

Covers **Scala 2.12**



SCALA

for the Impatient

Second Edition

Cay S. Horstmann

Foreword by Martin Odersky



FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Scala for the Impatient

Second Edition

This page intentionally left blank

Scala for the Impatient

Second Edition

Cay S. Horstmann

◆ Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016954825

Copyright © 2017 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. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

ISBN-13: 978-0-13-454056-6

ISBN-10: 0-13-454056-5

Text printed in the United States of America.

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

This page intentionally left blank

Contents

Foreword to the First Edition	xvii
Preface	xix
About the Author	xxi

1	THE BASICS A1	1
1.1	The Scala Interpreter	1
1.2	Declaring Values and Variables	4
1.3	Commonly Used Types	5
1.4	Arithmetic and Operator Overloading	6
1.5	More about Calling Methods	8
1.6	The <code>apply</code> Method	9
1.7	Scaladoc	10
	Exercises	15
2	CONTROL STRUCTURES AND FUNCTIONS A1	17
2.1	Conditional Expressions	18
2.2	Statement Termination	19
2.3	Block Expressions and Assignments	20

2.4	Input and Output	21
2.5	Loops	22
2.6	Advanced for Loops	24
2.7	Functions	25
2.8	Default and Named Arguments L1	26
2.9	Variable Arguments L1	26
2.10	Procedures	28
2.11	Lazy Values L1	28
2.12	Exceptions	29
	Exercises	31

3 WORKING WITH ARRAYS **A1** 35

3.1	Fixed-Length Arrays	35
3.2	Variable-Length Arrays: Array Buffers	36
3.3	Traversing Arrays and Array Buffers	37
3.4	Transforming Arrays	38
3.5	Common Algorithms	40
3.6	Deciphering Scaladoc	41
3.7	Multidimensional Arrays	42
3.8	Interoperating with Java	43
	Exercises	44

4 MAPS AND TUPLES **A1** 47

4.1	Constructing a Map	48
4.2	Accessing Map Values	48
4.3	Updating Map Values	49
4.4	Iterating over Maps	50
4.5	Sorted Maps	50
4.6	Interoperating with Java	50
4.7	Tuples	51
4.8	Zippping	52
	Exercises	52

5	CLASSES A1	55
5.1	Simple Classes and Parameterless Methods	55
5.2	Properties with Getters and Setters	56
5.3	Properties with Only Getters	59
5.4	Object-Private Fields	60
5.5	Bean Properties L1	61
5.6	Auxiliary Constructors	62
5.7	The Primary Constructor	63
5.8	Nested Classes L1	66
	Exercises	68
6	OBJECTS A1	71
6.1	Singletons	71
6.2	Companion Objects	72
6.3	Objects Extending a Class or Trait	73
6.4	The apply Method	73
6.5	Application Objects	74
6.6	Enumerations	75
	Exercises	77
7	PACKAGES AND IMPORTS A1	79
7.1	Packages	80
7.2	Scope Rules	81
7.3	Chained Package Clauses	83
7.4	Top-of-File Notation	83
7.5	Package Objects	83
7.6	Package Visibility	84
7.7	Imports	85
7.8	Imports Can Be Anywhere	85
7.9	Renaming and Hiding Members	86
7.10	Implicit Imports	86
	Exercises	87

- 8** **INHERITANCE** **A1** 91
 - 8.1 Extending a Class 91
 - 8.2 Overriding Methods 92
 - 8.3 Type Checks and Casts 93
 - 8.4 Protected Fields and Methods 94
 - 8.5 Superclass Construction 94
 - 8.6 Overriding Fields 95
 - 8.7 Anonymous Subclasses 97
 - 8.8 Abstract Classes 97
 - 8.9 Abstract Fields 97
 - 8.10 Construction Order and Early Definitions **L3** 98
 - 8.11 The Scala Inheritance Hierarchy 100
 - 8.12 Object Equality **L1** 102
 - 8.13 Value Classes **L2** 103
 - Exercises 105

- 9** **FILES AND REGULAR EXPRESSIONS** **A1** 109
 - 9.1 Reading Lines 109
 - 9.2 Reading Characters 110
 - 9.3 Reading Tokens and Numbers 111
 - 9.4 Reading from URLs and Other Sources 111
 - 9.5 Reading Binary Files 112
 - 9.6 Writing Text Files 112
 - 9.7 Visiting Directories 112
 - 9.8 Serialization 113
 - 9.9 Process Control **A2** 114
 - 9.10 Regular Expressions 116
 - 9.11 Regular Expression Groups 117
 - Exercises 118

- 10** **TRAITS** **L1** 121
 - 10.1 Why No Multiple Inheritance? 121
 - 10.2 Traits as Interfaces 123
 - 10.3 Traits with Concrete Implementations 124

10.4	Objects with Traits	125
10.5	Layered Traits	125
10.6	Overriding Abstract Methods in Traits	127
10.7	Traits for Rich Interfaces	127
10.8	Concrete Fields in Traits	128
10.9	Abstract Fields in Traits	130
10.10	Trait Construction Order	130
10.11	Initializing Trait Fields	132
10.12	Traits Extending Classes	133
10.13	Self Types L2	134
10.14	What Happens under the Hood	135
	Exercises	137

11 OPERATORS **L1** 141

11.1	Identifiers	142
11.2	Infix Operators	143
11.3	Unary Operators	143
11.4	Assignment Operators	144
11.5	Precedence	144
11.6	Associativity	145
11.7	The <code>apply</code> and <code>update</code> Methods	146
11.8	Extractors L2	147
11.9	Extractors with One or No Arguments L2	149
11.10	The <code>unapplySeq</code> Method L2	149
11.11	Dynamic Invocation L2	150
	Exercises	153

12 HIGHER-ORDER FUNCTIONS **L1** 157

12.1	Functions as Values	157
12.2	Anonymous Functions	159
12.3	Functions with Function Parameters	160
12.4	Parameter Inference	160
12.5	Useful Higher-Order Functions	161
12.6	Closures	162

- 12.7 SAM Conversions 163
- 12.8 Currying 164
- 12.9 Control Abstractions 166
- 12.10 The return Expression 167
- Exercises 168

13 COLLECTIONS **A2** 171

- 13.1 The Main Collections Traits 172
- 13.2 Mutable and Immutable Collections 173
- 13.3 Sequences 174
- 13.4 Lists 175
- 13.5 Sets 177
- 13.6 Operators for Adding or Removing Elements 178
- 13.7 Common Methods 180
- 13.8 Mapping a Function 182
- 13.9 Reducing, Folding, and Scanning **A3** 184
- 13.10 Zipping 187
- 13.11 Iterators 188
- 13.12 Streams **A3** 189
- 13.13 LazyViews **A3** 190
- 13.14 Interoperability with Java Collections 191
- 13.15 Parallel Collections 193
- Exercises 194

14 PATTERN MATCHING AND CASE CLASSES **A2** 197

- 14.1 A Better Switch 198
- 14.2 Guards 199
- 14.3 Variables in Patterns 199
- 14.4 Type Patterns 200
- 14.5 Matching Arrays, Lists, and Tuples 201
- 14.6 Extractors 202
- 14.7 Patterns in Variable Declarations 203
- 14.8 Patterns in for Expressions 204
- 14.9 Case Classes 205

- 14.10 The copy Method and Named Parameters 205
- 14.11 Infix Notation in case Clauses 206
- 14.12 Matching Nested Structures 207
- 14.13 Are Case Classes Evil? 208
- 14.14 Sealed Classes 209
- 14.15 Simulating Enumerations 209
- 14.16 The Option Type 210
- 14.17 Partial Functions **L2** 211
 - Exercises 212

15 ANNOTATIONS **A2** 215

- 15.1 What Are Annotations? 216
- 15.2 What Can Be Annotated? 216
- 15.3 Annotation Arguments 217
- 15.4 Annotation Implementations 218
- 15.5 Annotations for Java Features 219
 - 15.5.1 Java Modifiers 219
 - 15.5.2 Marker Interfaces 220
 - 15.5.3 Checked Exceptions 220
 - 15.5.4 Variable Arguments 221
 - 15.5.5 JavaBeans 221
- 15.6 Annotations for Optimizations 222
 - 15.6.1 Tail Recursion 222
 - 15.6.2 Jump Table Generation and Inlining 223
 - 15.6.3 Eliding Methods 224
 - 15.6.4 Specialization for Primitive Types 225
- 15.7 Annotations for Errors and Warnings 226
 - Exercises 227

16 XML PROCESSING **A2** 229

- 16.1 XML Literals 230
- 16.2 XML Nodes 230
- 16.3 Element Attributes 232
- 16.4 Embedded Expressions 233

- 16.5 Expressions in Attributes 234
- 16.6 Uncommon Node Types 235
- 16.7 XPath-like Expressions 235
- 16.8 Pattern Matching 237
- 16.9 Modifying Elements and Attributes 238
- 16.10 Transforming XML 239
- 16.11 Loading and Saving 239
- 16.12 Namespaces 242
 - Exercises 243

17 FUTURES **A2** 247

- 17.1 Running Tasks in the Future 248
- 17.2 Waiting for Results 250
- 17.3 The Try Class 251
- 17.4 Callbacks 251
- 17.5 Composing Future Tasks 252
- 17.6 Other Future Transformations 255
- 17.7 Methods in the Future Object 256
- 17.8 Promises 258
- 17.9 Execution Contexts 260
 - Exercises 260

18 TYPE PARAMETERS **L2** 265

- 18.1 Generic Classes 266
- 18.2 Generic Functions 266
- 18.3 Bounds for Type Variables 266
- 18.4 View Bounds 268
- 18.5 Context Bounds 268
- 18.6 The ClassTag Context Bound 269
- 18.7 Multiple Bounds 269
- 18.8 Type Constraints **L3** 269
- 18.9 Variance 271
- 18.10 Co- and Contravariant Positions 272

- 18.11 Objects Can't Be Generic 274
- 18.12 Wildcards 275
 - Exercises 275

19 ADVANCED TYPES **L2** 279

- 19.1 Singleton Types 280
- 19.2 Type Projections 281
- 19.3 Paths 282
- 19.4 Type Aliases 283
- 19.5 Structural Types 283
- 19.6 Compound Types 284
- 19.7 Infix Types 285
- 19.8 Existential Types 286
- 19.9 The Scala Type System 287
- 19.10 Self Types 288
- 19.11 Dependency Injection 289
- 19.12 Abstract Types **L3** 291
- 19.13 Family Polymorphism **L3** 293
- 19.14 Higher-Kinded Types **L3** 296
 - Exercises 299

20 PARSING **A3** 303

- 20.1 Grammars 304
- 20.2 Combining Parser Operations 305
- 20.3 Transforming Parser Results 307
- 20.4 Discarding Tokens 308
- 20.5 Generating Parse Trees 309
- 20.6 Avoiding Left Recursion 310
- 20.7 More Combinators 311
- 20.8 Avoiding Backtracking 314
- 20.9 Packrat Parsers 314
- 20.10 What Exactly Are Parsers? 315
- 20.11 Regex Parsers 316

20.12	Token-Based Parsers	317
20.13	Error Handling	319
	Exercises	320
21	IMPLICITS L3	323
21.1	Implicit Conversions	324
21.2	Using Implicits for Enriching Existing Classes	324
21.3	Importing Implicits	325
21.4	Rules for Implicit Conversions	326
21.5	Implicit Parameters	328
21.6	Implicit Conversions with Implicit Parameters	329
21.7	Context Bounds	329
21.8	Type Classes	331
21.9	Evidence	333
21.10	The <code>@implicitNotFound</code> Annotation	334
21.11	<code>CanBuildFrom</code> Demystified	334
	Exercises	336
	Index	338

Foreword to the First Edition

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 traditional languages such as Java, C#, 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 improve their productivity. 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. And Scala doesn't just target the Java virtual machine. The ScalaJS project emits JavaScript code, enabling you to write both the server-side and client-side parts of a web application in a language that isn't JavaScript. Scala 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.

This is the second edition of this book, and I updated it thoroughly for Scala 2.12. I added coverage of recent Scala features such as string interpolation, dynamic invocation, implicit classes, and futures, and updated all chapters to reflect current Scala usage.

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, Steve Haines, Susan Potter, 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, 2016

About the Author

Cay S. Horstmann is author of *Core Java™, Volumes I & II, Tenth Edition* (Prentice Hall, 2016), 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.

Classes

Topics in This Chapter **A1**

- 5.1 Simple Classes and Parameterless Methods — page 55
- 5.2 Properties with Getters and Setters — page 56
- 5.3 Properties with Only Getters — page 59
- 5.4 Object-Private Fields — page 60
- 5.5 Bean Properties **L1** — page 61
- 5.6 Auxiliary Constructors — page 62
- 5.7 The Primary Constructor — page 63
- 5.8 Nested Classes **L1** — page 66
- Exercises — page 68

Chapter

5

In this chapter, you will learn how to implement classes in Scala. If you know classes in Java or C++, you won't find this difficult, and you will enjoy the much more concise notation of Scala.

The key points of this chapter are:

- Fields in classes automatically come with getters and setters.
- You can replace a field with a custom getter/setter without changing the client of a class—that is the “uniform access principle.”
- Use the `@BeanProperty` annotation to generate the JavaBeans `getXxx/setXxx` methods.
- Every class has a primary constructor that is “interwoven” with the class definition. Its parameters turn into the fields of the class. The primary constructor executes all statements in the body of the class.
- Auxiliary constructors are optional. They are called this.

5.1 Simple Classes and Parameterless Methods

In its simplest form, a Scala class looks very much like its equivalent in Java or C++:


```
class Counter {
  private var value = 0 // You must initialize the field
  def increment() { value += 1 } // Methods are public by default
  def current() = value
}
```

In Scala, a class is not declared as `public`. A Scala source file can contain multiple classes, and all of them have public visibility.

To use this class, you construct objects and invoke methods in the usual way:

```
val myCounter = new Counter // Or new Counter()
myCounter.increment()
println(myCounter.current)
```

You can call a parameterless method (such as `current`) with or without parentheses:

```
myCounter.current // OK
myCounter.current() // Also OK
```

Which form should you use? It is considered good style to use `()` for a *mutator* method (a method that changes the object state), and to drop the `()` for an *accessor* method (a method that does not change the object state).

That's what we did in our example:

```
myCounter.increment() // Use () with mutator
println(myCounter.current) // Don't use () with accessor
```

You can enforce this style by declaring `current` without `()`:

```
class Counter {
  ...
  def current = value // No () in definition
}
```

Now class users must use `myCounter.current`, without parentheses.

5.2 Properties with Getters and Setters

When writing a Java class, we don't like to use public fields:

```
public class Person { // This is Java
  public int age; // Frowned upon in Java
}
```

With a public field, anyone could write to `fred.age`, making Fred younger or older. That's why we prefer to use getter and setter methods:

```
public class Person { // This is Java
    private int age;
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

A getter/setter pair such as this one is often called a *property*. We say that the class `Person` has an age property.

Why is this any better? By itself, it isn't. Anyone can call `fred.setAge(21)`, keeping him forever twenty-one.

But if that becomes a problem, we can guard against it:

```
public void setAge(int newValue) { if (newValue > age) age = newValue; }
// Can't get younger
```

Getters and setters are better than public fields because they let you start with simple get/set semantics and evolve them as needed.



NOTE: Just because getters and setters are better than public fields doesn't mean they are always good. Often, it is plainly bad if every client can get or set bits and pieces of an object's state. In this section, I show you how to implement properties in Scala. It is up to you to choose wisely when a gettable/settable property is an appropriate design.

Scala provides getter and setter methods for every field. Here, we define a public field:

```
class Person {
    var age = 0
}
```

Scala generates a class for the JVM with a *private* `age` field and getter and setter methods. These methods are public because we did not declare `age` as private. (For a private field, the getter and setter methods are private.)

In Scala, the getter and setter methods are called `age` and `age_`. For example,

```
println(fred.age) // Calls the method fred.age()
fred.age = 21 // Calls fred.age_(21)
```

In Scala, the getters and setters are not named `getXxx` and `setXxx`, but they fulfill the same purpose. Section 5.5, “Bean Properties,” on page 61 shows how to generate Java-style `getXxx` and `setXxx` methods, so that your Scala classes can interoperate with Java tools.



NOTE: To see these methods with your own eyes, compile the `Person` class and then look at the bytecode with `javap`:

```
$ scalac Person.scala
$ javap -private Person
Compiled from "Person.scala"
public class Person extends java.lang.Object implements scala.ScalaObject{
    private int age;
    public int age(); public void age_$eq(int);
    public Person();
}
```

As you can see, the compiler created methods `age` and `age_$eq`. (The `=` symbol is translated to `$eq` because the JVM does not allow an `=` in a method name.)



TIP: You can run the `javap` command inside the REPL as

```
:javap -private Person
```

At any time, you can redefine the getter and setter methods yourself. For example,

```
class Person {
    private var privateAge = 0 // Make private and rename

    def age = privateAge
    def age_=(newValue: Int) {
        if (newValue > privateAge) privateAge = newValue; // Can't get younger
    }
}
```

The user of your class still accesses `fred.age`, but now Fred can't get younger:

```
val fred = new Person
fred.age = 30
fred.age = 21
println(fred.age) // 30
```



NOTE: Bertrand Meyer, the inventor of the influential Eiffel language, formulated the *Uniform Access Principle* that states: "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation." In Scala, the caller of `fred.age` doesn't know whether `age` is implemented through a field or a method. (Of course, in the JVM, the service is *always* implemented through a method, either synthesized or programmer-supplied.)



TIP: It may sound scary that Scala generates getter and setter methods for every field. But you have some control over this process.

- If the field is private, the getter and setter are private.
- If the field is a `val`, only a getter is generated.
- If you don't want any getter or setter, declare the field as `private[this]` (see Section 5.4, “Object-Private Fields,” on page 60).

5.3 Properties with Only Getters

Sometimes you want a *read-only property* with a getter but no setter. If the value of the property never changes after the object has been constructed, use a `val` field:

```
class Message {  
  val timeStamp = java.time.Instant.now  
  ...  
}
```

The Scala compiler produces a Java class with a private `final` field and a public getter method, but no setter.

Sometimes, however, you want a property that a client can't set at will, but that is mutated in some other way. The `Counter` class from Section 5.1, “Simple Classes and Parameterless Methods,” on page 55 is a good example. Conceptually, the counter has a current property that is updated when the `increment` method is called, but there is no setter for the property.

You can't implement such a property with a `val`—a `val` never changes. Instead, provide a private field and a property getter, like this:

```
class Counter {  
  private var value = 0  
  def increment() { value += 1 }  
  def current = value // No () in declaration  
}
```

Note that there are no `()` in the definition of the getter method. Therefore, you *must* call the method without parentheses:

```
val n = myCounter.current // Calling myCounter.current() is a syntax error
```

To summarize, you have four choices for implementing properties:

1. `var foo`: Scala synthesizes a getter and a setter.
2. `val foo`: Scala synthesizes a getter.

3. You define methods `foo` and `foo_=`.
4. You define a method `foo`.



NOTE: In Scala, you cannot have a write-only property (that is, a property with a setter and no getter).



TIP: When you see a field in a Scala class, remember that it is not the same as a field in Java or C++. It is a private field *together with* a getter (for a `val` field) or a getter and a setter (for a `var` field).

5.4 Object-Private Fields

In Scala (as well as in Java or C++), a method can access the private fields of *all* objects of its class. For example,

```
class Counter {
  private var value = 0
  def increment() { value += 1 }

  def isLess(other : Counter) = value < other.value
    // Can access private field of other object
}
```

Accessing `other.value` is legal because `other` is also a `Counter` object.

Scala allows an even more severe access restriction with the `private[this]` qualifier:

```
private[this] var value = 0 // Accessing someObject.value is not allowed
```

Now, the methods of the `Counter` class can only access the `value` field of the current object, not of other objects of type `Counter`. This access is sometimes called *object-private*, and it is common in some OO languages such as SmallTalk.

With a class-private field, Scala generates private getter and setter methods. However, for an object-private field, no getters and setters are generated at all.



NOTE: Scala allows you to grant access rights to specific classes. The `private[ClassName]` qualifier states that only methods of the given class can access the given field. Here, the `ClassName` must be the name of the class being defined or an enclosing class. (See Section 5.8, “Nested Classes,” on page 66 for a discussion of inner classes.)

In this case, the implementation will generate auxiliary getter and setter methods that allow the enclosing class to access the field. These methods will be public because the JVM does not have a fine-grained access control system, and they will have implementation-dependent names.

5.5 Bean Properties

As you saw in the preceding sections, Scala provides getter and setter methods for the fields that you define. However, the names of these methods are not what Java tools expect. The JavaBeans specification (www.oracle.com/technetwork/articles/javase/spec-136004.html) defines a Java property as a pair of `getFoo/setFoo` methods (or just a `getFoo` method for a read-only property). Many Java tools rely on this naming convention.

When you annotate a Scala field with `@BeanProperty`, then such methods are automatically generated. For example,

```
import scala.beans.BeanProperty

class Person {
  @BeanProperty var name: String = _
}
```

generates *four* methods:

1. `name: String`
2. `name_=(newValue: String): Unit`
3. `getName(): String`
4. `setName(newValue: String): Unit`

Table 5–1 shows which methods are generated in all cases.

Table 5–1 Generated Methods for Fields

Scala Field	Generated Methods	When to Use
<code>val/var name</code>	<code>public name</code> <code>name_= (var only)</code>	To implement a property that is publicly accessible and backed by a field.
<code>@BeanProperty val/var name</code>	<code>public name</code> <code>getName()</code> <code>name_= (var only)</code> <code>setName(...)</code> (var only)	To interoperate with JavaBeans.
<code>private val/var name</code>	<code>private name</code> <code>name_= (var only)</code>	To confine the field to the methods of this class, just like in Java. Use private unless you really want a public property.
<code>private[this] val/var name</code>	none	To confine the field to methods invoked on the same object. Not commonly used.
<code>private[ClassName] val/var name</code>	implementation-dependent	To grant access to an enclosing class. Not commonly used.



NOTE: If you define a field as a primary constructor parameter (see Section 5.7, “The Primary Constructor,” on page 63), and you want JavaBeans getters and setters, annotate the constructor parameter like this:

```
class Person(@BeanProperty var name: String)
```

5.6 Auxiliary Constructors

As in Java or C++, a Scala class can have as many constructors as you like. However, a Scala class has one constructor that is more important than all the others, called the *primary constructor*. In addition, a class may have any number of *auxiliary constructors*.

We discuss auxiliary constructors first because they are easier to understand. They are similar to constructors in Java or C++, with just two differences.

1. The auxiliary constructors are called *this*. (In Java or C++, constructors have the same name as the class—which is not so convenient if you rename the class.)
2. Each auxiliary constructor *must* start with a call to a previously defined auxiliary constructor or the primary constructor.

Here is a class with two auxiliary constructors:

```
class Person {
  private var name = ""
  private var age = 0

  def this(name: String) { // An auxiliary constructor
    this() // Calls primary constructor
    this.name = name
  }

  def this(name: String, age: Int) { // Another auxiliary constructor
    this(name) // Calls previous auxiliary constructor
    this.age = age
  }
}
```

We will look at the primary constructor in the next section. For now, it is sufficient to know that a class for which you don't define a primary constructor has a primary constructor with no arguments.

You can construct objects of this class in three ways:

```
val p1 = new Person // Primary constructor
val p2 = new Person("Fred") // First auxiliary constructor
val p3 = new Person("Fred", 42) // Second auxiliary constructor
```

5.7 The Primary Constructor

In Scala, every class has a primary constructor. The primary constructor is not defined with a `this` method. Instead, it is interwoven with the class definition.

1. The parameters of the primary constructor are placed *immediately after the class name*.

```
class Person(val name: String, val age: Int) {
  // Parameters of primary constructor in (...)
  ...
}
```

Parameters of the primary constructor turn into fields that are initialized with the construction parameters. In our example, `name` and `age` become fields of the `Person` class. A constructor call such as `new Person("Fred", 42)` sets the `name` and `age` fields.

Half a line of Scala is the equivalent of seven lines of Java:


```
public class Person { // This is Java
    private String name; private int age; public Person(String name, int age) {
        this.name = name; this.age = age;
    }
    public String name() { return this.name; } public int age() { return this.age; }
    ...
}
```

2. The primary constructor executes *all statements in the class definition*. For example, in the following class

```
class Person(val name: String, val age: Int) {
    println("Just constructed another person")
    def description = s"$name is $age years old"
}
```

the `println` statement is a part of the primary constructor. It is executed whenever an object is constructed.

This is useful when you need to configure a field during construction. For example:

```
class MyProg {
    private val props = new Properties
    props.load(new FileReader("myprog.properties"))
    // The statement above is a part of the primary constructor
    ...
}
```



NOTE: If there are no parameters after the class name, then the class has a primary constructor with no parameters. That constructor simply executes all statements in the body of the class.



TIP: You can often eliminate auxiliary constructors by using default arguments in the primary constructor. For example:

```
class Person(val name: String = "", val age: Int = 0)
```

Primary constructor parameters can have any of the forms in Table 5–1. For example,

```
class Person(val name: String, private var age: Int)
```

declares and initializes fields

```
val name: String
private var age: Int
```

Construction parameters can also be regular method parameters, without `val` or `var`. How these parameters are processed depends on their usage inside the class.

- If a parameter without `val` or `var` is used inside at least one method, it becomes a field. For example,

```
class Person(name: String, age: Int) {
  def description = s"$name is $age years old"
}
```

declares and initializes immutable fields `name` and `age` that are object-private.

Such a field is the equivalent of a `private[this] val` field (see Section 5.4, “Object-Private Fields,” on page 60).

- Otherwise, the parameter is not saved as a field. It’s just a regular parameter that can be accessed in the code of the primary constructor. (Strictly speaking, this is an implementation-specific optimization.)

Table 5–2 summarizes the fields and methods that are generated for different kinds of primary constructor parameters.

Table 5–2 Fields and Methods Generated for Primary Constructor Parameters

Primary Constructor Parameter	Generated Field/Methods
<code>name: String</code>	object-private field, or no field if no method uses <code>name</code>
<code>private val/var name: String</code>	private field, private getter/setter
<code>val/var name: String</code>	private field, public getter/setter
<code>@BeanProperty val/var name: String</code>	private field, public Scala and JavaBeans getters/setters

If you find the primary constructor notation confusing, you don’t need to use it. Just provide one or more auxiliary constructors in the usual way, but remember to call `this()` if you don’t chain to another auxiliary constructor.

However, many programmers like the concise syntax. Martin Odersky suggests to think about it this way: In Scala, classes take parameters, just like methods do.



NOTE: When you think of the primary constructor’s parameters as class parameters, parameters without `val` or `var` become easier to understand. The scope of such a parameter is the entire class. Therefore, you can use the parameter in methods. If you do, it is the compiler’s job to save it in a field.



TIP: The Scala designers think that *every keystroke is precious*, so they let you combine a class with its primary constructor. When reading a Scala class, you need to disentangle the two. For example, when you see

```
class Person(val name: String) {
  var age = 0
  def description = s"$name is $age years old"
}
```

take this definition apart into a class definition:

```
class Person(val name: String) {
  var age = 0
  def description = s"$name is $age years old"
}
```

and a constructor definition:

```
class Person(val name: String) {
  var age = 0
  def description = s"$name is $age years old"
}
```



NOTE: To make the primary constructor private, place the keyword `private` like this:

```
class Person private(val id: Int) { ... }
```

A class user must then use an auxiliary constructor to construct a `Person` object.

5.8 Nested Classes **L1**

In Scala, you can nest just about anything inside anything. You can define functions inside other functions, and classes inside other classes. Here is a simple example of the latter:

```
import scala.collection.mutable.ArrayBuffer
class Network {
  class Member(val name: String) {
    val contacts = new ArrayBuffer[Member]
  }

  private val members = new ArrayBuffer[Member]

  def join(name: String) = {
    val m = new Member(name)
    members += m
  }
}
```

```

    m
  }
}

```

Consider two networks:

```

val chatter = new Network
val myFace = new Network

```

In Scala, each *instance* has its own class `Member`, just like each instance has its own field members. That is, `chatter.Member` and `myFace.Member` are *different classes*.



NOTE: This is different from Java, where an inner class belongs to the outer class.

The Scala approach is more regular. For example, to make a new inner object, you simply use `new` with the type name: `new chatter.Member`. In Java, you need to use a special syntax, `chatter.new Member()`.

In our network example, you can add a member within its own network, but not across networks.

```

val fred = chatter.join("Fred")
val wilma = chatter.join("Wilma")
fred.contacts += wilma // OK
val barney = myFace.join("Barney") // Has type myFace.Member
fred.contacts += barney
// No—can't add a myFace.Member to a buffer of chatter.Member elements

```

For networks of people, this behavior probably makes sense. If you don't want it, there are two solutions.

First, you can move the `Member` type somewhere else. A good place would be the `Network` companion object. (Companion objects are described in Chapter 6.)

```

object Network {
  class Member(val name: String) {
    val contacts = new ArrayBuffer[Member]
  }
}

class Network {
  private val members = new ArrayBuffer[Network.Member]
  ...
}

```

Alternatively, you can use a *type projection* `Network#Member`, which means “a `Member` of *any* `Network`.” For example,

```
class Network {
    class Member(val name: String) {
        val contacts = new ArrayBuffer[Network#Member]
    }
    ...
}
```

You would do that if you want the fine-grained “inner class per object” feature in some places of your program, but not everywhere. See Chapter 19 for more information about type projections.



NOTE: In a nested class, you can access the `this` reference of the enclosing class as `EnclosingClass.this`, like in Java. If you like, you can establish an alias for that reference with the following syntax:

```
class Network(val name: String) { outer =>
    class Member(val name: String) {
        ...
        def description = s"$name inside ${outer.name}"
    }
}
```

The class `Network { outer =>` syntax makes the variable `outer` refer to `Network.this`. You can choose any name for this variable. The name `self` is common, but perhaps confusing when used with nested classes.

This syntax is related to the “self type” syntax that you will see in Chapter 19.

Exercises

1. Improve the `Counter` class in Section 5.1, “Simple Classes and Parameterless Methods,” on page 55 so that it doesn’t turn negative at `Int.MaxValue`.
2. Write a class `BankAccount` with methods `deposit` and `withdraw`, and a read-only property `balance`.
3. Write a class `Time` with read-only properties `hours` and `minutes` and a method `before(other: Time): Boolean` that checks whether this time comes before the other. A `Time` object should be constructed as `new Time(hrs, min)`, where `hrs` is in military time format (between 0 and 23).
4. Reimplement the `Time` class from the preceding exercise so that the internal representation is the number of minutes since midnight (between 0 and $24 \times 60 - 1$). *Do not* change the public interface. That is, client code should be unaffected by your change.

5. Make a class `Student` with read-write JavaBeans properties `name` (of type `String`) and `id` (of type `Long`). What methods are generated? (Use `javap` to check.) Can you call the JavaBeans getters and setters in Scala? Should you?
6. In the `Person` class of Section 5.1, “Simple Classes and Parameterless Methods,” on page 55, provide a primary constructor that turns negative ages to 0.
7. Write a class `Person` with a primary constructor that accepts a string containing a first name, a space, and a last name, such as `new Person("Fred Smith")`. Supply read-only properties `firstName` and `lastName`. Should the primary constructor parameter be a `var`, a `val`, or a plain parameter? Why?
8. Make a class `Car` with read-only properties for manufacturer, model name, and model year, and a read-write property for the license plate. Supply four constructors. All require the manufacturer and model name. Optionally, model year and license plate can also be specified in the constructor. If not, the model year is set to -1 and the license plate to the empty string. Which constructor are you choosing as the primary constructor? Why?
9. Reimplement the class of the preceding exercise in Java, C#, or C++ (your choice). How much shorter is the Scala class?
10. Consider the class

```
class Employee(val name: String, var salary: Double) {  
  def this() { this("John Q. Public", 0.0) }  
}
```

Rewrite it to use explicit fields and a default primary constructor. Which form do you prefer? Why?

Index

Symbols and Numbers

- (minus sign)
 - in identifiers, 142
 - operator:
 - arithmetic, 6
 - for collections, 178–179
 - for maps, 49
 - for type parameters, 271
 - left-associative, 145
 - precedence of, 145
 - unary (negation), 14, 143
- operator, 177–179
 - not used for arithmetic decrements, 7
- = operator
 - arithmetic, 7
 - for collections, 178–179
 - for maps, 49
- = operator, 178–179
- _ (underscore)
 - as wildcard:
 - for XML elements, 236
 - in case clauses, 30, 198–199, 237
 - in imports, 8, 76, 85–86
 - in tuples, 52
 - for function calls, 158, 288
 - for function parameters, 161
 - in identifiers, 142, 317
- _* syntax
 - for arrays, 201
 - for nested structures, 207
 - in function parameters, 27
 - in pattern matching, 237
- _=, in setter methods, 57
- _1, _2, _3 methods (tuples), 51
- ;(semicolon)
 - after statements, 5, 18–20
 - inside loops, 24–25
- :(colon)
 - followed by annotations, 217
 - in case clauses, 200–201
 - in identifiers, 142
 - in implicits, 329–330
 - in operator names, 285
 - and precedence, 145
 - right-associative, 146, 185
 - in type parameters, 268–269
- :: operator
 - for lists, 175–176, 178–179
 - in case clauses, 201, 206
 - right-associative, 146, 176
- :::, += operators, 178–179
- :\, /: operators, 185
- := operator, 179
- ! (exclamation mark)
 - in identifiers, 142
 - operator:
 - in shell scripts, 114–115
 - precedence of, 145
 - unary, 143
- !! operator, in shell scripts, 114
- != operator, 144
- ? (question mark)
 - in identifiers, 142
 - in parsers, 308
- ?: operator, 18
- ??? method, 101
- / (slash)
 - in identifiers, 142
 - in XPath, 235
 - operator, 6
 - precedence of, 145
- //
 - for comments, 317
 - in XPath, 235
- /* ... */ comments, parsing, 317

- `/%` method (`BigInt`), 7, 203
- ``...`` (backquotes), for identifiers, 142, 200
- `^` (caret)
 - in identifiers, 142
 - in Pascal, 153
 - operator, 6
 - precedence of, 145
- `^?` operator, 313
- `^^` operator, 307–309, 312
- `^^^` operator, 312
- `~` (tilde)
 - in identifiers, 142
 - operator:
 - in case clauses, 206
 - in parsers, 305–311, 313–314
 - unary, 143
- `~!` operator, 313–314
- `>>` operator, 308–309, 312
- `'` (single quote)
 - in symbols, 226
 - parsing, 317
- `"` (double quote), parsing, 317
- `""`, in regular expressions, 116
- `()` (parentheses)
 - as value of `Unit`, 18–21
 - discarding, in parsers, 309
 - for annotations, 216
 - for `apply` method, 9–10
 - for array elements, 36
 - for curried parameters, 165
 - for functions, 158–161, 166
 - for maps, 48
 - for primary constructors, 63
 - for tuples, 51, 287
 - in case clauses, 201, 205
 - in method declarations, 8, 56, 59
 - in regular expressions, 117
 - to access XML attributes, 232
- `[]` (square brackets)
 - for arrays, 36
 - for methods in traits, 126
 - for type parameters, 14, 266, 287
- `{}` (braces)
 - for block expressions, 20–21
 - for existential types, 286
 - for function arguments, 159
 - for structural types, 283–285
 - in formatted strings, 21
 - in imports, 86
 - in package clauses, 83
 - in pattern matching, 211–212, 237
 - in REPL, 19
 - in XML literals, 234
 - Kernighan & Ritchie style for, 20
- `@` (at)
 - for XML attributes, 236
 - in case clauses, 207
 - in identifiers, 142
- `$` (dollar sign), in formatted strings, 21
- `*` (asterisk)
 - as wildcard in Java, 8, 85
 - in identifiers, 142
 - in parsers, 308
 - operator, 6, 327
 - no infix notation for, 286
 - precedence of, 145
- `**`
 - in Fortran, 153
 - in identifiers, 142
- `\` (backslash)
 - in identifiers, 142
 - operator, 235–236
- `\\` operator, 235–236
- `&` (ampersand)
 - in identifiers, 142
 - operator:
 - arithmetic, 6
 - for sets, 177–179
 - precedence of, 145
- `&...;` (XML), 231
- `&~` operator, 177–178
- `&#...;` (XML), 232
- `#` (number sign)
 - for type projections, 67, 281–283, 286
 - in identifiers, 142
- `#::` operator, 189
- `#&&, #<, #>, #>>, #||` operators (shell scripts), 115
- `#|` operator (shell scripts), 114
- `%` (percent sign)
 - for XML attributes, 238
 - in identifiers, 142
 - operator, 6
 - precedence of, 145
- `+` (plus sign)
 - in identifiers, 142
 - operator:
 - arithmetic, 6
 - for collections, 178–179
 - for maps, 49
 - for type parameters, 271
 - precedence of, 145
 - unary, 143

- +:: operator
 - for collections, 178–179
 - in case clauses, 207
 - right-associative, 146, 179
- ++ operator, 177–179
 - not used for arithmetic increments, 7
- ++, +=, -= operators, 178–179
- += operator
 - for array buffers, 36
 - for collections, 178–179
- = operator
 - arithmetic, 7
 - assignment, 144
 - for array buffers, 36, 42
 - for collections, 178–179, 335
 - for maps, 49
 - of Buffer, 335
- < (left angle bracket)
 - in identifiers, 142
 - in XML literals, 230
 - operator:
 - and implicits, 329
 - precedence of, 145
- <- operator, 23–24, 204
- <: operator, 267–271, 286, 293
- << operator, 269, 333
- <!--...-->, <?...?> comments (XML), 231
- <?xml...?> (XML), 241
- <- operator, 308–309, 312
- ⌘ operator, 268
- << operator, 6
- <= operator, 144
- = (equal sign)
 - in identifiers, 142
 - operator, 49, 144
 - precedence of, 145
- := operator, 269, 333
- == operator, 144
 - for collections, 173
 - for reference types, 103
- ===, != operators, 144
- => operator, 269
 - for functions, 166, 287–288
 - for self types, 134–135, 288–289, 295
 - in case clauses, 198
- > (right angle bracket)
 - in identifiers, 142
 - operator, 145
- > operator, 48
 - precedence of, 145
- >: operator, 267, 269
- >= operator, 144

- >> operator
 - arithmetic, 6
 - in parsers, 312
- | (vertical bar)
 - in case clauses, 198
 - in identifiers, 142
 - operator:
 - arithmetic, 6
 - for sets, 177–178
 - in parsers, 304–319
 - precedence of, 145
- √ (square root), 142
- 80 bit extended precision, 219

A

- abstract keyword, 97, 123, 127
- abstract types, 291–293, 315
 - bounds for, 293
 - made concrete in subclass, 283, 291
 - naming, 296
 - vs. type parameters, 292
- accept combinator, 313
- ActionEvent class (Java), 293–294
- actionPerformed method (listeners), 293
- addString method
 - of Iterable, 181
 - of Iterator, 189
- aggregate method
 - of Iterable, 180
 - of Iterator, 189
 - of parallel collections, 194
- aliases
 - for this, 68, 289
 - for types, 174, 283
- andThen method (Future), 256
- Annotation trait, 218
- annotations, 215–226, 287
 - arguments of, 217–218
 - for compiler optimizations, 222–226
 - for errors and warnings, 226
 - implementing, 218–219
 - in Java, 216–221
 - meta-annotations for, 219
 - order of, 216
- anonymous functions, 25, 159–160, 167
- Any class, 100, 103
 - asInstanceOf, isInstanceOf methods, 93, 100
- AnyRef class, 100, 112
 - == method, 103
 - eq, equals methods, 102
 - notify, notifyAll, synchronized, wait methods, 100

- AnyVal class, 100, 104
 - Apache Commons Resolver project, 240
 - App trait, 75
 - append, appendAll methods, 41
 - Application trait, 75
 - application objects, 74
 - apply method, 9–10, 73–74, 146–148
 - of Array, 10, 74
 - of BigInt, 10
 - of Buffer, CanBuildFrom, 335
 - of case classes, 205
 - of collections, 173
 - of Future, 253, 260
 - of PartialFunction, 211
 - of Process, 115
 - of StringOps, 9
 - applyDynamic method (Dynamic), 150–152
 - applyDynamicNamed method (Dynamic), 151–152
 - args property, 75
 - array buffers, 36–37
 - adding/removing elements of, 36–37
 - appending collections to, 36
 - converting to arrays, 37
 - displaying contents of, 40
 - empty, 36
 - largest/smallest elements in, 40
 - sorting, 40
 - transforming, 38–40
 - traversing, 37–38
 - Array class, 35–36, 269
 - apply method, 74
 - corresponds method, 165, 271
 - mkString, quickSort methods, 40
 - ofDim method, 42
 - toBuffer method, 37
 - Array companion object, 10, 202
 - ArrayBuffer class, 36–37, 172, 335
 - max, min, mkString methods, 40
 - mutable, 175
 - serializing, 113
 - sorted, sortWith methods, 40
 - subclasses of, 42
 - toArray method, 37
 - ArrayList class (Java), 36, 43, 173
 - arrays, 35–44
 - converting to array buffers, 37
 - displaying contents of, 40
 - fixed-length, 35–36
 - function call syntax for, 146
 - generic, 269
 - interoperating with Java, 43–44
 - invariant, 272
 - largest/smallest elements in, 40
 - multidimensional, 42–43, 74
 - pattern matching for, 201
 - ragged, 43
 - sorting, 40
 - transforming, 38–40
 - traversing, 37–38
 - vs. lists, 172
 - Arrays class (Java), 43
 - ArrayStoreException, 273
 - asAttrMap method, 233
 - ASCII characters, 142
 - asInstanceOf method (Any), 93, 100
 - asJavaXxx, asScalaXxx functions (JavaConversions), 192
 - assert method (Predef), 225
 - AssertionError, 225
 - assignments, 20–21, 144
 - no chaining of, 21
 - precedence of, 145
 - right-associative, 146, 179
 - value of, 20–21
 - Async library, 254
 - Atom class, 233–235
 - Attribute trait, 238
 - attributes (XML), 232–233
 - atoms in, 234
 - entity references in, 232, 234
 - expressions in, 234–235
 - in pattern matching, 238
 - iterating over, 233
 - modifying, 238
 - namespace of, 243
 - no wildcard notation for, 236
 - attributes property, 232
 - automatic conversions. *See* implicit conversions
 - Await object, 250
- ## B
- backtracking, 313–314
 - balanced trees, 50
 - bash shell, 114
 - bean properties, 61–62
 - @BeanDescription, @BeanDisplayName, @BeanInfo, @BeanInfoSkip annotations, 221
 - @beanGetter, @beanSetter annotations, 219
 - @BeanProperty annotation, 61–62, 216, 221
 - generated methods for, 65
 - BigDecimal class, 6–7
 - BigInt class, 6–9
 - /% method, 7, 203

- BigInt class (*continued*)
 - pow method, 153
 - unary_- method, 14
 - BigInt companion object
 - apply method, 10
 - probablePrime method, 9
 - binary files, reading, 112
 - binary functions, 162, 184
 - binary operators, 143–146
 - BitSet class, 177
 - blocking keyword, 260
 - blocking calls, 250
 - blocks, 20–21
 - BNF (Backus-Naur Form), 304
 - Boolean type, 5
 - reading, 22
 - @BooleanBeanProperty annotation, 221
 - Breaks object, break method, 23
 - Buffer class, 298, 335
 - bufferAsJavalist function (JavaConversions), 192
 - buffered method (Source), 110, 188
 - BufferedInputStream class (Java), 138
 - BufferedIterator trait, 188
 - Builder trait, 335
 - Byte type, 5
 - arrays of, 112
 - reading, 22
- C**
- C++ programming language
 - ?: operator in, 18
 - arrays in, 36
 - assignments in, 21
 - construction order in, 100
 - expressions in, 17
 - functions in, 25
 - implicit conversions in, 324
 - linked lists in, 176
 - loops in, 22, 38
 - methods in, 72, 94, 219
 - multiple inheritance in, 121–122
 - namespaces in, 80
 - operators in, 145
 - protected fields in, 94
 - reading files in, 110
 - singleton objects in, 72
 - statements in, 17–20
 - switch in, 198–199, 223
 - virtual base classes in, 122
 - void in, 18, 20, 102
 - cached thread pool, 260
 - cake pattern, 290
 - callbacks, 251–252
 - call-by-name parameters, 167
 - case clause, 198–212
 - _ in, 199, 237
 - : in, 200–201
 - :: in, 201, 206
 - `...` in, 200
 - ~ in, 206
 - () in, 201
 - {_*} in, 237
 - @, +: in, 207
 - =>, | in, 198
 - catch-all pattern for, 198–199
 - enclosed in braces, 211–212, 252
 - followed by variable, 199–200
 - infix notation in, 206–207
 - variables in, 199–200
 - XML literals in, 237
 - case classes, 205–211
 - applicability of, 208–209
 - declaring, 205
 - default methods of, 148, 205, 208–209
 - extending other case classes, 209
 - in parsers, 306, 309
 - modifying properties in, 205
 - sealed, 209
 - with variable fields, 208
 - case objects, 205
 - casts, 93–94
 - CatalogResolver class (Java), 240
 - catch statement, 30–31
 - as a partial function, 212
 - CDATA markup, 235, 240
 - ceil method (scala.math), 158
 - chaining
 - assignments, 21
 - auxiliary constructors, 65
 - method calls, 42
 - packages, 82–83
 - chain11 method (Parsers), 312
 - ChangeEvent class (Java), 293
 - Char type, 5, 22, 315
 - character references, 232
 - character sets, 111
 - characters
 - common, in two strings, 6
 - in identifiers, 142, 317
 - reading, 22, 110
 - sequences of, 14
 - uppercase, 14

- circular dependencies, 29, 135
- class keyword, 55, 287
- ClassCastException, 225
- classes, 10, 55–68, 287
 - abstract, 97, 125
 - abstract types in, 291
 - and primitive types, 5
 - annotated, 216
 - combined with primary constructor, 66
 - companion objects to, 9, 67, 72–73, 146, 281, 325
 - concrete, 130
 - definitions of, 56
 - executing statements in, 64
 - using traits in, 125
 - enriching, 324–325
 - equality checks in, 102
 - extending, 73, 91–92
 - Java classes, 95
 - multiple traits, 124
 - only one superclass, 121, 129
 - final, 92
 - generic, 266, 270
 - granting access to, 61–62
 - immutable, 7, 208
 - implementing, 265
 - importing members of, 76, 85
 - inheritance hierarchy of, 100–102
 - inlined, 103–105
 - interoperating with Java, 57
 - linearization of, 131
 - mutable, 208
 - names of, 142
 - nested, 66–68, 281
 - properties of, 57, 59
 - recompiling, 129
 - serializable, 113, 220
 - type aliases in, 283
 - visibility of, 56
 - vs. traits, 132
- classOf method (scala.Predef), 93
- ClassTag trait, 269, 299
- Cloneable interface (Java), 124, 220
- @cloneable annotation, 220
- close method (Source), 110
- closures, 162–163
- collect method
 - of Future, 256
 - of GenTraversable, 211
 - of Iterable, 180
 - of Iterator, 189
 - with partial functions, 183
- collectionAsScalaIterable function (JavaConversions), 192
- collections, 171–194
 - adding/removing elements of, 178–179
 - appending to array buffers, 36
 - applying functions to all elements of, 161, 180–183
 - combining, 187–188
 - companion objects of, 173, 335
 - constructing in a loop, 24, 38
 - converting to specific type, 181
 - filtering, 180
 - folding, 180, 184–186
 - hierarchy of, 41, 172–173
 - immutable, 173–174
 - instances of, 173
 - interoperating with Java, 191–192
 - iterators for, 188–189
 - methods for, 180–182
 - mutable, 173–174, 179, 191
 - ordered/unordered, 172, 179
 - parallel implementations of, 193–194
 - reducing, 180, 184
 - scanning, 180, 186
 - serializing, 113
 - sorted, 172
 - traits for, 172–173
 - traversing, 23, 38, 172, 222–223
 - unevaluated, 190
 - unordered, 179
- com.sun.org.apache.xml.internal.resolver.tools
 - package, 240
- combinations method (Seq), 182
- combinators, 311–313
- command-line arguments, 75
- comma-separated lists, 311
- comments
 - in lexical analysis, 304
 - in XML, 231
 - parsing, 240, 316–317
- companion objects, 9, 67, 72–73, 281, 328
 - apply method in, 146
 - implicit in, 325
 - of collections, 173, 335
- Comparable interface (Java), 42, 267–268, 329
- Comparator interface (Java), 226
- compareTo method, 267
- compiler
 - future flag, 268
 - implicit in, 327, 334
 - internal types in, 288
 - language:existentials flag, 287

- compiler (*continued*)
 - language:higherKinds flag, 299
 - language:implicitConversions flag, 324
 - language:postfixOps flag, 144
 - nested type expressions in, 283
 - optimizations in, 222–226
 - Scala annotations in, 216
 - Xcheckinit flag, 99
 - Xprint flag, 327
 - compiler plugin, 216
 - compile-time errors, 22
 - CompletableFuture class (Java), 260
 - CompletionStage interface (Java), 249
 - Component class (Java), 137
 - compound types, 284–287
 - comprehensions, 24
 - concurrency, 193
 - console
 - input from, 22, 111
 - printing to, 21
 - constants. *See* values
 - ConstructingParser class, 240–241
 - constructors
 - auxiliary, 62–63, 94
 - chaining, 65
 - eliminating, 64
 - executing, 72
 - order of, 98–100, 130–132
 - parameterless, 64, 132
 - parameters of, 62–66
 - annotated, 218–219
 - implicit, 269
 - primary, 62–66, 94
 - annotated, 216
 - default parameters in, 64
 - private, 66
 - superclass, 94–95
 - vals in, 99
 - Container trait, 298
 - Container class (Java), 137
 - contains method, 48
 - of BitSet, 177
 - of Iterator, 189
 - of Seq, 181
 - containsSlice method, 42
 - of Iterator, 189
 - of Seq, 181
 - of StringOps, 14
 - context bounds, 268–269, 329–330
 - control abstractions, 166–167
 - copy method, 238
 - of case classes, 205, 208–209
 - copyToArray method, 42
 - of Iterable, 181
 - of Iterator, 189
 - copyToBuffer method
 - of Iterable, 181
 - of Iterator, 189
 - corresponds method (Array), 165, 271
 - count method, 41
 - of Iterable, 180
 - of Iterator, 189
 - of StringOps, 14
 - Curry, Haskell Brooks, 164
 - currying, 164–165
- ## D
- debugging
 - implicit conversions, 326
 - reading from strings for, 111
 - reporting types for, 41
 - decrements, 7
 - def keyword, 25, 159, 165
 - abstract, 95
 - in parsers, 314
 - overriding, 95–96
 - parameterless, 95
 - return value of, 314
 - default statement, 198
 - definitions, 24–25
 - DelayedInit trait, 75
 - Delimiters type, 328
 - dependency injections, 289–291
 - @deprecated annotation, 219, 226
 - @deprecatedInheritance, @deprecatedOverriding
 - annotations, 226
 - @deprecatedName annotation, 218, 226
 - destructuring
 - of lists, 207
 - of tuples, 202
 - diamond inheritance problem, 122–123
 - dictionaryAsScalaMap function (JavaConversions), 192
 - diff method
 - of Iterator, 189
 - of Seq, 182
 - of sets, 177
 - directories
 - and packages, 80
 - naming, 15
 - traversing, 112–113
 - division, quotient and remainder of, 7, 203
 - do loop, 22
 - docElem method (ConstructingParser), 240

DocType class, 241
domain-specific languages, 141, 303
Double type, 5
 reading, 22
DoubleLinkedList class (deprecated), 176
drop, dropWhile methods
 of Iterable, 180
 of Iterator, 189
dropRight method (Iterable), 180
DTDs (Document Type Definitions), 239–241
duck typing, 284
Duration object, 250
Dynamic trait, 150
dynamic invocation, 150–153
dynamically typed languages, 284

E

early definitions, 99, 132–133
EBNF (Extended Backus-Naur Form), 305–306
Eclipse-based Scala IDE, 2
Eiffel programming language, 58
Either type, 300
Elem type, 230, 315
 prefix, scope values, 238, 243
elements (XML), 230
 attributes of. *See* attributes (XML)
 child, 237–238
 empty, 242
 matching, 236
 modifying, 238
endsWith method
 of Iterator, 189
 of Seq, 181
entity references, 231
 in attributes, 232, 234
 resolving, 241
EntityRef class, 232
Enumeration class, 75–77
enumerationAsScalaIterator function (JavaConversions), 192
enumerations, 75–77
 simulating, 209
 values of, 76–77
eq method (AnyRef), 102
equals method
 of AnyRef, 102–103
 of case classes, 205, 208–209
 of value classes, 104
 overriding, 103
 parameter type of, 103
Equiv type, 332

err combinator, 313
Error class, 315
error messages
 explicit, 319
 for implicit conversions, 326, 334
 generated with annotations, 226
 suppressed, 226
 type projections in, 283
 with override modifier, 92
escape hatch, 142
evidence objects, 333
Exception trait, 288
exceptions, 29–31
 catching, 30
 checked, in Java, 29, 220
execution contexts, 260
ExecutionContext trait, 248
Executor interface (Java), 248
Executors, 260
existential types, 287
exists method
 of Iterable, 180
 of Iterator, 189
expressions
 annotated, 217
 conditional, 18–19
 traversing values of, 23
 type of, 18
 vs. statements, 17
extends keyword, 91, 99, 123–124
extractors, 118, 147–149, 202–203

F

f prefix, in formatted strings, 21–22, 112
failed method
 of Future object, 258
 of Future trait, 255–256
failure method (Promise), 259
Failure class, 251, 315
failure combinator, 313
fallbackTo method (Future), 255
fall-through problem, 198
family polymorphism, 293–296
@field annotation, 219
fields
 abstract, 97–98, 130, 132
 accessing uninitialized, 99
 annotated, 216
 comparing, 208
 concrete, 98, 128–129
 copying, 208

- fields (*continued*)
 - for primary constructor parameters, 62, 65
 - getter/setter methods for, 57, 61–62, 65
 - hash codes of, 103, 208
 - immutable, 65
 - object-private, 60–61, 65
 - overriding, 91, 95–96, 130, 132
 - printing, 208
 - private, private final, 59
 - protected, 94
 - public, 56
 - static, 71
 - transient, volatile, 219
- FileInputStream class (Java), 112
- files
 - and packages, 80
 - appending, 115
 - binary, 112
 - naming, 15
 - processing, 109–115
 - reading, 109–111
 - redirecting input/output from/to, 115
 - saving, 241–242
 - writing, 112
- Files class (Java), 112–113
- filter method, 39, 162, 210
 - of Future, 256
 - of Iterable, 180
 - of Iterator, 189
- final keyword, 59, 92
- finally statement, 30–31
- find, firstCompletedOf methods (Future object), 257–258
- findAllIn, findFirstIn methods, 116
- flatMap method, 182–183
 - of Future, 254
 - of Iterable, 180
 - of Iterator, 189
 - of Try, 255
- Float type, 5
 - reading, 22
- floating-point calculations, 219
- fluent interfaces, 280–281
- fold method
 - of Iterable, 180
 - of Iterator, 189
 - of parallel collections, 194
- foldLeft method, 168, 184–185
 - of Future object, 257
 - of Iterable, 180, 273
 - of Iterator, 189
 - of parallel collections, 194
- foldRight method, 185
 - of Iterable, 180
 - of Iterator, 189
 - of parallel collections, 194
- for loop, 22–25
 - constructing collections in, 24, 38
 - enhanced (Java), 38
 - for arrays, 37–40
 - for futures, 254
 - for maps, 50
 - for regex groups, 118
 - guards in, 204, 254
 - pattern matching in, 204
 - range-based (C++), 38
 - regular expressions in, 116
 - with Option type, 210
- forall method
 - of Iterable, 180
 - of Iterator, 189
- force method, 190
- foreach method, 162, 183, 210
 - of Future, 255–256
 - of Iterable, 180
 - of Iterator, 189
- fork-join pool, 260
- formatted strings, 21–22, 112
- Fortran programming language, 153
- Fraction class, 146–147, 324–327
- Fraction companion object, 325
- Fractional type, 332
- fragile base class problem, 92
- French delimiters, 328
- fromString, fromURL methods (Source), 111
- fromTry method (Future), 258
- functional programming languages, 157
- functions, 25–26, 157–167, 287
 - anonymous, 25, 159–160, 167
 - as method parameters, 14, 159
 - binary, 162, 184
 - calling, 8–9, 158
 - converting to Java interfaces, 164
 - curried, 164–167, 328
 - exiting immediately, 25
 - for all elements of a collection, 161
 - from methods, 288
 - generic, 266, 269
 - higher-order, 160–162
 - identity, 333
 - implementing, 265
 - left-recursive, 310
 - mapping, 182–183
 - names of, 14, 142, 324

- nested, 23
- parameterless, 166–167
- parameters of, 25
 - call-by-name, 167
 - default, 26
 - named, 26
 - other functions as, 160
 - type deduction in, 160
 - variable, 26–27
- partial, 183, 211–212, 313
- passing to another function, 159–160, 163
- recursive, 25–27
- return type of, 5, 25, 28
- return value of, 166–167
- scope of, 163
- single-argument, 161, 272
- storing in variables, 157–159
- syntax of, 146–147
- variance of, 272
- vs. variables, in parsers, 313
- future method (Promise), 259
- Future companion object, 256–258
 - apply method, 253, 260
 - find, firstCompletedOf methods, 257–258
 - foldLeft, reduceLeft, traverse methods, 257
 - fromTry, never, successful, unit methods, 258
- Future interface (Java), 249
- Future trait, 248–260
 - andThen, collect methods, 256
 - failed, fallbackTo method, 255–256
 - filter method, 256
 - flatMap method, 254
 - foreach method, 255–256
 - isCompleted method, 250
 - map method, 253–254
 - onComplete, onSuccess, onFailure methods, 252
 - ready, result methods, 250
 - recover, recoverWith methods, 255–256
 - transform, transformWith methods, 256
 - value method, 250
 - zip, zipWith methods, 255
- futures
 - blocking waits for, 251
 - chaining, 253
 - delaying creation of, 254
 - execution context of, 260
 - tasks of:
 - composing, 252–255
 - failing, 250
 - running, 248–249
 - transformations of, 255–256
 - vs. promises, 259

G

- generators, 24–25
- generic arrays, 269
- generic classes, 266
 - conditionally used methods in, 270
- generic functions, 266, 269
- generic methods, 266
- generic types
 - erased, in JVM, 269
 - variance of, 271–275
- GenTraversable trait, 211
- GenXxx sequences, 14
- get method (Try), 251
- get, getOrElse methods (Map), 48, 210, 232
- getLines method (Source), 109, 190
- @getter annotation, 219
- getXxx methods, 57, 61, 221
- grammars, 304–305
 - left-recursive, 314
- Group type, 235
- groupBy method, 183
- grouped method
 - of Iterable, 181, 188
 - of Iterator, 189
- guard combinator, 313
- guards, 24–25, 38, 199
 - for pattern matching, 238
 - in for statements, 204, 254
 - variables in, 199

H

- hash codes, 100, 103
- hash sets, 177
- hash tables, 47, 50
- hashCode method, 177
 - of AnyRef, 103
 - of case classes, 205, 208–209
 - of value classes, 104
 - overriding, 103
- Hashing type, 332
- Haskell programming language, 26, 331
- hasNext method
 - of Iterable, 188
 - of Iterator, 127
- head method, 110
 - of Iterable, 180, 188
 - of lists, 175–176
- headOption method (Iterable), 180
- Hindley-Milner algorithm, 26
- HTTP (Hypertext Transfer Protocol), 111, 303

- I
- id method, 76
- @Id annotation, 219
- ident method, 317
- Identifier type, 317
- identifiers, 142, 317
- identity functions, 333
- IEEE double values, 219
- if expression, in loop generators, 24, 38
- if/else expression, 18–19, 30
- IllegalStateException, 259
- immutability, 7
- implements keyword, 123
- implicit conversions, 42–43, 141, 323–336
 - ambiguous or multiple, 327
 - debugging, 326
 - for parsers, 316
 - for strings to ProcessBuilder objects, 114
 - for type parameters, 268
 - importing, 325–326, 330
 - naming, 324
 - rules for, 326–327
 - unwanted, 191, 324, 326
 - uses of, 324–325
- implicit evidence parameter, 270
- implicit keyword, 324, 328–329
- implicit parameters, 9, 14, 268, 299, 328–336
 - not available, 226, 334
 - of common types, 328
- implicit values, 268–269
- @implicitAmbiguous annotation, 226
- implicitly method (Predef), 330, 333
- @implicitNotFound annotation, 226, 334
- :implicits in REPL, 326
- import statement, 76, 80, 85–87
 - implicit, 86–87
 - location of, 85
 - overriding, 86
 - selectors for, 86
 - wildcards in, 8, 85–86
- increments, 7
- IndexedSeq companion object, 336
- IndexedSeq trait, 172, 174–175, 336
- indexXXX methods
 - of Iterator, 189
 - of Seq, 181
- infix notation
 - for operators, 143–145
 - precedence of, 285
 - for types, 285–287
 - in case clauses, 206–207
 - in math, 285
 - with anonymous functions, 159
- inheritance, 91–105
 - diamond, 122–123
 - hierarchy of, 100–102
 - multiple, 121–123
- init method (Iterable), 180
- @inline annotation, 224
- inlining, 224
- InputStream class (Java), 239
- Int type, 5, 268, 270
 - immutability of, 7
 - no null value in, 100
 - reading, 22
- int2Fraction method, 325
- Integer class (Java), 225
- interfaces
 - fluent, 280–281
 - rich, 127
- interpolated strings. *See* formatted strings
- InterruptedException, 251
- intersect method
 - of Iterator, 189
 - of Seq, 182
 - of sets, 177
 - of StringOps, 6
- intersection types. *See* compound types
- intertwining classes and constructors, 94–95
- into combinator, 311–312
- isCompleted method
 - of Future, 250
 - of Promise, 259
- isDefinedAt method (PartialFunction), 211
- isEmpty method
 - of Iterable, 180
 - of Iterator, 189
- isInstanceOf method (Any), 93, 100, 200
- isSuccess, isFailure methods (Try), 251
- IsTraversableXXX type classes, 332
- istream::peek function (C++), 110
- itemStateChanged method (listeners), 293
- Iterable trait, 41, 172, 296–299
 - methods of, 180–182, 188, 273
- iterableAsScalaIterable function (JavaConversions), 192
- iterator method (collections), 188
- Iterator trait, 127, 172
 - methods of, 189
- iterators, 110, 188–189
 - constructing streams from, 190
 - fragility of, 189
 - turning into arrays, 116

J

- Java programming language
 - ?: operator in, 18
 - annotations in, 216–221
 - arrays in, 36, 43, 173, 273
 - assertions in, 225
 - assignments in, 21
 - checked exceptions in, 29, 220
 - classes in, 91–92
 - hierarchy of, 67
 - serializable, 113
 - closures in, 163
 - collections in, 172
 - completable futures in, 260
 - construction order in, 100
 - dependencies in, 290
 - event handling in, 293
 - expressions in, 17
 - fields in:
 - protected, 94
 - public, 56
 - functional (SAM) interfaces in, 164
 - futures in, 249
 - generic types in, 275
 - hashCode method for each object in, 177
 - identifiers in, 142
 - interfaces in, 121–124, 135, 137
 - interoperating with Scala:
 - arrays, 43–44
 - classes, 57, 95, 216
 - collections, 191–192
 - maps, 50–51, 204
 - methods, 221
 - traits, 135, 137
 - lambda expressions in, 164
 - linked lists in, 173, 176
 - loops in, 22, 38
 - maps in, 173
 - methods in, 72, 92, 94
 - abstract, 97
 - overriding, 99
 - static, 9, 25
 - with variable arguments, 27
 - missing values in, 270
 - modifiers in, 219
 - no multiple inheritance in, 121
 - no operator overloading in, 8
 - no variance in, 226
 - null value in, 100
 - operators in, 145
 - packages in, 80, 82, 84
 - primitive types in, 36, 100
 - reading files in, 110–112
 - singleton objects in, 72
 - statements in, 17–20
 - superclass constructors in, 95
 - switch in, 223
 - synchronized in, 100
 - toString method in, 41
 - traversing directories in, 112–113
 - type checks and casts in, 93
 - void in, 18, 20, 102
 - wildcards in, 85, 275, 286
- Java AWT library, 137
- Java EE (Java Platform, Enterprise Edition), 216
- java.awt.Component, java.awt.Container classes, 137
- java.io.BufferedInputStream class, 138
- java.io.FileInputStream class, 112
- java.io.InputStream class, 239
- java.io.PrintWriter class, 112
- java.io.Reader class, 239
- java.io.Serializable interface, 124
- java.io.Writer class, 241
- java.lang package, 86–87
- java.lang.Cloneable interface, 124, 220
- java.lang.Comparable interface, 42
- java.lang.Integer class, 225
- java.lang.Object class, 100
- java.lang.ProcessBuilder class, 44
- java.lang.String class, 6, 36
- java.lang.Throwable class, 29
- java.math.BigDecimal classes, 6
- java.nio.file.Files class, 112–113
- java.rmi.Remote interface, 220
- java.util package, 192
- java.util.ArrayList class, 36, 43, 173
- java.util.Arrays class, 43
- java.util.Comparator class, 226
- java.util.concurrent.CompletableFuture class, 260
- java.util.concurrent.CompletionStage interface, 249
- java.util.concurrent.Executor interface, 248
- java.util.concurrent.Executors class, 260
- java.util.concurrent.Future interface, 249
- java.util.LinkedList class, 173
- java.util.List interface, 43, 173
- java.util.Properties class, 51, 204
- java.util.RandomAccess interface, 173
- java.util.Scanner class, 53, 111
- JavaBeans, 61–62, 221
 - property change listeners in, 137

JavaConversions class, 43, 51, 191–192
 Javadoc, 10
 javap command, 58
 JavaScript, 235

- closures in, 163
- duck typing in, 284
- translating from Scala, 10

 JavaTokenParsers trait, 316
 JButton, JComponent classes (Swing), 137
 JDK (Java Development Kit), source code for, 212
 JSON (JavaScript Object Notation), 303
 jump tables, 223
 JUnit framework, 216–217
 JVM (Java Virtual Machine)

- arrays in, 36
- classes in, 129
- generic types in, 269
- inlining in, 224
- stack size of, 222
- transient/volatile fields in, 219

K

Kernighan & Ritchie brace style, 20
 keySet method (Map), 50
 Keyword type, 317

L

last, lastOption methods (Iterable), 180
 lastIndexOf, lastIndexOfSlice methods

- of Iterator, 189
- of Seq, 181

 lazy keyword, 28–29, 133
 length method

- of Iterable, 180
- of Iterator, 189

 lexical analysis, 304
 li element (XML), 233–234
 lines

- iterating over, 110
- reading, 109

 linked hash sets, 177
 LinkedHashMap class, 50
 LinkedList class (deprecated), 176
 LinkedList class (Java), 173
 List class, 296, 306

- immutable, 174
- implemented with case classes, 208

 List interface (Java), 43, 173
 list method (java.nio.file.Files), 112

ListBuffer class, 175
 lists, 175–176

- adding/removing elements of, 178–179
- constructing, 146, 175
- destructuring, 176, 207
- empty, 101
- heterogeneous, 213
- immutable, 189, 274
- linked, 172
- pattern matching for, 201–202, 206
- traversing, 176
- vs. arrays, 172

 literals. *See* XML literals
 loadFile method (XML), 239
 log combinator, 311, 313
 log messages

- adding timestamp to, 125
- printing, 313
- truncating, 125, 130
- types of, 127

 LoggedException trait, 135, 288–289
 Logger trait, 128
 Long type, 5

- reading, 22

 loops, 22–25

- breaking out of, 23
- for collections, 23
- variables within, 24
- vs. folding, 186

M

main method, 74
 makeURL function, 234
 ManagedException trait, 289
 map method, 39, 161, 210

- of Future, 253–254
- of Iterable, 180–184, 297–298
- of Iterator, 189
- of Try, 255

 Map trait, 48, 172

- get, getOrElse methods, 48, 210, 232
- immutable, 174
- keySet, values methods, 50

 mapAsXxxMap functions (JavaConversions), 51, 192
 maps, 47–52

- blank, 48
- constructing, 48
 - from collection of pairs, 52
- function call syntax for, 146
- interoperating with Java, 50–51

- iterating over, 50
 - keys of:
 - checking, 48
 - removing, 49
 - visiting in insertion order, 50
 - mutable/immutable, 48–49, 173
 - reversing, 50
 - sorted, 50
 - traversing, 204
 - updating values of, 49
 - values of, 48
 - match expression, 198–210, 224, 271
 - annotated, 223
 - processing Try instances with, 251
 - Match class, 117
 - MatchError, 198
 - Math class, 8
 - mathematical functions, 8, 14
 - max method, 42
 - of ArrayBuffer, 40
 - of Iterable, 180
 - of Iterator, 189
 - maximum munch rule, 317
 - MessageFormat.format method (Java), 27
 - MetaData type, 232–233, 238
 - method types (in compiler), 288
 - methods
 - abstract, 95–98, 123, 127, 135
 - abundance of, 10, 14
 - accessor, 56
 - annotated, 216
 - as arithmetic operators, 6
 - calling, 3, 5, 8–9, 56–57, 59, 126
 - chained, 280
 - clashes of, in sub- and superclasses, 92
 - co-/contravariant, 333
 - curried, 165, 271
 - declaring, 56
 - executed lazily, 189
 - final, 92, 103, 223
 - for primary constructor parameters, 65
 - generic, 266
 - getter/setter, 56–60, 98, 216, 221
 - inlining, 224
 - modifiers for, 84
 - mutator, 56
 - names of, 7
 - misspelled, 92
 - overriding, 91–93, 95–96, 127
 - parallelized, 193
 - parameterless, 8, 56, 95
 - parameters of, 266, 273, 280
 - implicit, 14, 268
 - using functions for, 14, 159
 - wrong type, 92
 - private, 59, 223
 - protected, 94
 - public, 57
 - return type of, 273, 280
 - return value of, 266
 - static, 71
 - turning into functions, 158, 288
 - used under certain conditions, 270
 - variable-argument, 27, 221
 - with two parameters, 7
 - Meyer, Bertrand, 58
 - min method
 - of ArrayBuffer, 40
 - of Iterable, 180
 - of Iterator, 189
 - of scala.math, 8
 - mkString method
 - of Array, ArrayBuffer, 40
 - of Iterable, 181
 - of Iterator, 189
 - of Source, 110–111
 - ML programming language, 26
 - mulBy function, 160, 163
 - multiple inheritance, 121–123
 - MutableList class (deprecated), 176
 - mutableXXXAsJavaXXX functions (JavaConversions), 192
- ## N
- NamespaceBinding class, 242
 - namespaces, 242–243
 - @native annotation, 219
 - negation operator, 14
 - never method (Future), 258
 - new keyword, 66–67
 - and nested expressions, 74
 - constructing objects without, 146
 - omitting, 205, 207–208
 - newline character
 - in long statements, 20
 - in printed values, 21
 - inside loops, 25
 - next method
 - of Iterable, 188
 - of Iterator, 127
 - Nil list, 101, 175–176, 226

- node sequences, 230
 - binding variables to, 237
 - building programmatically, 231
 - descendants of, 236
 - grouping, 235
 - immutability of, 232, 238
 - in embedded blocks, 234
 - traversing, 231
 - turning into strings, 232
- Node type, 230–232
- NodeBuffer class, 231–232
- NodeSeq type, 230–232
 - XPath-like expressions in, 235–236
- @noinline annotation, 224
- None object, 210, 306–307
- nonterminal symbols, 305
- NoSuchElementException, 254
- not combinator, 313
- Nothing type, 30, 100, 271, 274
- notify, notifyAll methods (AnyRef), 100
- NotImplementedError, 101
- @NotNull annotation, 218
- Null type, 100, 238
- null value, 100, 270
- NumberFormatException, 111, 251
- numbers
 - average value of, 332
 - classes for, 14
 - converting:
 - between numeric types, 6, 10
 - to arrays, 111
 - greatest common divisor of, 153
 - in identifiers, 317
 - invoking methods on, 5
 - parsing, 312, 317
 - random, 9
 - ranges of, 14
 - reading, 22, 111
 - sums of, 40
- Numeric type, 331
- numericLit method, NumericLit type, 317
- O**
- Object class, 100
- Object class (Java), 100
- object keyword, 71–77, 281
- objects, 71–77
 - adding traits to, 125
 - cloneable, 220
 - companion, 9, 67, 72–73, 146, 173, 281, 325, 328, 335
 - compound, 208
 - constructing, 10, 56, 72, 125
 - default methods for, 177
 - equality of, 102–103
 - extending class or trait, 73
 - extracting values from, 202
 - importing members of, 76, 85
 - nested, 207
 - nested classes in, 66–68, 281
 - no type parameters for, 274
 - of a given class, testing for, 93–94
 - pattern matching for, 200
 - remote, 220
 - scope of, 282
 - serializing, 113, 284
 - type aliases in, 283
 - variance of, 272
- ofDim method (Array), 42
- onComplete, onSuccess, onFailure methods (Future), 252
- operators, 141–149
 - arithmetic, 6–8
 - assignment, 144–146
 - associativity of, 145, 194
 - binary, 143–146
 - for adding/removing elements, 178–179
 - infix, 143, 145
 - overloading, 8
 - parsing, 317
 - postfix, 143–145
 - precedence of, 144–145, 285, 307
 - unary, 143–144
- opt method (Parsers), 305–306
- optimization, 222–226
- Option class, 48, 116, 147, 149, 180, 210, 233, 235, 306–307
 - orNull method, 270
 - using with for, 210
- Ordered trait, 40, 42, 329–330
- Ordering type, 42, 329–332
- OSGi (Open Services Gateway initiative framework), 290
- OutOfMemoryError, 190
- override keyword, 92–93, 95–96, 123, 127
 - omitted, 97–98
- @Overrides annotation, 92
- P**
- package objects, 8, 83–84
- packages, 80–87
 - adding items to, 80

- chained, 82–83
 - defined in multiple files, 80
 - importing, 8, 85–87
 - always, 86–87
 - selected members of, 86
 - modifiers for, 84
 - naming, 82, 87
 - nested, 81–82
 - scope of, 282
 - top-of-file notation for, 83
- packrat parsers, 314–315
- PackratParsers trait, 314
- PackratReader class, 315
- padTo method
 - of Iterator, 189
 - of Seq, 182
- Pair class, 275
- par method (parallel collections), 193
- parallel implementations, 193
- @param annotation, 219
- parameters
 - annotated, 216
 - curried, 271
 - deprecated, 226
 - implicit, 9, 14, 299
 - named, 205
- ParMap trait, 193
- parse method, 306
- parse trees, 308–309
- parseAll method, 306, 313, 315, 318–319
- ParSeq trait, 193
- parsers, 303–319
 - backtracking in, 313–314
 - combining operations in, 305–307
 - entity map of, 241
 - error handling in, 319
 - implicit conversions for, 316
 - numbers in, 312
 - output of, 307–308
 - packrat, 314–315
 - regex, 316, 319
 - skipping comments in, 316
 - strings in, 312
 - whitespace in, 316
- Parsers trait, 305, 315–319
- ParSet trait, 193
- partial functions, 183, 211–212
- PartialFunction class, 211–212
- partition method
 - of Iterable, 180
 - of Iterator, 189
 - of StringOps, 52
- Pascal programming language, 153
- paths, 282–283
- pattern matching, 197–211
 - and += operator, 179
 - by type, 200–201
 - classes for. *See* case classes
 - extractors in, 147
 - failed, 147
 - for arrays, 201
 - for lists, 176, 201–202, 206
 - for maps, 50
 - for objects, 200
 - for tuples, 52, 201–202
 - guards in, 199, 238
 - in for expressions, 204
 - in variable declarations, 203–204
 - in XML, 237–238
 - jump tables for, 223
 - not exhaustive, 226
 - of nested structures, 207
 - regular expressions in, 203
 - variables in, 199–200
 - vs. type checks and casts, 93–94
 - with extractors, 202
 - with Option type, 210
 - with partial functions, 211–212
- PCData type, 235
- permutations method
 - of Iterator, 189
 - of Seq, 182
- phrase combinator, 313
- piping, 114
- Play web framework, 249
- Point class, 332
- polymorphism, 208
- Positional trait, 313, 319
- positioned combinator, 313, 319
- postfix operators, 143–145
 - in parsers, 308
- pow method (scala.math), 8, 153
- Predef object
 - always imported, 86–87, 93, 174
 - assert method, 225
 - asInstanceOf method, 93
 - implicitly method, 330, 333
 - implicit in, 329–333
- prefixedKey method, 243
- prefixLength method
 - of Iterator, 189
 - of Seq, 181
- PrettyPrinter class, 242
- primitive types, 5, 36, 225

- print function, 21, 111
 - printf function, 21, 112
 - println function, 21
 - PrintStream.printf method (Java), 27
 - PrintWriter class (Java), 112
 - PriorityQueue class, 175
 - private keyword, 57–68, 84
 - probablePrime method (BigInt), 9
 - procedures, 28
 - process control, 114–115
 - Process object, 115
 - ProcessBuilder class (Java), 44
 - constructing, 115
 - implicit conversions to, 114
 - processing instructions, 231
 - product method
 - of Iterable, 180
 - of Iterator, 189
 - programs
 - concurrent, 193
 - displaying elapsed time for, 75
 - implicit imports in, 86–87
 - piping, 114
 - readability of, 8
 - self-documenting, 296
 - Promise trait, 258–260
 - promises, 258–260
 - properties, 57
 - in Java. *See* bean properties
 - read-only, 59
 - write-only (not possible), 60
 - Properties class (Java), 51, 204
 - propertiesAsScalaMap function (JavaConversions), 51, 192
 - propertiesAsScalaMap method (JavaConversions), 204
 - property change listeners, 137
 - protected keyword, 84, 94
 - public keyword, 56, 84
 - PushbackInputStreamReader class (Java), 110
 - Python programming languages, closures in, 163
- Q**
- Queue class, 174–175
 - quickSort method (Array), 40
- R**
- r method (String), 116–117
 - Random object, 9
 - RandomAccess interface (Java), 173
 - Range class, 5, 14, 297–298, 336
 - immutable, 174–175
 - traversing, 23
 - raw prefix, in formatted strings, 22
 - raw string syntax, 116
 - readBoolean, readByte, readChar, readFloat, readLine, readShort methods (scala.io.StdIn), 22
 - readDouble, readInt, readLong methods (scala.io.StdIn), 22, 111
 - Reader class (Java), 239, 315
 - ready, result methods (Await), 250
 - recover, recoverWith methods (Future), 255–256
 - recursions, 174
 - for lists, 176
 - left, 310–311
 - tail, 223
 - reduce method
 - of Iterable, 180
 - of Iterator, 189
 - of parallel collections, 193
 - reduceLeft method, 162, 184
 - of Future object, 257
 - of Iterable, 180
 - of Iterator, 189
 - of parallel collections, 193
 - reduceRight method, 184
 - of Iterable, 180
 - of Iterator, 189
 - of parallel collections, 193
 - reference types
 - = operator for, 103
 - assigning null to, 100
 - reflective calls, 284
 - Regex class, 116–117
 - replaceXXXIn methods, 116, 211
 - RegexParsers trait, 305, 315–316, 319
 - regular expressions, 116–117
 - for extractors, 203
 - grouping, 117–118
 - in parsers, 316
 - matching tokens against, 305
 - raw string syntax in, 116
 - return value of, 306
 - Remote interface (Java), 220
 - @remote annotation, 220
 - rep method (Parsers), 305–306, 311–312
 - REPL (read-eval-print loop), 3–4
 - implicits in, 326, 333
 - nearsightedness of, 19
 - paste mode in, 19, 73
 - types in, 158, 283
 - replaceXXXIn methods (Regex), 116, 211

- repXxx methods (Parsers), 312
- result method, 223
- return keyword, 25, 167
- reverse method
 - of Iterator, 189
 - of Seq, 182
- RewriteRule class, 239
- rich interfaces, 127
- RichChar class, 6
- RichDouble class, 6, 14
- RichFile class, 325
- RichInt class, 6, 14, 22, 268
- _root_ prefix, in package names, 82
- Ruby programming language
 - closures in, 163
 - duck typing in, 284
- RuleTransformer class, 239

S

- s prefix, in formatted strings, 22
- SAM (single abstract method) conversions, 163
- save method (XML), 241
- SAX parser, 239
- scala package, always imported, 82, 86–87, 113, 174
- Scala programming language
 - embedded languages in, 141, 303
 - interoperating with shell programs, 114
 - interpreter of, 1–4
 - older versions of, 75
 - strongly typed, 150
 - translating to JavaScript, 10
- scala/bin directory, 1
- scala. prefix, in package names, 8, 87
- scala.collection package, 173, 192
- Scala.js project, 10
- scala.language.existentials, importing, 287
- scala.language.higherKinds, importing, 299
- scala.language.implicitConversions, importing, 324
- scala.math package, 8, 14
 - ceil method, 158
- scala.sys.process package, 114
- scala.util package, 9
- scala-ARM library, 31
- scalac program. *See* compiler
- Scaladoc, 6, 10–15, 41–42, 221
- ScalaObject interface, 100
- scanLeft, scanRight methods, 186
- Scanner class (Java), 53, 111
- sealed keyword, 209
- segmentLength method
 - of Iterator, 189
 - of Seq, 181
- selectDynamic method (Dynamic), 151–152
- selectors, for imports, 86
- self types, 68, 134–135, 288–295
 - dependency injections in, 290–291
 - no automatic inheritance for, 289
 - structural types in, 135
 - typesafe, 294
 - vs. traits with supertypes, 135
- Seq trait, 27, 41, 172–175, 235
 - methods of, 181–182
- seq method (parallel collections), 193
- Seq[Char] class, 14
- Seq[Node] class, 231–232
- seqAsJavaList function (JavaConversions), 192
- sequences, 14
 - adding/removing elements of, 179
 - as function parameters, 27
 - comparing, 165, 271
 - extracting values from, 149–150
 - filtering, 162
 - integer, 175
 - mutable/immutable, 174–175
 - of characters, 14
 - reversing, 182
 - sorting, 162, 182
 - with fast random access, 175
- Serializable interface (Java), 124
- Serializable trait, 113
- serialization, 113
- @SerialVersionUID annotation, 113, 220
- Set trait, 172
 - immutable, 174
- setAsJavaSet function (JavaConversions), 192
- sets, 177
 - difference, intersection, union operations on, 177–178
 - elements of, 177–179
- @setter annotation, 219
- setXxx methods, 57, 61, 221
- shadowing rule, 24
- shell scripts, 114–115
- Short type, 5
 - reading, 22
- singleton objects, 8–9, 71–72, 281
 - case objects for, 205
- singleton types, 280–281, 283, 287
- slice, span, splitAt methods
 - of Iterable, 180
 - of Iterator, 189

- sliding method
 - of Iterable, 181, 188
 - of Iterator, 189
- SmallTalk programming language, 60
- Some class, 210, 306–307
- sortBy method
 - of Iterator, 189
 - of Seq, 182
- sorted method
 - of ArrayBuffer, 40
 - of Iterator, 189
 - of Seq, 182
 - of StringOps, 8, 14
- sorted sets, 177
- SortedMap trait, 50, 172
- SortedSet trait, 172
- sortWith method, 162
 - of ArrayBuffer, 40
 - of Iterator, 189
 - of Seq, 182
- Source object, 109–111
 - buffered, close, toArray, toBuffer methods, 110
 - fromString, fromURL methods, 111
 - getLines method, 109, 190
 - mkString method, 110–111
- @specialized annotation, 225–226
- Spring framework, 290
- sqrt method (scala.math), 8
- src.zip file (JDK), 212
- Stack class, 174–175
- stack overflow, 222
- standard input, 111
- StandardTokenParsers class, 317
- start symbol, 305–306
- startsWith method
 - of Iterator, 189
 - of Seq, 181
- stateChanged method (listeners), 293
- statements
 - line breaks in, 20
 - terminating, 19–20
 - vs. expressions, 17
- stdin method, 111
- StdIn object, 22, 111
- StdLexical trait, 318
- StdTokenParsers trait, 315, 318
- StdTokens trait, 317
- Stream class, 174
- streams, 189–190
- @strictfp annotation, 219
- String class, 116–117
- String class (Java), 6, 36
- string interpolation, 21
- stringLit method, StringLit type, 317
- StringOps class, 6, 14, 52
 - apply method, 9
 - containsSlice method, 14
 - sorted method, 8, 14
- strings, 6
 - characters in:
 - common, 6
 - uppercase, 14
 - classes for, 14
 - converting:
 - from any objects, 6
 - to numbers, 10, 111
 - to ProcessBuilder objects, 114
 - formatted, 21–22, 112
 - parsing, 312, 317
 - sorting, 8
 - testing for sequences in, 14
 - traversing, 23
 - vs. symbols, 226
- structural types, 97, 135, 283–284
 - adding to compound types, 285
- subclasses
 - anonymous, 97
 - concrete, 98
 - early definitions in, 99
 - equality in, 103
 - implementing abstract methods in, 123
 - mixing traits into, 288–289
- subsetOf method (BitSet), 177
- Success class, 251, 315
- success combinator, 313
- success method (Promise), 259
- successful method (Future), 258
- sum method
 - of ArrayBuffer, 40
 - of Iterable, 180
 - of Iterator, 189
- super keyword, 92–93, 126
- super keyword (Java), 95
- superclasses, 133–134
 - abstract fields in, 98
 - constructing, 94–95
 - extending, 137
 - methods of:
 - abstract, 97
 - clashing with subclass methods, 92
 - overriding, 96
 - no multiple inheritance of, 121, 124, 129
 - scope of, 282
 - sealed, 209

supertypes, 18, 42, 102
Swing toolkit, 137–138
switch statement, 19, 198
@switch annotation, 223–224
symbols, 226
synchronized method (AnyRef), 100
syntactic sugar, 275, 286

T

tab completion, 3
tail method
 of Iterable, 180
 of lists, 175–176
TailCalls, TailRec objects, 223
@tailrec annotation, 222
take, takeWhile methods
 of Iterable, 180
 of Iterator, 189
takeRight method (Iterable), 180
@Test annotation, 217
Text class, 233
 pattern matching for, 237
text method, 232
this keyword, 42, 62, 65, 134–135, 280, 288–289, 295
 aliases for, 68, 289
 scope of, 282
 with private, 59
 with protected, 94
thread pool, 248
threads
 assigning tasks to, 248
 blocking, 250–251
throw expression, 30
Throwable class (Java), 29
@throws annotation, 220
to method (RichInt), 6, 22, 175
toArray method
 of ArrayBuffer, 37
 of Iterable, 181
 of Iterator, 189
 of Source, 110
toBuffer method
 of Array, 37
 of Source, 110
toChar method, 6
toIndexedSeq, toIterable methods
 of Iterable, 181
 of Iterator, 189
toInt, toDouble methods, 6, 111
token method (StdLexical), 318
Token type, 315
tokens, 304
 discarding, 308–309
 matching against regular expressions, 305
Tokens trait, 317
toList, toStream methods
 of Iterable, 181
 of Iterator, 189
toMap method, 52, 173
 of Iterable, 181
 of Iterator, 189
toSeq, toSet methods, 173
 of Iterable, 181
 of Iterator, 189
toString method, 6, 41, 76, 233
 of case classes, 205, 208–209
trait keyword, 123, 287
traits, 123–137, 287
 abstract types in, 291
 adding to objects, 125
 construction order of, 126–127, 130–132
 dependency injections in, 135, 290–291
 extending, 73
 classes, 133–134
 superclass, 137
 fields in:
 abstract, 130, 132
 concrete, 128–129
 for collections, 172–173
 for rich interfaces, 127
 implementing, 124, 269
 layered, 125–126
 methods of, 124–126, 135
 overriding, 127
 unimplemented, 123
 mixing into classes, 288–289
 no constructor parameters in, 132–133
 self types in, 134–135
 type parameters in, 266
 universal, 104
 vs. classes, 132
 vs. Java interfaces, 121–124, 135
 vs. structural types, 284
trampolining, 223
transform method
 of ArrayBuffer, 183
 of RewriteRule, 239
transform, transformWith methods (Future), 256
@transient annotation, 219
Traversable, TraversableOnce traits, 41
TraversableLike trait, 334
traverse method (Future), 257

- trimEnd method, 36
 - Try class, 31, 251
 - flatMap, map methods, 255
 - get method, 251
 - isSuccess, isFailure methods, 251
 - try statement, 30–31
 - exceptions in, 167
 - trySuccess method (Promise), 259
 - tuples, 47, 51–52, 287
 - accessing components of, 51
 - converting to maps, 52
 - pattern matching for, 201–202
 - zipping, 52
 - type classes, 331–333
 - type constraints, 269–271, 333
 - type constructors, 296–299
 - type errors, 150, 273
 - type inference, 270
 - type keyword, 280–281, 283, 287
 - type parameters, 14, 97, 265–275, 287, 328
 - annotated, 217, 225
 - context bounds of, 329–330
 - implicit conversions for, 268
 - not possible for objects, 274
 - structural, 283–284
 - vs. abstract types, 292
 - type projections, 67, 281–283, 287
 - in forSome blocks, 286
 - type variables, bounds for, 266–269
 - TypeAnnotation trait, 218
 - types, 5–6, 279–299
 - abstract, 283, 291–293, 315
 - aliases for, 174, 283
 - annotated, 217
 - anonymous, 98
 - checking, 93–94
 - compound, 284–287
 - converting, 6, 270
 - dynamic, 150
 - equality of, 270
 - existential, 287
 - generic, 269, 271–275
 - implementing multiple traits, 269
 - infix, 285–287
 - invariant, 272
 - naming, 296
 - pattern matching by, 200–201
 - primitive, 5, 36, 225
 - self, 134–135, 288–295
 - structural, 135, 285
 - subtypes of, 270
 - supertypes of, 18, 102
 - wrapper, 6
- ## U
- ul element (XML), 233–234
 - unapply method, 147–150, 202, 206–207
 - of case classes, 205
 - unapplySeq method, 149–150, 202
 - unary operators, 143–144
 - unary_- method (BigInt), 14
 - unary_op methods, 143
 - @unchecked annotation, 218, 226
 - @uncheckedVariance annotation, 226
 - Unicode characters, 142
 - uniform access principle, 58
 - uniform creation principle, 173
 - uniform return type principle, 182
 - union method (sets), 177
 - Unit class, 28, 100, 102
 - value of, 18–21
 - unit method (Future), 258
 - universal traits, 104
 - Unparsed type, 235
 - until method, 175
 - until statement, 166
 - update method, 146–147
 - updateDynamic method (Dynamic), 151–152
 - URIs (Uniform Resource Identifiers), 242
 - URLs (Uniform Resource Locators)
 - loading files from, 239
 - reading from, 111
 - redirecting input from, 115
- ## V
- val fields, 4
 - declarations of, 4–5
 - early definitions of, 99
 - final, 99
 - generated methods for, 59, 61–62, 65
 - in forSome blocks, 286
 - in parsers, 314
 - initializing, 5, 20, 28–29
 - lazy, 28–29, 99, 133, 314
 - overriding, 95–96, 98
 - private, 62
 - scope of, 282
 - specifying type of, 5
 - storing functions in, 157–159
 - value classes, 103–105, 208

Value method (enumerations), 75–76
value method (Future), 250
valueAtOneQuarter function, 160
values
 binding to variables, 207
 naming, 200
 printing, 21
values method (Map), 50
van der Linden, Peter, 198
var fields, 4
 annotated, 216
 declarations of, 4–5
 extractors in, 147
 pattern matching in, 203–204
 generated methods for, 61–62, 65
 initializing, 5
 no path elements in, 282
 overriding, 96
 private, 62
 specifying type of, 5, 266
 updating, 49
 vs. function calls, in parsers, 313
@varargs annotation, 221
variables
 binding to:
 node sequences, 237
 values, 207
 declaring as Java SAM interfaces, 164
 in case clauses, 199–200
 naming, 142, 200
 within loops, 24
variance, 226
Vector class, 174–175
vector type (C++), 36
view bounds, 268
view method, 190–191
void keyword (C++, Java), 18, 20, 102
@volatile annotation, 219

W

wait method (AnyRef), 100
walk method (java.nio.file.Files), 112
walkFileTree method (Java), 112–113
warnings, 226
while loop, 22, 166
whitespace
 in lexical analysis, 304
 parsing, 240, 316–317
wildcards, 275
 for XML elements, 236
 in catch statements, 30

 in imports, 8, 85–86
 in Java, 85, 286
with keyword, 99, 124–125, 284–287
wrapper types, 6
write method (XML), 241
Writer class (Java), 241

X

XHTML (Extensible Hypertext Markup Language), 235
XhtmlParser class, 241
XML (Extensible Markup Language), 229–243
 attributes in, 232–235, 238
 character references in, 232
 comments in, 231
 elements in, 238, 242
 entity references in, 231–232, 241
 including non-XML text into, 235
 indentation in, 242
 loading, 239
 malformed, 235
 namespaces in, 242–243
 nodes in, 230–232
 processing instructions in, 231
 saving, 233, 241–242
 self-closing tags in, 242
 transforming, 239
XML declarations, 241
XML literals
 braces in, 234
 defining, 230
 embedded expressions in, 233–234
 entity references in, 232
 in nested Scala code, 233
 in pattern matching, 237–238
XPath (XML Path language), 235–236

Y

yield keyword
 as Java method, 142
 in loops, 24, 38–39

Z

zip method, 52, 181, 187–189
zip, zipWith methods (Future), 255
zipAll, zipWithIndex methods, 187
 of Iterable, 181
 of Iterator, 189
zipping, 187–188