/smallest elements in, 34
 - multidimensional, 37, 68
 - parallel implementations for, 178
 - pattern matching for, 187
 - ragged, 37
 - sorting, 34
 - transforming, 32–33
 - traversing, 18, 31–32
 - variable-length. *See* array buffers
 - vs. lists, 156
 - ArrayStoreException, 239
 - asAttrMap method, 217
 - ASCII characters, 132
 - asInstanceOf method, 87, 94, 186
 - asJavaCollection function, 176
 - asJavaConcurrentMap function, 176
 - asJavaDictionary function, 176
 - asJavaEnumeration function, 176
 - asJavaIterable function, 176
 - asJavaIterator function, 176
 - asScalaBuffer function, 176
 - asScalaConcurrentMap function, 176
 - asScalaIterator function, 176
 - asScalaSet function, 176
 - assert method, 209
 - AssertionError, 209
 - assignments, 16–17, 133–134
 - no chaining of, 17
 - precedence of, 134
 - right-associative, 135, 163
 - value of, 17
 - Atom class, 217–219
 - Attribute trait, 222
 - attributes (XML), 216–217
 - atoms in, 218
 - entity references in, 218
 - expressions in, 218–219
 - iterating over, 217
 - matching, 222
 - modifying, 222–223
 - automatic conversions. *See* implicits
- ## B
- backtracking, 279–280
 - balanced trees, 44
 - parallel implementations for, 178
 - bash shell, 105
 - bean properties, 55–56
 - @BeanDescription annotation, 206
 - @BeanDisplayName annotation, 206
 - @beanGetter annotation, 203
 - @BeanInfo annotation, 206
 - @BeanInfoSkip annotation, 206
 - @BeanProperty annotation, 55–56, 200, 205
 - generated methods for, 59
 - @beanSetter annotation, 203
 - BigDecimal class, 5–6
 - BigInt class, 5–7, 139
 - BigInt companion object, 7–8
 - BitSet class, 162
 - blocks, 16–17
 - BNF (Backus-Naur Form), 270
 - Boolean type, 4, 17
 - @BooleanBeanProperty annotation, 205
 - break method, 19
 - Breaks object, 19
 - Buffer class, 315
 - bufferAsJavaList function, 176
 - buffered method, 100
 - BufferedInputStream class (Java), 128
 - Byte type, 4, 17
 - arrays of, 102
- ## C
- C programming language, 184
 - C++ programming language
 - ?: operator in, 14
 - arrays in, 30

- C++ programming language (*cont.*)
 - assignments in, 17
 - construction order in, 94
 - exceptions in, 24
 - expressions in, 13–15
 - functions in, 20–21
 - implicit conversions in, 306
 - linked lists in, 160
 - loops in, 18, 32
 - methods in, 66, 88
 - multiple inheritance in, 111–112
 - namespaces in, 74
 - operators in, 134
 - protected fields in, 88
 - reading files in, 100
 - singleton objects in, 66
 - statements in, 13, 15–16
 - switch in, 207
 - virtual base classes in, 112
 - void in, 15, 17, 95
- cake pattern, 256
- case keyword, 184, 189
 - catch-all pattern for, 184–185
 - enclosed in braces, 195–196
 - followed by variable, 185
 - infix notation in, 191
- case classes, 189–196
 - applicability of, 192–193
 - declaring, 190
 - default methods of, 137, 190, 193
 - extending other case classes, 193
 - for channels, 294–295
 - for messages from actors, 291–292
 - in parsers, 272, 275
 - modifying properties in, 190
 - sealed, 193–194
 - with variable fields, 193
- case objects, 189–190
- casts, 87–88
- CatalogResolver class (Java), 224
- catch statement, 25–26
- CDATA markup, 219, 224
- chaining
 - assignments, 17
 - auxiliary constructors, 59
 - method calls, 36
 - packages, 76–77
- chain11 method, 278
- Channel class, 294–295
- Char type, 4, 17, 281
- character references, 216
- character sets, 102
- characters
 - common, in two strings, 5
 - in identifiers, 132, 283
 - reading, 17, 100–101
 - sequences of, 10
 - uppercase, 10
- circular dependencies, 24, 125
- class keyword, 49, 253
- class files, 202
- ClassCastException, 209
- classes, 8, 49–62, 253
 - abstract, 91
 - abstract types in, 257
 - and primitive types, 4
 - annotated, 200
 - case. *See* case classes
 - combined with primary constructor, 60
 - concrete, 120
 - definitions of, 58
 - using traits in, 115
 - equality in, 95
 - extending, 67, 85–86
 - Java classes, 89
 - only one superclass, 119
 - granting access to, 55–56
 - immutable, 6
 - implementing, 231
 - importing members of, 70, 79
 - inheritance hierarchy of, 94–95
 - interoperating with Java, 52
 - linearization of, 121
 - mutable, 193
 - names of, 131–132
 - nested, 60–62, 247
 - properties of, 51, 53
 - serializable, 104, 204
 - type aliases in, 249
 - type parameters in, 232

- visibility of, 50
 - vs. singletons, 7
 - vs. traits, 122
- ClassfileAnnotation trait, 202
- classOf method, 87
- Cloneable interface (Java), 114, 204
- @cloneable annotation, 204
- close method, 100
- closures, 148
- collect method, 165, 168, 173, 196
- collectionAsScalaIterable function, 176
- collections, 155–179
 - adding/removing elements of, 163–164
 - applying functions to all elements of, 147, 165–168
 - combining, 171–172
 - companion objects of, 315
 - constructing instances of, 157
 - converting to specific type, 166
 - filtering, 165
 - folding, 165, 169–171
 - hierarchy of, 35, 156–157
 - immutable, 157–158
 - interoperating with Java, 175–177
 - methods for, 164–167
 - mutable, 157–158, 164, 177
 - ordered, 156, 163
 - parallel, 178–179
 - reducing, 165, 168–169
 - scanning, 165, 171
 - serializing, 104
 - threadsafe, 177
 - traits for, 156–157
 - traversing, 18, 32, 156, 206–207
 - unevaluated, 174
 - unordered, 156, 163–164
 - vs. iterators, 173
- com.sun.org.apache.xml.internal.resolver.tools package, 224
- combinators, 277–280
- command-line arguments, 69
- comma-separated lists, 277
- comments
 - in lexical analysis, 270
 - in XML, 215
 - parsing, 224, 282–283
- companion objects, 7, 62, 66–67, 136, 157, 248, 310
 - implicits in, 307
- Comparable interface (Java), 36, 233–234, 310
- Comparator class (Java), 210–211
- compareTo method, 233
- compiler
 - CPS transformations in, 332
 - implicits in, 309, 313–314
 - internal types in, 254
 - optimizations in, 206–210
 - Scala annotations in, 200
 - transforming continuations in, 325
- compiler plugin, 200
- Component class (Java), 127
- compound types, 250–251, 253
- comprehensions, 20
- computation with a hole, 321–324, 328
- concurrency, 178
- ConcurrentHashMap class (Java), 177
- ConcurrentSkipListMap class (Java), 177
- console
 - input from, 17, 101
 - printing to, 17, 103
- Console class, 103
- ConsoleLogger trait, 115
- constants. *See* values
- ConstructingParser class, 224–225
- constructors
 - auxiliary, 56–57, 88
 - chaining, 59
 - eliminating, 58
 - order of, 92–94
 - parameterless, 58, 122
 - parameters of, 55, 57–60
 - annotated, 203
 - implicit, 235
 - primary, 56–60, 88
 - annotated, 201
 - private, 60
 - superclass, 88–89
 - vals in, 93
- Container class (Java), 127
- contains method, 42, 162, 166, 173
- containsSlice method, 166, 173
- context bounds, 234–235

- continuations, 319–336
 - boundaries of, 320
 - capturing, 320–321, 326, 330
 - in web applications, 329–332
 - invoking, 320–323
 - plugin for, 321
- control abstractions, 151–152
- control flow
 - combinators for, 298
 - inversion of, 329
 - using continuations for, 319–336
- ControlContext class, 332–336
- copy method, 193, 222
 - of case classes, 190
- copyToArray method, 36, 166, 173
- copyToBuffer method, 166, 173
- corresponds method, 150, 237
- count method, 10, 36, 165, 173
- CPS (continuation-passing style)
 - transformations, 325–327, 332–336
 - code generated by, 334
 - of nexted control contexts, 334–336
- @cps annotation, 325–327, 330
- @cpsParam annotation, 325
- Curry, Haskell Brooks, 149
- D**
- deadlocks, 289, 295, 302
- debugging
 - reading from strings for, 102
 - reporting types for, 34
- def keyword, 20
 - abstract, 89
 - in parsers, 280
 - overriding, 89–90
 - parameterless, 89
 - return value of, 280
- default statement, 184
- definitions, 19–20
- DelayedInit trait, 69
- Delimiters type, 309
- dependency injections, 255–257
- @deprecated annotation, 203, 210
- @deprecatedName annotation, 202, 210
- destructuring, 188, 191
- diamond inheritance problem, 112–113
- dictionaryAsScalaMap function, 176
- diff method, 162, 167, 173
- directories
 - and packages, 74
 - naming, 11
 - printing, 104, 326
 - traversing, 103–104
- Directory class, 103
- do loop, 18
- docElem method, 224
- DocType class, 225
- domain-specific languages, 131, 269
- Double type, 4, 17
- DoubleLinkedList class (Java), 159, 161
- drop method, 165, 173
- dropRight method, 165
- dropWhile method, 165, 173
- DTDs (Document Type Definitions), 224–225
- duck typing, 250
- dynamically typed languages, 250
- E**
- early definitions, 93, 122–123
- EBNF (Extended Backus-Naur Form), 271–272
- Eiffel programming language, 53
- Either type, 266
- elem keyword, 160–161
- Elem type, 214, 222, 227, 281
- elements (XML), 214
 - attributes of. *See* attributes (XML)
 - child, 221–222
 - empty, 226
 - matching, 220
 - modifying, 222–223
- @elidable annotation, 208–209
- empty keyword, 161
- Empty class, 240
- endsWith method, 166, 173
- entity references, 215
 - in attributes, 216, 218
 - resolving, 225
- EntityRef class, 216
- Enumeration class, 69–71
- enumerationAsScalaIterator function, 176

- enumerations, 69–71
 - simulating, 194
 - eq method, 95
 - equals method, 95–96, 190, 193
 - overriding, 96
 - parameter type of, 96
 - err method, 279
 - error messages, 86
 - explicit, 285
 - type projections in, 249
 - escape hatch, 132
 - event handlers, 297
 - eventloop method, 299
 - evidence objects, 313
 - Exception trait, 254
 - exceptionHandler method, 300
 - exceptions, 24–26
 - catching, 25
 - checking at compile time, 24
 - in Java, 204–205
 - exists method, 165, 173
 - exit method, 299–300
 - expressions
 - annotated, 201
 - conditional, 14–15
 - traversing values of, 18
 - type of, 14
 - vs. statements, 13
 - extends keyword, 85, 93, 113–114
 - extractors, 107, 136–138, 188
- F**
- failure method, 279
 - fall-through problem, 184
 - family polymorphism, 259–262
 - @field annotation, 203
 - fields
 - abstract, 91–92, 119–120, 122
 - accessing uninitialized, 93
 - annotated, 200
 - comparing, 193
 - concrete, 92, 118–119
 - copying, 193
 - for primary constructor parameters, 55, 59
 - getter/setter methods for, 51, 55–56, 59
 - hash codes of, 96, 193
 - immutable, 59
 - object-private, 54–55, 59
 - overriding, 89–90, 119–120, 122
 - printing, 193
 - private, 53–54
 - private final, 53
 - protected, 88
 - public, 50
 - static, 65
 - transient, 203
 - volatile, 203
 - File class, 103
 - file2RichFile method, 308
 - FileInputStream class (Java), 102
 - files
 - and packages, 74
 - appending, 105
 - binary, 102
 - naming, 11
 - processing, 99–106
 - reading, 100–101, 320
 - redirecting input/output for, 105
 - saving, 225–226
 - writing, 102–103
 - Files class (Java), 103–104
 - FileVisitor interface (Java), 103
 - filter method, 33, 147, 165, 173, 195
 - final keyword, 53
 - finally statement, 25–26
 - findAllIn, findFirstIn methods, 106
 - findPrefixOf method, 107
 - flatMap method, 165, 167–168, 173, 333, 335–336
 - Float type, 4, 17
 - floating-point calculations, 204
 - fluent interfaces, 246–247
 - fold method, 165, 173, 179
 - foldLeft method, 152–153, 165, 169–170, 173, 179, 239
 - foldRight method, 165, 170, 173, 179
 - for loop, 18–20
 - annotated as CPS, 327
 - enhanced (Java), 32
 - for arrays, 31–33
 - for maps, 43–44

- for loop (*cont.*)
 - for regex groups, 107
 - parallel implementations for, 178
 - pattern matching in, 189
 - range-based (C++), 32
 - regular expressions in, 106
 - with `Option` type, 195
 - `forall` method, 165, 173
 - `force` method, 175
 - `foreach` method, 147, 165, 168, 173, 195, 327
 - `format` method, 102
 - Fortran programming language, 139
 - `Fraction` class, 136–137
 - `Fraction` companion object, 307
 - `fraction2Double` method, 308
 - `FractionConversions` companion object, 307
 - fragile base class problem, 86
 - French delimiters, 310
 - `fromString` method, 102
 - `fromURL` method, 102
 - functional programming languages, 143
 - functions, 20–21, 143–152, 253
 - anonymous, 21, 144–146, 152
 - as method parameters, 10, 144
 - binary, 147–148, 168
 - calling, 7, 144
 - curried, 149–151, 309
 - defining, 20
 - exiting immediately, 21
 - from methods, 254
 - higher-order, 145–148
 - implementing, 231
 - importing, 7
 - left-recursive, 276
 - mapping, 167–168
 - names of, 10, 131–132, 306
 - nested, 19
 - parameterless, 150–151, 320
 - parameters of, 20, 145–146
 - call-by-name, 151
 - default, 21
 - named, 21
 - only one, 146, 238
 - type, 232
 - type deduction in, 146
 - variable, 22–23
 - partial, 168, 195–196, 279, 292, 297
 - passing to another function, 144–146, 149
 - recursive, 20–22
 - return type of, 4, 20, 23
 - return value of, 150–152, 320
 - scope of, 148
 - storing in variables, 143–144
 - syntax of, 135–136
 - vs. variables, in parsers, 279
- ## G
- generators, 19–20
 - `GenIterable` trait, 178
 - `GenMap` trait, 178
 - `GenSeq` trait, 178
 - `GenSet` trait, 178
 - `GenTraversable` trait, 196
 - `get` method, 42, 194, 216
 - `getLines` method, 100, 174
 - `getOrElse` method, 42, 195, 217
 - `getResponse` method, 329–331
 - `@getter` annotation, 203
 - `getXxx` methods, 52, 55, 205
 - grammars, 270–271
 - left-recursive, 280
 - `Group` type, 219
 - `grouped` method, 166, 172–173
 - `guard` method, 279
 - guards, 19–20, 32, 185
 - for pattern matching, 222
 - in for statements, 189
 - variables in, 185
- ## H
- hash codes, 94, 96
 - hash maps, 293
 - hash sets, 161
 - hash tables, 41, 44
 - parallel implementations for, 178
 - `hashCode` method, 96, 161, 190, 193
 - overriding, 96
 - Haskell programming language, 21
 - `hasNext` method, 118, 173
 - `head` method, 100, 159–160, 165
 - `headOption` method, 165

Hindley-Milner algorithm, 21
HTTP (Hypertext Transfer Protocol), 102, 269

I

id method, 70
ident method, 284
identifiers, 131–132, 283
identity functions, 313
IEEE double values, 204
if/else expression, 14–15, 25
implements keyword, 113
implicit keyword, 10, 306, 309–311
implicit conversions, 10, 36–37, 131, 149, 305–316
 adapting functions with, 103
 ambiguous, 308–309
 for parsers, 282
 for strings to `ProcessBuilder` objects, 105
 for type parameters, 234
 importing, 307–308, 312
 multiple, 308
 naming, 306
 rules for, 308–309
 unwanted, 175, 306–307
 uses of, 306–307
implicit parameters, 235, 265, 309–316
 not available, 210, 313
 of common types, 310
implicit values, 234–235
implicitly method, 311–313
@implicitNotFound annotation, 210, 313–314
:implicits in REPL, 307
import statement, 70, 74, 79–81
 implicit, 80–81, 104
 location of, 80
 overriding, 81
 selectors for, 80
 wildcards in, 7, 79–80
inching forward, 333
IndexedSeq trait, 156, 315
IndexedSeq companion object, 315
indexOf method, 166, 173
indexOfSlice method, 166, 173
indexWhere method, 166, 173
infix notation, 132–133, 251–253

 in case clauses, 191
 in math, 251
 with anonymous functions, 145
inheritance hierarchy, 94–95
init method, 165
inline annotation, 208
InputChannel trait, 294
InputStream class (Java), 223
Int type, 4, 17, 234, 236
 immutability of, 6
 no null value in, 95
int2Fraction method, 307–308
Integer class (Java), 209
intersect method, 5, 162, 167, 173
intersection types. *See* compound types
into combinator, 277–278
inversion of control problem, 297
isDefinedAt method, 195
isEmpty method, 165, 173
isInstanceOf method, 87, 94, 186
isSet method, 296
istream::peek function (C++), 100
Iterable trait, 35, 156, 239, 263–265
 and parallel implementations, 178
 important methods of, 164–167, 173
iterableAsScalaIterable function, 176
iterator method, 172
Iterator trait, 118, 156, 173
iterators, 100, 172–173
 from iterations, 337
 from recursive visits, 326–329, 334–336
 mutable, 173
 next method of, 329
 turning into arrays, 106
 vs. collections, 173
 weakly consistent, 177

J

Java programming language
 ?: operator in, 14
 annotations in, 200–206
 arrays in, 30, 37, 157, 239
 assertions in, 209
 assignments in, 17
 asynchronous channels in, 302
 casts in, 87

- Java programming language (*cont.*)
 - checked exceptions in, 205
 - classes in, 85–86
 - hierarchy of, 61
 - serializable, 104
 - vs. Scala, 8
 - closures in, 148
 - construction order in, 94
 - dependencies in, 256
 - event handling in, 259
 - exceptions in, 24, 204
 - expressions in, 13–15
 - fields in:
 - protected, 88
 - public, 50
 - identifiers in, 131–132
 - imports in, 7
 - interfaces in, 111–114, 125–126
 - interoperating with Scala:
 - arrays, 37
 - classes, 52, 89, 200, 204
 - collections, 175–177
 - fields, 203–204
 - maps, 44–45, 189
 - methods, 204–205
 - traits, 125–126
 - linked lists in, 157, 160
 - loops in, 18, 32
 - maps in, 156
 - methods in, 66, 86, 88
 - abstract, 91
 - overriding, 93
 - static, 7, 20–21
 - with variable arguments, 23
 - missing values in, 236
 - modifiers in, 203–204
 - no multiple inheritance in, 111
 - no variance in, 211
 - null value in, 95
 - objects in, 161
 - operators in, 134
 - packages in, 74, 76, 78
 - primitive types in, 30, 94
 - reading files in, 100–102
 - SAM types in, 149
 - singleton objects in, 66
 - statements in, 13, 15–16
 - superclass constructors in, 89
 - switch in, 207
 - synchronized in, 95
 - toString in, 34
 - traversing directories in, 103–104
 - type checks in, 87
 - void in, 15, 17, 95
 - wildcards in, 79, 241, 252
- Java AWT library, 127
- java.io.InputStream class, 223
- java.io.Reader class, 223
- java.io.Writer class, 225
- java.lang package, 80–81
- java.lang.Integer class, 209
- java.lang.ProcessBuilder class, 37, 105–106
- java.lang.String class, 5, 234
- java.lang.Throwable class, 24
- java.math.BigDecimal class, 5
- java.math.BigInteger class, 5
- java.nio.file.Files class, 103–104
- java.util package, 176
- java.util.Comparator class, 210–211
- java.util.concurrent package, 177
- java.util.Properties class, 44, 189
- java.util.Scanner class, 46, 101
- java.util.TreeSet class, 162
- JavaBeans, 55–56, 127, 205–206
- JavaConversions class, 37, 44, 175–177
- JavaEE, 200
- JavaScript, 219
 - closures in, 148
 - duck typing in, 250
- JavaTokenParsers trait, 282–283
- JButton class (Swing), 127
- JComponent class (Swing), 127
- JDK (Java Development Kit), 196, 224
- JSON (JavaScript Object Notation), 269
- jump tables, 207
- JUnit, 200–201
- JVM (Java Virtual Machine)
 - continuation support in, 332
 - generic types in, 235
 - inlining in, 208
 - stack in, 206, 325
 - transient/volatile fields in, 203

K

Kernighan & Ritchie brace style, 16
keySet method, 44

L

last method, 165
lastIndexOf method, 166, 173
lastIndexOfSlice method, 166, 173
lastOption method, 165
lazy keyword, 23–24, 123
length method, 165, 173
lexers, 270
lexical analysis, 270
li (XML), 217–218
link method, 300–301
linked hash sets, 162
LinkedHashMap class, 44
LinkedList class (Java), 157, 159–161
List class, 191, 263, 272
 immutable, 157–158
 implemented with case classes, 193
List interface (Java), 157
lists, 159–160
 adding/removing elements of, 163–164
 constructing, 135, 159
 destructuring, 160, 191
 empty, 95
 heterogeneous, 196
 immutable, 173, 240
 linked, 156
 mutable, 160–161
 order of elements in, 161
 pattern matching for, 187–188
 traversing, 160
 vs. arrays, 156
literals. *See* XML literals
loadFile method, 223
locks, 289
log method, 279
log messages
 adding timestamp to, 116
 printing, 279
 truncating, 116
 types of, 118
Logged trait, 115–116
LoggedException trait, 125

Logger trait, 118
Long type, 4, 17
loop combinator, 298
loops, 18–20
 breaking out of, 19
 for collections, 18
 infinite, 298–299
 variables within, 19
 vs. folding, 170–171
loopWhile combinator, 298

M

mailboxes, 292–293, 296–299, 301–302
main method, 68
makeURL method, 218
Manifest object, 235, 265
map method, 33, 147, 165, 167–168, 173, 195,
 263–264, 333–335
Map trait, 41–42, 156, 194
 immutable, 157
mapAsJavaMap function, 176
mapAsScalaMap function, 44, 176
maps, 41–46
 blank, 42
 constructing, 41–42
 from collection of pairs, 46
 function call syntax for, 136
 immutable, 42–43
 interoperating with Java, 44–45
 iterating over, 43–44
 keys of:
 checking, 42
 removing, 43
 visiting in insertion order, 44
 mutable, 42–43
 reversing, 44
 sorted, 44
 traversing, 189
 values of, 42–43
match expression, 184–188, 190–192,
 194–195, 207–208, 237
MatchError, 184
mathematical functions, 7, 10
max method, 34, 36, 165, 173
maximum munch rule, 284
MessageFormat.format method (Java), 23

- messages
 - asynchronous, 291–292
 - case classes for, 291–292
 - contextual data in, 302
 - receiving, 292–293
 - returning to sender, 294–295
 - sending, 293–294
 - serializing, 293
 - synchronous, 295–296, 302
 - MetaData classtype, 216–217, 222
 - method types (in compiler), 254
 - methods
 - abstract, 89, 91–92, 113, 117, 125
 - abundance of, 8, 10
 - accessor, 50
 - annotated, 200
 - calling, 2, 4, 7, 50–51, 117
 - chained, 246
 - co-/contravariant, 313
 - concrete, 125
 - declaring, 50
 - eliding, 208–209
 - executed lazily, 174–175
 - final, 86, 96, 207
 - for primary constructor parameters, 59
 - getter, 51–54, 92, 200, 205
 - in superclass, 86–87
 - inlining, 208
 - modifiers for, 78–79
 - mutator, 50
 - names of, 6
 - misspelled, 86
 - overriding, 86–87, 89–90, 117
 - parameterless, 7, 50, 89
 - parameters of, 86, 232, 239, 246
 - two, 6
 - type, 232
 - using functions for, 10, 144
 - private, 53, 207
 - protected, 88, 299
 - public, 51
 - return type of, 239, 246, 326, 336
 - return value of, 232
 - setter, 51–54, 92, 200, 205
 - static, 65, 125
 - turning into functions, 144, 254
 - used under certain conditions, 236
 - variable-argument, 23, 205
 - with shift, 325–326
 - Meyer, Bertrand, 53
 - min method, 7, 34, 165, 173
 - mkString method, 34, 166, 173
 - ML programming language, 21
 - monad laws, 333
 - mulBy function, 145, 148
 - multiple inheritance, 111–113
 - mutableMapAsJavaMap function, 176
 - mutableSeqAsJavaList function, 176
 - mutableSetAsJavaSet function, 176
- ## N
- NamespaceBinding class, 226
 - namespaces, 226–227
 - @native annotation, 204
 - negation operator, 10
 - new keyword, 61
 - omitting, 136, 190, 192–193
 - newline character
 - in long statements, 16
 - in printed values, 17
 - inside loops, 20
 - next method, 118, 160–161, 173, 329
 - Nil list, 95, 159–160, 210, 240
 - Node type, 214–216, 240
 - node sequences, 214
 - binding variables to, 221
 - descendants of, 220
 - grouping, 219
 - immutable, 216, 222
 - traversing, 214
 - turning into strings, 216
 - NodeBuffer class, 215–216
 - NodeSeq type, 214–216, 220–221
 - @noinline annotation, 208
 - None object, 194–195, 272–273
 - nonterminal symbols, 271
 - not method, 279
 - Nothing type, 25, 95, 237, 240
 - notify method, 95
 - notifyAll method, 95
 - null value, 95, 236
 - Null type, 95, 223

NumberFormatException, 101

numbers

classes for, 10

converting:

between numeric types, 5, 8

to arrays, 101

greatest common divisor of,
139

in identifiers, 283

invoking methods on, 4

parsing, 278, 283

random, 7

ranges of, 10

reading, 17, 101

sums of, 34

writing, 102

numericLit method, 284

O

object keyword, 65–70, 247

Object class, 94–95

objects, 65–70

adding traits to, 115

cloneable, 204

compound, 193

constructing, 8, 50, 66, 115

default methods for, 161

equality of, 94–96

extending class or trait, 67

extracting values from, 188

importing members of, 70, 79

nested, 192

nested classes in, 60–62, 247

no type parameters for, 240

of a given class, 87–88

pattern matching for, 186

remote, 204

scope of, 248

serializable, 104, 250

type aliases in, 249

ofDim method, 37

operators, 131–138

arithmetic, 5–6

assignment, 133–135

associativity of, 135, 179

binary, 133–135

for adding/removing elements, 162–164

infix, 132–134

parsing, 284

postfix, 134

precedence of, 134–135, 252, 273

unary, 133

opt method, 271–272

Option class, 42, 106, 136, 138, 165, 194–195,
217, 236, 272–273

Ordered trait, 34, 36, 234, 310–312

Ordering type, 36, 311–312

orNull method, 236

OSGi (Open Services Gateway initiative
framework), 256

OutOfMemoryError, 174

OutputChannel trait, 294

override keyword, 86–87, 89–90, 113, 117
omitted, 91–92

@Overrides annotation, 86

P

package objects, 78

packages, 74–81

adding items to, 74

chained, 76–77

defined in multiple files, 74

importing, 79–81

always, 80–81, 104

selected members of, 80

modifiers for, 78–79, 88

naming, 76–77, 81

nested, 75–77

scope of, 248

top-of-file notation for, 77

packrat parsers, 280–281

PackratParsers trait, 280

PackratReader class, 281

padTo method, 36, 166, 173

Pair class, 239, 241

par method, 178

@param annotation, 203

parameters

annotated, 200

curried, 237

deprecated, 210

named, 190

- ParIterable trait, 178
- ParMap trait, 178
- parse method, 272
- parse trees, 274–275
- parseAll method, 272, 279, 281, 284–285
- ParSeq trait, 178
- parsers, 269–286
 - backtracking in, 279–280
 - entity map of, 225
 - error handling in, 285–286
 - numbers in, 278
 - output of, 273–274
 - regex, 282–283, 286
 - strings in, 278
 - whitespace in, 282
- Parsers trait, 271, 281–286
- ParSet trait, 178
- PartialFunction class, 195–196, 292
- partition method, 46, 165, 173
- Pascal programming language, 139
- patch method, 10
- paths, 248–249
- pattern matching, 183–196
 - and `+` operator, 164
 - by type, 186–187
 - classes for. *See* case classes
 - extractors in, 136
 - failed, 136
 - for arrays, 187
 - for lists, 160, 187–188
 - for maps, 43
 - for objects, 186
 - for tuples, 45, 187–188
 - guards in, 185
 - in actors, 291
 - in XML, 221–222
 - jump tables for, 207
 - nested, 192
 - not exhaustive, 210
 - variables in, 185–186
 - vs. type checks and casts, 87–88
 - with `Option` type, 195
- PCData type, 219
- permutations method, 167, 173
- phrase method, 279
- pipng, 105
- polymorphism, 192
- Positional trait, 279, 286
- positioned method, 279, 286
- pow method, 7, 139
- Predef object, 87, 157, 209
 - always imported, 80–81
 - implicits in, 310–313
- prefixLength method, 166, 173
- PrettyPrinter class, 226
- prev method, 161
- print method, 17, 101
- printf method, 17, 102–103
- println method, 17
- PrintStream.printf method (Java), 23
- PrintWriter class (Java), 102
- PriorityQueue class, 159
- private keyword, 51–62, 78
- probablePrime method, 7
- procedures, 23
- process method, 330–331
- Process object, 106
- process control, 105–106
- ProcessBuilder class (Java), 37
 - constructing, 106
 - implicit conversions to, 105
- processing instructions, 215
- product method, 165, 173
- programs
 - concurrent, 178
 - displaying elapsed time for, 69
 - implicit imports in, 80–81, 104
 - pipng, 105
 - readability of, 6
 - self-documenting, 262
- properties, 51
 - in Java. *See* bean properties
 - read-only, 53
 - write-only, 54
- Properties class (Java), 44, 189
- propertiesAsScalaMap function, 176
- property change listener, 127
- PropertyChangeSupport class (Java), 127
- protected keyword, 78, 88
- public keyword, 50, 78
- PushbackInputStream class (Java), 100
- Python, 148

Q

Queue class, 158–159
quickSort method, 34

R

r method, 106
race conditions, 289, 293–294
Random object, 7
RandomAccess interface (Java), 157
Range class, 4, 10, 263–264, 315
 immutable, 158
 traversing, 18
raw string syntax, 106
react method, 294, 297–299, 302
reactWithin method, 296
read method, 308
readBoolean method, 17
readByte method, 17
readChar method, 17
readDouble method, 17, 101
Reader class (Java), 223
readFloat method, 17
readInt method, 17, 101
readLine method, 17
readLong method, 17, 101
readShort method, 17
receive method, 292–295
receiveWithin method, 296
recursions, 158
 for lists, 160
 infinite, 298
 left, 276–277
 tail, 206–207
 turning into iterations, 326–329, 334–336
red-black trees, 162
reduce method, 165, 173, 179
reduceLeft method, 147, 165, 168, 173, 179
reduceRight method, 165, 169, 173, 179
reference types
 = operator for, 96
 assigning null to, 95
reflective calls, 250
Regex class, 106
RegexParsers trait, 271, 281–283, 286
regular expressions, 106–107
 for extractors, 188

 grouping, 107
 in parsers, 282–283
 matching tokens against, 271
 raw string syntax in, 106
 return value of, 272
Remote interface (Java), 204
@remote annotation, 204
rep method, 271–272, 277–278
rep1 method, 278
rep1sep method, 278
REPL (read-eval-print loop), 2–3
 braces in, 15
 implicits in, 307, 313
 paste mode in, 15, 67
 types in, 144, 249
replaceAllIn method, 107
replaceFirstIn method, 107
reply method, 295
repN method, 278
repsep method, 278
reset method, 320–336
 value of, 323
 with type parameters, 323–325
restart method, 301
result method, 207
return keyword, 21, 152
reverse method, 167, 173
RewriteRule class, 223
rich interfaces, 118
RichChar class, 5
RichDouble class, 5, 10
RichFile class, 306–307
RichInt class, 5, 10, 18, 31, 234
RichString class, 234
root in package names, 76–77
Ruby programming language
 closures in, 148
 duck typing in, 250
RuleTransformer class, 223
run method (Java), 290
Runnable interface (Java), 290

S

SAM (single abstract method) conversions,
 149
save method, 225

- SAX parser, 224
- scala package, 157
 - always imported, 76, 80–81, 104
- Scala programming language
 - embedded languages in, 131, 269
 - interoperating with:
 - Java, 37, 44–45, 52, 89, 125–126, 175–177, 189, 200–206
 - shell programs, 105
 - interpreter of, 1–3
 - older versions of, 69, 103
- scala/bin directory, 1
- scala.collection package, 157, 176
- scala.collection.JavaConversions package, 189
- scala.math package, 7, 10
- scala.sys.process package, 105
- scala.tools.nsc.io package, 103
- scala.util package, 7
- Scaladoc, 5, 8–11, 35–36, 206
- ScalaObject interface, 95
- scanLeft method, 171
- Scanner class (Java), 46, 101
- scanRight method, 171
- sealed keyword, 193–194
- segmentLength method, 166, 173
- self types, 62, 124–125
 - dependency injections in, 256–257
 - no automatic inheritance for, 255
 - structural types in, 125
 - typesafe, 260
 - vs. traits with supertypes, 125
- Seq trait, 22, 35, 156, 237
 - important methods of, 166
- Seq[Char] class, 10
- Seq[Node] class, 214, 216
- seqAsJavalist function, 176
- sequences
 - adding/removing elements of, 164
 - comparing, 150, 237
 - extracting values from, 138–139
 - filtering, 147
 - immutable, 158–159
 - integer, 158
 - mutable, 159
 - of characters, 10
 - reversing, 167
 - sorting, 148, 167
 - with fast random access, 158
 - ser method, 178
- Serializable trait, 104, 204
- Serializable interface (Java), 114
- @serializable annotation, 204
- serialization, 104
- @SerialVersionUID annotation, 104, 204
- Set trait, 156–157
- setAsJavaSet function, 176
- sets, 161–162
 - adding/removing elements of, 163–164
 - difference of, 162–163
 - finding elements in, 161
 - hash. *See* hash sets
 - intersection of, 162–163
 - order of elements in, 161
 - sorted. *See* sorted sets
 - union of, 162–163
- @setter annotation, 203
- setXxx methods, 52, 55, 205
- shared states, 289, 301
- shell scripts, 105–106
- shift method, 320–335
 - with type parameters, 323–325
- Short type, 4, 17
- singleton objects, 7, 65–66, 247
 - case objects for, 189
 - vs. classes, 7
- singleton types, 246–247, 249, 253
- slice method, 165, 173
- sliding method, 166, 172–173
- SmallTalk programming language, 54
- Some class, 194–195, 272–273
- sortBy method, 167, 173
- sorted method, 34, 167, 173
- sorted sets, 162
- SortedMap trait, 156
- SortedSet trait, 156
- sortWith method, 148, 167, 173
- Source object, 100–102
- span method, 165, 173
- @specialized annotation, 209–210
- splitAt method, 165, 173
- Spring framework, 256
- sqrt method, 7, 308

- Stack class, 158–159
 - stack overflow, 206
 - standard input, 102
 - StandardTokenParsers class, 283
 - start method, 290, 299
 - start symbol, 271–272
 - startsWith method, 166, 173
 - statements
 - and line breaks, 16
 - terminating, 15–16
 - vs. expressions, 13
 - StaticAnnotation trait, 202
 - stdin method, 102
 - StdLexical trait, 284–285
 - StdTokenParsers trait, 281, 284
 - StdTokens trait, 283
 - Stream class, 158
 - streams, 173–174
 - @strictfp annotation, 203–204
 - String class, 102, 106
 - stringLit method, 284
 - StringOps class, 5, 10, 46
 - strings, 5
 - characters in:
 - common, 5
 - distinct, 7
 - uppercase, 10
 - classes for, 10
 - converting:
 - from any objects, 5
 - to numbers, 8, 101
 - to ProcessBuilder objects, 105
 - parsing, 278, 283
 - traversing, 18
 - vs. symbols, 210
 - structural types, 91, 125, 250
 - adding to compound types, 251
 - subclasses
 - anonymous, 91
 - concrete, 92
 - equality in, 96
 - implementing abstract methods in, 113
 - subsetOf method, 162
 - success method, 279
 - sum method, 34, 165, 173
 - super keyword, 86–87, 117
 - super keyword (Java), 89
 - superclasses, 123–124
 - abstract fields in, 92
 - constructing, 88–89
 - extending, 126
 - methods of:
 - abstract, 91
 - new, 86
 - overriding, 90
 - no multiple inheritance of, 111, 114, 119
 - scope of, 248
 - sealed, 193–194
 - supertypes, 14, 36
 - supervisors, 300–302
 - @suspendable annotation, 325
 - Swing toolkit, 127–128
 - switch statement, 15, 184
 - @switch annotation, 207–208
 - Symbol class, 202
 - symbols, 210
 - synchronized method, 95
 - SynchronizedBuffer trait, 177
 - SynchronizedMap trait, 177
 - SynchronizedPriorityQueue trait, 177
 - SynchronizedQueue trait, 177
 - SynchronizedSet trait, 177
 - SynchronizedStack trait, 177
 - syntactic sugar, 241, 252
- ## T
- tab completion, 2
 - tail method, 159–160, 165
 - TailCalls object, 207
 - TailRec object, 207
 - @tailrec annotation, 207
 - take method, 165, 173
 - takeRight method, 165
 - takeWhile method, 165, 173
 - @Test annotation, 201
 - text method, 216
 - Text class, 217
 - pattern matching for, 221
 - this keyword, 36, 53, 59, 88, 124–125, 246, 260
 - aliases for, 62, 255
 - scope of, 248

- threads
 - blocking, 331
 - sharing, 296–299, 302
- throw expression, 25
- Throwable class (Java), 24
- @throws annotation, 204
- TIMEOUT object, 296
- to method, 5, 18, 159
- toArray method, 31, 100, 166, 173
- toBuffer method, 31, 100
- toChar method, 5
- toDouble method, 5, 101
- toIndexedSeq method, 166, 173
- toInt method, 5, 101
- toIterable method, 166, 173
- token method, 284–285
- Token type, 281
- tokens, 270
 - discarding, 274–275
 - matching against regexes, 271
- Tokens trait, 283
- toList method, 166, 173
- toMap method, 46, 166, 173
- toSeq method, 166, 173
- toSet method, 166, 173
- toStream method, 166, 173
- toString method, 5, 34, 70, 190, 193, 217
- trait keyword, 113, 253
- traits, 113–126, 253
 - abstract types in, 257
 - adding to objects, 115
 - construction order of, 116–117, 120–122
 - dependencies in, 125, 256–257
 - extending, 67
 - classes, 123–124
 - other traits, 115–119, 122–123
 - superclass, 126
 - fields in:
 - abstract, 119–120, 122
 - concrete, 118–119
 - for collections, 156–157
 - for rich interfaces, 118
 - implementing, 114, 235
 - layered, 116–117
 - methods in, 114–115, 125
 - overriding, 117
 - unimplemented, 113
 - parameterless constructors of, 122–123
 - type parameters in, 232
 - vs. classes, 122
 - vs. Java interfaces, 111–114, 125
 - vs. structural types, 250
- trampolining, 207
- transform method, 223
- @transient annotation, 203
- Traversable trait, 35
- TraversableOnce trait, 35
- TreeMap class (Java), 44
- TreeSet class (Java), 162
- trees, 326–329
- trimEnd method, 30
- try statement, 25–26
 - exceptions in, 152
- tuples, 41, 45–46, 253
 - accessing components of, 45
 - converting to maps, 46
 - pattern matching for, 187–188
 - zipping, 46
- type keyword, 246–247, 249, 253
- type constraints, 312–313
- type constructors, 263–265
- type parameters, 91, 231–241, 253, 258, 310
 - annotated, 201
 - bounds for, 232–235
 - context bounds of, 311–312
 - implicit conversions for, 234
 - infix notation for, 251–252
 - not possible for objects, 240
 - structural, 250
 - with continuations, 323–325
- type projections, 62, 247–249, 253
 - in forSome blocks, 253
- types, 4–5
 - abstract, 257, 281
 - bounds for, 259
 - made concrete in subclass, 249, 257
 - aliases for, 157
 - annotated, 201
 - anonymous, 92
 - checking, 87–88

- constraints of, 236–237
 - converting between, 5
 - enriched, 306–307
 - equality of, 236
 - errors in, 239
 - existential, 241
 - generic, 235, 241
 - implementing multiple traits, 235
 - inference of, 236–237
 - invariant, 241
 - matching by, 186–187
 - naming, 262
 - primitive, 4, 30, 209
 - subtypes of, 236
 - variance of, 237–241
 - view-convertible, 236
 - wrapper, 4
- U**
- u1 (XML), 218
 - unapply method, 136–138, 188, 190–191
 - unapplySeq method, 138–139, 188
 - unary_ methods, 133
 - UncaughtException, 300
 - @unchecked annotation, 210
 - @uncheckedVariance annotation, 210–211
 - Unicode characters, 132
 - uniform access principle, 53
 - uniform creation principle, 157
 - uniform return type principle, 167
 - union method, 162
 - Unit class, 23, 94–95, 320, 323, 326, 335–336
 - value of, 14–15, 17
 - Unparsed type, 219
 - until method, 18, 31, 151–152, 159
 - update method, 135–136
 - URIs (Uniform Resource Identifiers), 226
 - URLs (Uniform Resource Locators)
 - loading files from, 223
 - reading from, 102
 - redirecting input from, 106
- V**
- val fields, 3
 - declarations of, 3–4
 - early definitions of, 93
 - final, 93
 - generated methods for, 53, 56, 59
 - in forSome blocks, 253
 - in parsers, 280
 - initializing, 3, 16, 23–24
 - lazy, 23–24, 93, 123, 280
 - overriding, 89–90, 92
 - private, 56
 - scope of, 248
 - specifying type of, 3
 - storing functions in, 143–144
 - Value method, 69–70
 - value classes, 193
 - valueAtOneQuarter method, 146
 - values
 - binding to variables, 192
 - naming, 186
 - printing, 17
 - values method, 44
 - var fields, 3
 - annotated, 200
 - declarations of, 3–4
 - extractors in, 136
 - pattern matching in, 188–189
 - generated methods for, 56, 59
 - initializing, 3
 - no path elements in, 249
 - overriding, 90
 - private, 56
 - specifying type of, 4, 232
 - updating, 43
 - vs. function calls, in parsers, 279
 - @varargs annotation, 205
 - variables
 - binding to values, 192
 - in case clauses, 185
 - naming, 131–132, 186
 - vector type (C++), 30
 - Vector class, 158
 - view method, 174–175
 - view bounds, 234–235
 - void keyword (C++, Java), 15, 17, 95
 - @volatile annotation, 203

W

- wait method, 95
- walkFileTree method (Java), 103–104
- web applications, 329–332
- while loop, 18, 151
 - annotated as CPS, 327
- whitespace
 - in lexical analysis, 270
 - parsing, 224, 282–283
- wildcards
 - for XML elements, 220
 - in catch statements, 25
 - in imports, 7, 79–80
 - in Java, 79, 241, 252
- with keyword, 93, 114–115, 235, 250–251, 253
- wrapper types, 4
- Writer class (Java), 225

X

- Xcheckinit compiler flag, 93
- Xelide-below compiler flag, 208–209
- XHTML (Extensible Hypertext Markup Language), 219
- XhtmlParser class, 225
- XML (Extensible Markup Language), 213–227
 - attributes in, 216–219, 222–223
 - character references in, 216
 - comments in, 215

- elements in, 222–223, 226
- entity references in, 215–216, 225
- including non-XML text into, 219
- loading, 223
- malformed, 219
- namespaces in, 226–227
- nodes in, 214–216
- processing instructions in, 215
- saving, 217, 225–226
- self-closing tags in, 226
- transforming, 223
- XML declarations, 225
- XML literals, 214
 - braces in, 218
 - embedded expressions in, 217–218
 - entity references in, 216
 - in pattern matching, 221–222
- XPath (XML Path language), 220–221
- Xprint compiler flag, 309, 334

Y

- yield keyword
 - as Java method, 132
 - in loops, 20, 32, 178

Z

- zip method, 46, 165–173
- zipAll method, 165, 172–173
- zipWithIndex method, 165, 172–173