



Core Java[®]

Covers
Java SE 8

for the Impatient



Cay S. Horstmann

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

Core Java® for the Impatient

This page intentionally left blank

Core Java[®] for the Impatient

Cay S. Horstmann

◆◆Addison-Wesley

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

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

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

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 international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Horstmann, Cay S., 1959-

Core Java for the impatient / Cay S. Horstmann.

pages cm

Includes index.

ISBN 978-0-321-99632-9 (pbk. : alk. paper)—ISBN 0-321-99632-1 (pbk. : alk. paper)

1. Java (Computer program language) I. Title.

QA76.73.J38H67535 2015

005.13'3—dc23

2014046523

Copyright © 2015 Pearson Education, Inc.

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

ISBN-13: 978-0-321-99632-9

ISBN-10: 0-321-99632-1

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

First printing, February 2015

To Chi—the most patient person in my life.

This page intentionally left blank

Contents

Preface	xxi
Acknowledgments	xxiii
About the Author	xxv

1	FUNDAMENTAL PROGRAMMING STRUCTURES	1
1.1	Our First Program	2
1.1.1	Dissecting the “Hello, World” Program	2
1.1.2	Compiling and Running a Java Program	3
1.1.3	Method Calls	5
1.2	Primitive Types	7
1.2.1	Integer Types	7
1.2.2	Floating-Point Types	8
1.2.3	The char Type	9
1.2.4	The boolean Type	10
1.3	Variables	10
1.3.1	Variable Declarations	10
1.3.2	Names	11

1.3.3	Initialization	11
1.3.4	Constants	12
1.4	Arithmetic Operations	13
1.4.1	Assignment	14
1.4.2	Basic Arithmetic	14
1.4.3	Mathematical Methods	15
1.4.4	Number Type Conversions	16
1.4.5	Relational and Logical Operators	18
1.4.6	Big Numbers	19
1.5	Strings	20
1.5.1	Concatenation	20
1.5.2	Substrings	21
1.5.3	String Comparison	21
1.5.4	Converting Between Numbers and Strings	23
1.5.5	The String API	24
1.5.6	Code Points and Code Units	25
1.6	Input and Output	26
1.6.1	Reading Input	27
1.6.2	Formatted Output	28
1.7	Control Flow	30
1.7.1	Branches	30
1.7.2	Loops	32
1.7.3	Breaking and Continuing	34
1.7.4	Local Variable Scope	36
1.8	Arrays and Array Lists	37
1.8.1	Working with Arrays	37
1.8.2	Array Construction	38
1.8.3	Array Lists	39
1.8.4	Wrapper Classes for Primitive Types	40
1.8.5	The Enhanced for Loop	41
1.8.6	Copying Arrays and Array Lists	41
1.8.7	Array Algorithms	42
1.8.8	Command-Line Arguments	43
1.8.9	Multidimensional Arrays	44

- 1.9 Functional Decomposition 46
 - 1.9.1 Declaring and Calling Static Methods 47
 - 1.9.2 Array Parameters and Return Values 47
 - 1.9.3 Variable Arguments 48
- Exercises 49

2 OBJECT-ORIENTED PROGRAMMING 53

- 2.1 Working with Objects 54
 - 2.1.1 Accessor and Mutator Methods 56
 - 2.1.2 Object References 56
- 2.2 Implementing Classes 58
 - 2.2.1 Instance Variables 59
 - 2.2.2 Method Headers 59
 - 2.2.3 Method Bodies 60
 - 2.2.4 Instance Method Invocations 60
 - 2.2.5 The `this` Reference 61
 - 2.2.6 Call by Value 62
- 2.3 Object Construction 63
 - 2.3.1 Implementing Constructors 63
 - 2.3.2 Overloading 64
 - 2.3.3 Calling One Constructor from Another 65
 - 2.3.4 Default Initialization 65
 - 2.3.5 Instance Variable Initialization 66
 - 2.3.6 Final Instance Variables 67
 - 2.3.7 The Constructor with No Arguments 68
- 2.4 Static Variables and Methods 68
 - 2.4.1 Static Variables 68
 - 2.4.2 Static Constants 69
 - 2.4.3 Static Initialization Blocks 70
 - 2.4.4 Static Methods 71
 - 2.4.5 Factory Methods 72
- 2.5 Packages 72
 - 2.5.1 Package Declarations 73
 - 2.5.2 The Class Path 74

2.5.3	Package Scope	76
2.5.4	Importing Classes	77
2.5.5	Static Imports	78
2.6	Nested Classes	79
2.6.1	Static Nested Classes	79
2.6.2	Inner Classes	81
2.6.3	Special Syntax Rules for Inner Classes	83
2.7	Documentation Comments	84
2.7.1	Comment Insertion	85
2.7.2	Class Comments	85
2.7.3	Method Comments	86
2.7.4	Variable Comments	86
2.7.5	General Comments	87
2.7.6	Links	87
2.7.7	Package and Overview Comments	88
2.7.8	Comment Extraction	88
	Exercises	89

3 INTERFACES AND LAMBDA EXPRESSIONS 93

3.1	Interfaces	94
3.1.1	Declaring an Interface	94
3.1.2	Implementing an Interface	95
3.1.3	Converting to an Interface Type	97
3.1.4	Casts and the instanceof Operator	97
3.1.5	Extending Interfaces	98
3.1.6	Implementing Multiple Interfaces	98
3.1.7	Constants	99
3.2	Static and Default Methods	99
3.2.1	Static Methods	99
3.2.2	Default Methods	100
3.2.3	Resolving Default Method Conflicts	101
3.3	Examples of Interfaces	102
3.3.1	The Comparable Interface	103
3.3.2	The Comparator Interface	104

3.3.3	The Runnable Interface	105
3.3.4	User Interface Callbacks	106
3.4	Lambda Expressions	107
3.4.1	The Syntax of Lambda Expressions	107
3.4.2	Functional Interfaces	109
3.5	Method and Constructor References	110
3.5.1	Method References	110
3.5.2	Constructor References	111
3.6	Processing Lambda Expressions	112
3.6.1	Implementing Deferred Execution	112
3.6.2	Choosing a Functional Interface	113
3.6.3	Implementing Your Own Functional Interfaces	115
3.7	Lambda Expressions and Variable Scope	116
3.7.1	Scope of a Lambda Expression	116
3.7.2	Accessing Variables from the Enclosing Scope	117
3.8	Higher-Order Functions	120
3.8.1	Methods that Return Functions	120
3.8.2	Methods That Modify Functions	120
3.8.3	Comparator Methods	121
3.9	Local Inner Classes	122
3.9.1	Local Classes	122
3.9.2	Anonymous Classes	123
	Exercises	124

4 INHERITANCE AND REFLECTION 127

4.1	Extending a Class	128
4.1.1	Super- and Subclasses	128
4.1.2	Defining and Inheriting Subclass Methods	129
4.1.3	Method Overriding	129
4.1.4	Subclass Construction	131
4.1.5	Superclass Assignments	131
4.1.6	Casts	132
4.1.7	Final Methods and Classes	133
4.1.8	Abstract Methods and Classes	133

4.1.9	Protected Access	134
4.1.10	Anonymous Subclasses	135
4.1.11	Inheritance and Default Methods	136
4.1.12	Method Expressions with <code>super</code>	137
4.2	Object: The Cosmic Superclass	137
4.2.1	The <code>toString</code> Method	138
4.2.2	The <code>equals</code> Method	140
4.2.3	The <code>hashCode</code> Method	143
4.2.4	Cloning Objects	144
4.3	Enumerations	147
4.3.1	Methods of Enumerations	147
4.3.2	Constructors, Methods, and Fields	149
4.3.3	Bodies of Instances	149
4.3.4	Static Members	150
4.3.5	Switching on an Enumeration	151
4.4	Runtime Type Information and Resources	151
4.4.1	The Class Class	152
4.4.2	Loading Resources	155
4.4.3	Class Loaders	155
4.4.4	The Context Class Loader	157
4.4.5	Service Loaders	159
4.5	Reflection	160
4.5.1	Enumerating Class Members	160
4.5.2	Inspecting Objects	161
4.5.3	Invoking Methods	162
4.5.4	Constructing Objects	163
4.5.5	JavaBeans	164
4.5.6	Working with Arrays	165
4.5.7	Proxies	167
	Exercises	169

5 EXCEPTIONS, ASSERTIONS, AND LOGGING 173

5.1	Exception Handling	174
5.1.1	Throwing Exceptions	174

5.1.2	The Exception Hierarchy	175
5.1.3	Declaring Checked Exceptions	177
5.1.4	Catching Exceptions	178
5.1.5	The Try-with-Resources Statement	179
5.1.6	The finally Clause	181
5.1.7	Rethrowing and Chaining Exceptions	182
5.1.8	The Stack Trace	184
5.1.9	The <code>Objects.requireNonNull</code> Method	185
5.2	Assertions	185
5.2.1	Using Assertions	185
5.2.2	Enabling and Disabling Assertions	186
5.3	Logging	187
5.3.1	Using Loggers	187
5.3.2	Loggers	187
5.3.3	Logging Levels	188
5.3.4	Other Logging Methods	189
5.3.5	Logging Configuration	190
5.3.6	Log Handlers	191
5.3.7	Filters and Formatters	194
	Exercises	194

6 GENERIC PROGRAMMING 199

6.1	Generic Classes	200
6.2	Generic Methods	201
6.3	Type Bounds	202
6.4	Type Variance and Wildcards	203
6.4.1	Subtype Wildcards	204
6.4.2	Supertype Wildcards	205
6.4.3	Wildcards with Type Variables	206
6.4.4	Unbounded Wildcards	207
6.4.5	Wildcard Capture	208
6.5	Generics in the Java Virtual Machine	208
6.5.1	Type Erasure	209

6.5.2	Cast Insertion	209
6.5.3	Bridge Methods	210
6.6	Restrictions on Generics	211
6.6.1	No Primitive Type Arguments	212
6.6.2	At Runtime, All Types Are Raw	212
6.6.3	You Cannot Instantiate Type Variables	213
6.6.4	You Cannot Construct Arrays of Parameterized Types	215
6.6.5	Class Type Variables Are Not Valid in Static Contexts	216
6.6.6	Methods May Not Clash after Erasure	216
6.6.7	Exceptions and Generics	217
6.7	Reflection and Generics	218
6.7.1	The <code>Class<T></code> Class	219
6.7.2	Generic Type Information in the Virtual Machine	220
	Exercises	221

7 COLLECTIONS 227

7.1	An Overview of the Collections Framework	228
7.2	Iterators	232
7.3	Sets	233
7.4	Maps	235
7.5	Other Collections	238
7.5.1	Properties	238
7.5.2	Bit Sets	240
7.5.3	Enumeration Sets and Maps	241
7.5.4	Stacks, Queues, Deques, and Priority Queues	242
7.5.5	Weak Hash Maps	243
7.6	Views	244
7.6.1	Ranges	244
7.6.2	Empty and Singleton Views	244
7.6.3	Unmodifiable Views	245
	Exercises	246

8	STREAMS	249
8.1	From Iterating to Stream Operations	250
8.2	Stream Creation	251
8.3	The filter, map, and flatMap Methods	252
8.4	Extracting Substreams and Combining Streams	254
8.5	Other Stream Transformations	254
8.6	Simple Reductions	255
8.7	The Optional Type	256
8.7.1	How to Work with Optional Values	256
8.7.2	How Not to Work with Optional Values	257
8.7.3	Creating Optional Values	258
8.7.4	Composing Optional Value Functions with flatMap	258
8.8	Collecting Results	259
8.9	Collecting into Maps	260
8.10	Grouping and Partitioning	262
8.11	Downstream Collectors	262
8.12	Reduction Operations	264
8.13	Primitive Type Streams	266
8.14	Parallel Streams	267
	Exercises	269
9	PROCESSING INPUT AND OUTPUT	273
9.1	Input/Output Streams, Readers, and Writers	274
9.1.1	Obtaining Streams	274
9.1.2	Reading Bytes	275
9.1.3	Writing Bytes	276
9.1.4	Character Encodings	276
9.1.5	Text Input	279
9.1.6	Text Output	280
9.1.7	Reading and Writing Binary Data	281
9.1.8	Random-Access Files	282
9.1.9	Memory-Mapped Files	283
9.1.10	File Locking	283

9.2	Paths, Files, and Directories	284
9.2.1	Paths	284
9.2.2	Creating Files and Directories	286
9.2.3	Copying, Moving, and Deleting Files	287
9.2.4	Visiting Directory Entries	288
9.2.5	ZIP File Systems	291
9.3	URL Connections	292
9.4	Regular Expressions	293
9.4.1	The Regular Expression Syntax	293
9.4.2	Finding One or All Matches	298
9.4.3	Groups	298
9.4.4	Removing or Replacing Matches	299
9.4.5	Flags	300
9.5	Serialization	301
9.5.1	The Serializable Interface	301
9.5.2	Transient Instance Variables	303
9.5.3	The readObject and writeObject Methods	303
9.5.4	The readResolve and writeReplace Methods	304
9.5.5	Versioning	306
	Exercises	307

10 CONCURRENT PROGRAMMING 311

10.1	Concurrent Tasks	312
10.1.1	Running Tasks	312
10.1.2	Futures and Executor Services	314
10.2	Thread Safety	317
10.2.1	Visibility	317
10.2.2	Race Conditions	319
10.2.3	Strategies for Safe Concurrency	321
10.2.4	Immutable Classes	322
10.3	Parallel Algorithms	323
10.3.1	Parallel Streams	323
10.3.2	Parallel Array Operations	324

10.4	Threadsafe Data Structures	324
10.4.1	Concurrent Hash Maps	325
10.4.2	Blocking Queues	326
10.4.3	Other Threadsafe Data Structures	328
10.5	Atomic Values	329
10.6	Locks	331
10.6.1	Reentrant Locks	331
10.6.2	The <code>synchronized</code> Keyword	333
10.6.3	Waiting on Conditions	335
10.7	Threads	337
10.7.1	Starting a Thread	337
10.7.2	Thread Interruption	338
10.7.3	Thread-Local Variables	339
10.7.4	Miscellaneous Thread Properties	340
10.8	Asynchronous Computations	341
10.8.1	Long-Running Tasks in User Interface Callbacks	341
10.8.2	Completable Futures	342
10.9	Processes	345
10.9.1	Building a Process	345
10.9.2	Running a Process	347
	Exercises	348

11 ANNOTATIONS 355

11.1	Using Annotations	356
11.1.1	Annotation Elements	356
11.1.2	Multiple and Repeated Annotations	358
11.1.3	Annotating Declarations	358
11.1.4	Annotating Type Uses	359
11.1.5	Making Receivers Explicit	360
11.2	Defining Annotations	361
11.3	Standard Annotations	364
11.3.1	Annotations for Compilation	365
11.3.2	Annotations for Managing Resources	366
11.3.3	Meta-Annotations	366

- 11.4 Processing Annotations at Runtime 368
- 11.5 Source-Level Annotation Processing 371
 - 11.5.1 Annotation Processors 372
 - 11.5.2 The Language Model API 372
 - 11.5.3 Using Annotations to Generate Source Code 373
- Exercises 376

12 THE DATE AND TIME API 379

- 12.1 The Time Line 380
- 12.2 Local Dates 382
- 12.3 Date Adjusters 385
- 12.4 Local Time 386
- 12.5 Zoned Time 387
- 12.6 Formatting and Parsing 390
- 12.7 Interoperating with Legacy Code 393
- Exercises 394

13 INTERNATIONALIZATION 397

- 13.1 Locales 398
 - 13.1.1 Specifying a Locale 399
 - 13.1.2 The Default Locale 401
 - 13.1.3 Display Names 402
- 13.2 Number Formats 403
- 13.3 Currencies 403
- 13.4 Date and Time Formatting 404
- 13.5 Collation and Normalization 406
- 13.6 Message Formatting 408
- 13.7 Resource Bundles 410
 - 13.7.1 Organizing Resource Bundles 410
 - 13.7.2 Bundle Classes 412
- 13.8 Character Encodings 413
- 13.9 Preferences 413
- Exercises 415

14	COMPILING AND SCRIPTING	419
14.1	The Compiler API	420
14.1.1	Invoking the Compiler	420
14.1.2	Launching a Compilation Task	420
14.1.3	Reading Source Files from Memory	421
14.1.4	Writing Byte Codes to Memory	422
14.1.5	Capturing Diagnostics	423
14.2	The Scripting API	424
14.2.1	Getting a Scripting Engine	424
14.2.2	Bindings	425
14.2.3	Redirecting Input and Output	426
14.2.4	Calling Scripting Functions and Methods	426
14.2.5	Compiling a Script	428
14.3	The Nashorn Scripting Engine	428
14.3.1	Running Nashorn from the Command Line	429
14.3.2	Invoking Getters, Setters, and Overloaded Methods	430
14.3.3	Constructing Java Objects	431
14.3.4	Strings in JavaScript and Java	432
14.3.5	Numbers	432
14.3.6	Working with Arrays	433
14.3.7	Lists and Maps	434
14.3.8	Lambdas	435
14.3.9	Extending Java Classes and Implementing Java Interfaces	435
14.3.10	Exceptions	437
14.4	Shell Scripting with Nashorn	437
14.4.1	Executing Shell Commands	438
14.4.2	String Interpolation	438
14.4.3	Script Inputs	439
	Exercises	440
	Index	443

This page intentionally left blank

Preface

Java is now about twenty years old, and the classic book, *Core Java™*, covers, in meticulous detail, not just the language but all core libraries and a multitude of changes between versions, spanning two volumes and well over 2,000 pages. But Java 8 changes everything. Many of the old Java idioms are no longer required, and there is a much faster, easier pathway for learning Java. In this book, I show you the “good parts” of modern Java so you can put them to work quickly.

As with my previous “Impatient” books, I quickly cut to the chase, showing you what you need to know for solving a programming problem without lecturing about the superiority of one paradigm over another. I also present the information in small chunks, organized so that you can quickly retrieve it when needed.

Assuming you are proficient in some other programming language, such as C++, JavaScript, Objective C, PHP, or Ruby, with this book you will learn how to become a competent Java programmer. I cover all aspects of Java that a developer needs to know, including the powerful lambda expressions and streams that were introduced in Java 8. I tell you where to find out more about old-fashioned concepts that you might still see in legacy code, but I don’t dwell on them.

A key reason to use Java is to tackle concurrent programming. With parallel algorithms and threadsafe data structures readily available in the Java library, the way application programmers should handle concurrent programming has completely changed. I provide fresh coverage, showing you how to use the powerful library features instead of error-prone, low-level constructs.

Traditional books on Java are focused on user interface programming—but nowadays, few developers produce user interfaces on desktop computers. If you intend to use Java for server-side programming or Android programming, you will be able to use this book effectively without being distracted by desktop GUI code.

Finally, this book is written for application programmers, not for a college course and not for systems wizards. The book covers issues that application programmers need to wrestle with, such as logging and working with files—but you won't learn how to implement a linked list by hand or how to write a web server.

I hope you enjoy this rapid-fire introduction into modern Java, and I hope it will make your work with Java productive and enjoyable.

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

Acknowledgments

My thanks go, as always, to my editor Greg Doench, who enthusiastically supported the vision of a short book that gives a fresh introduction to Java 8. Dmitry Kirsanov and Alina Kirsanova once again turned an XHTML manuscript into an attractive book with amazing speed and attention to detail. My special gratitude goes to the excellent team of reviewers who spotted many errors and gave thoughtful suggestions for improvement. They are: Andres Almiray, Brian Goetz, Marty Hall, Mark Lawrence, Doug Lea, Simon Ritter, Yoshiki Shibata, and Christian Ullenboom.

*Cay Horstmann
Biel/Bienne, Switzerland
January 2015*

This page intentionally left blank

About the Author

Cay S. Horstmann is the author of *Java SE 8 for the Really Impatient* and *Scala for the Impatient* (both from Addison-Wesley), is principal author of *Core Java™, Volumes I and II, Ninth Edition* (Prentice Hall, 2013), and has written a dozen other books for professional programmers and computer science students. He is a professor of computer science at San Jose State University and is a Java Champion.

Chapter

3

Java was designed as an object-oriented programming language in the 1990s when object-oriented programming was the principal paradigm for software development. Interfaces are a key feature of object-oriented programming: They let you specify what should be done, without having to provide an implementation.

Long before there was object-oriented programming, there were functional programming languages, such as Lisp, in which functions and not objects are the primary structuring mechanism. Recently, functional programming has risen in importance because it is well suited for concurrent and event-driven (or “reactive”) programming. Java supports function expressions that provide a convenient bridge between object-oriented and functional programming. In this chapter, you will learn about interfaces and lambda expressions.

The key points of this chapter are:

- An interface specifies a set of methods that an implementing class must provide.
- An interface is a supertype of any class that implements it. Therefore, one can assign instances of the class to variables of the interface type.
- An interface can contain static methods. All variables of an interface are automatically static and final.
- An interface can contain default methods that an implementing class can inherit or override.

- The `Comparable` and `Comparator` interfaces are used for comparing objects.
- A lambda expression denotes a block of code that can be executed at a later point in time.
- Lambda expressions are converted to functional interfaces.
- Method and constructor references refer to methods or constructors without invoking them.
- Lambda expressions and local inner classes can access effectively final variables from the enclosing scope.

3.1 Interfaces

An *interface* is a mechanism for spelling out a contract between two parties: the supplier of a service and the classes that want their objects to be usable with the service. In the following sections, you will see how to define and use interfaces in Java.

3.1.1 Declaring an Interface

Consider a service that works on sequences of integers, reporting the average of the first n values:

```
public static double average(IntSequence seq, int n)
```

Such sequences can take many forms. Here are some examples:

- A sequence of integers supplied by a user
- A sequence of random integers
- The sequence of prime numbers
- The sequence of elements in an integer array
- The sequence of code points in a string
- The sequence of digits in a number

We want to implement *a single mechanism* for deal with all these kinds of sequences.

First, let us spell out what is common between integer sequences. At a minimum, one needs two methods for working with a sequence:

- Test whether there is a next element
- Get the next element

To declare an interface, you provide the method headers, like this:

```
public interface IntSequence {
    boolean hasNext();
    int next();
}
```

You need not implement these methods, but you can provide default implementations if you like—see Section 3.2.2, “Default Methods,” on p. 100. If no implementation is provided, we say that the method is *abstract*.



NOTE: All methods of an interface are automatically public. Therefore, it is not necessary to declare `hasNext` and `next` as public. Some programmers do it anyway for greater clarity.

The methods in the interface suffice to implement the average method:

```
public static double average(IntSequence seq, int n) {
    int count = 0;
    double sum = 0;
    while (seq.hasNext() && count < n) {
        count++;
        sum += seq.next();
    }
    return count == 0 ? 0 : sum / count;
}
```

3.1.2 Implementing an Interface

Now let’s look at the other side of the coin: the classes that want to be usable with the average method. They need to *implement* the `IntSequence` interface. Here is such a class:

```
public class SquareSequence implements IntSequence {
    private int i;

    public boolean hasNext() {
        return true;
    }

    public int next() {
        i++;
        return i * i;
    }
}
```

There are infinitely many squares, and an object of this class delivers them all, one at a time.

The `implements` keyword indicates that the `SquareSequence` class intends to conform to the `IntSequence` interface.



CAUTION: The implementing class must declare the methods of the interface as `public`. Otherwise, they would default to package access. Since the interface requires public access, the compiler would report an error.

This code gets the average of the first 100 squares:

```
SquareSequence squares = new SquareSequence();  
double avg = average(squares, 100);
```

There are many classes that can implement the `IntSequence` interface. For example, this class yields a finite sequence, namely the digits of a positive integer starting with the least significant one:

```
public class DigitSequence implements IntSequence {  
    private int number;  
  
    public DigitSequence(int n) {  
        number = n;  
    }  
  
    public boolean hasNext() {  
        return number != 0;  
    }  
  
    public int next() {  
        int result = number % 10;  
        number /= 10;  
        return result;  
    }  
  
    public int rest() {  
        return number;  
    }  
}
```

An object `new DigitSequence(1729)` delivers the digits 9 2 7 1 before `hasNext` returns false.



NOTE: The `SquareSequence` and `DigitSequence` classes implement all methods of the `IntSequence` interface. If a class only implements some of the methods, then it must be declared with the `abstract` modifier. See Chapter 4 for more information on abstract classes.

3.1.3 Converting to an Interface Type

This code fragment computes the average of the digit sequence values:

```
IntSequence digits = new DigitSequence(1729);
double avg = average(digits, 100);
// Will only look at the first four sequence values
```

Look at the `digits` variable. Its type is `IntSequence`, not `DigitSequence`. A variable of type `IntSequence` refers to an object of some class that implements the `IntSequence` interface. You can always assign an object to a variable whose type is an implemented interface, or pass it to a method expecting such an interface.

Here is a bit of useful terminology. A type *S* is a *supertype* of the type *T* (the *subtype*) when any value of the subtype can be assigned to a variable of the supertype without a conversion. For example, the `IntSequence` interface is a supertype of the `DigitSequence` class.



NOTE: Even though it is possible to declare variables of an interface type, you can never have an object whose type is an interface. All objects are instances of classes.

3.1.4 Casts and the instanceof Operator

Occasionally, you need the opposite conversion—from a supertype to a subtype. Then you use a *cast*. For example, if you happen to know that the object stored in an `IntSequence` is actually a `DigitSequence`, you can convert the type like this:

```
IntSequence sequence = ...;
DigitSequence digits = (DigitSequence) sequence;
System.out.println(digits.rest());
```

In this scenario, the cast was necessary because `rest` is a method of `DigitSequence` but not `IntSequence`.

See Exercise 2 for a more compelling example.

You can only cast an object to its actual class or one of its supertypes. If you are wrong, a compile-time error or class cast exception will occur:

```
String digitString = (String) sequence;
// Cannot possibly work—IntSequence is not a supertype of String
RandomSequence randomness = (RandomSequence) sequence;
// Could work, throws a class cast exception if not
```

To avoid the exception, you can first test whether the object is of the desired type, using the `instanceof` operator. The expression

```
object instanceof Type
```

returns true if *object* is an instance of a class that has *Type* as a supertype. It is a good idea to make this check before using a cast.

```
if (sequence instanceof DigitSequence) {
    DigitSequence digits = (DigitSequence) sequence;
    ...
}
```

3.1.5 Extending Interfaces

An interface can *extend* another, providing additional methods on top of the original ones. For example, `Closeable` is an interface with a single method:

```
public interface Closeable {
    void close();
}
```

As you will see in Chapter 5, this is an important interface for closing resources when an exception occurs.

The `Channel` interface extends this interface:

```
public interface Channel extends Closeable {
    boolean isOpen();
}
```

A class that implements the `Channel` interface must provide both methods, and its objects can be converted to both interface types.

3.1.6 Implementing Multiple Interfaces

A class can implement any number of interfaces. For example, a `FileSequence` class that reads integers from a file can implement the `Closeable` interface in addition to `IntSequence`:

```
public class FileSequence implements IntSequence, Closeable {  
    ...  
}
```

Then the `FileSequence` class has both `IntSequence` and `Closeable` as supertypes.

3.1.7 Constants

Any variable defined in an interface is automatically `public static final`.

For example, the `SwingConstants` interface defines constants for compass directions:

```
public interface SwingConstants {  
    int NORTH = 1;  
    int NORTH_EAST = 2;  
    int EAST = 3;  
    ...  
}
```

You can refer to them by their qualified name, `SwingConstants.NORTH`. If your class chooses to implement the `SwingConstants` interface, you can drop the `SwingConstants` qualifier and simply write `NORTH`. However, this is not a common idiom. It is far better to use enumerations for a set of constants; see Chapter 4.



NOTE: You cannot have instance variables in an interface. An interface specifies behavior, not object state.

3.2 Static and Default Methods

In earlier versions of Java, all methods of an interface had to be abstract—that is, without a body. Nowadays you can add two kinds of methods with a concrete implementation: static and default methods. The following sections describe these methods.

3.2.1 Static Methods

There was never a technical reason why an interface could not have static methods, but they did not fit into the view of interfaces as abstract specifications. That thinking has now evolved. In particular, factory methods make a lot of sense in interfaces. For example, the `IntSequence` interface can have a static method `digitsOf` that generates a sequence of digits of a given integer:

```
IntSequence digits = IntSequence.digitsOf(1729);
```

The method yields an instance of some class implementing the `IntSequence` interface, but the caller need not care which one it is.

```
public interface IntSequence {  
    ...  
    public static IntSequence digitsOf(int n) {  
        return new DigitSequence(n);  
    }  
}
```



NOTE: In the past, it had been common to place static methods in a companion class. You find pairs of interfaces and utility classes, such as `Collection/Collections` or `Path/Paths`, in the standard library. This split is no longer necessary.

3.2.2 Default Methods

You can supply a *default* implementation for any interface method. You must tag such a method with the `default` modifier.

```
public interface IntSequence {  
    default boolean hasNext() { return true; }  
    // By default, sequences are infinite  
    int next();  
}
```

A class implementing this interface can choose to override the `hasNext` method or to inherit the default implementation.



NOTE: Default methods put an end to the classic pattern of providing an interface and a companion class that implements most or all of its methods, such as `Collection/AbstractCollection` or `WindowListener/WindowAdapter` in the Java API. Nowadays you should just implement the methods in the interface.

An important use for default methods is *interface evolution*. Consider for example the `Collection` interface that has been a part of Java for many years. Suppose that way back when, you provided a class

```
public class Bag implements Collection
```

Later, in Java 8, a `stream` method was added to the interface.

Suppose the `stream` method was not a default method. Then the `Bag` class no longer compiles since it doesn't implement the new method. Adding a nondefault method to an interface is not *source-compatible*.

But suppose you don't recompile the class and simply use an old JAR file containing it. The class will still load, even with the missing method. Programs can still construct `Bag` instances, and nothing bad will happen. (Adding a method to an interface is *binary-compatible*.) However, if a program calls the `stream` method on a `Bag` instance, an `AbstractMethodError` occurs.

Making the method a default method solves both problems. The `Bag` class will again compile. And if the class is loaded without being recompiled and the `stream` method is invoked on a `Bag` instance, the `Collection.stream` method is called.

3.2.3 Resolving Default Method Conflicts

If a class implements two interfaces, one of which has a default method and the other a method (default or not) with the same name and parameter types, then you must resolve the conflict. This doesn't happen very often, and it is usually easy to deal with the situation.

Let's look at an example. Suppose we have an interface `Person` with a `getId` method:

```
public interface Person {  
    String getName();  
    default int getId() { return 0; }  
}
```

And suppose there is an interface `Identified`, also with such a method.

```
public interface Identified {  
    default int getId() { return Math.abs(hashCode()); }  
}
```

You will see what the `hashCode` method does in Chapter 4. For now, all that matters is that it returns some integer that is derived from the object.

What happens if you form a class that implements both of them?

```
public class Employee implements Person, Identified {  
    ...  
}
```

The class inherits two `getId` methods provided by the `Person` and `Identified` interfaces. There is no way for the Java compiler to choose one over the other. The compiler reports an error and leaves it up to you to resolve the ambiguity. Provide a `getId` method in the `Employee` class and either implement your own ID scheme, or delegate to one of the conflicting methods, like this:

```
public class Employee implements Person, Identified {  
    public int getId() { return Identified.super.getId(); }  
    ...  
}
```



NOTE: The `super` keyword lets you call a supertype method. In this case, we need to specify which supertype we want. The syntax may seem a bit odd, but it is consistent with the syntax for invoking a superclass method that you will see in Chapter 4.

Now assume that the `Identified` interface does not provide a default implementation for `getId`:

```
interface Identified {  
    int getId();  
}
```

Can the `Employee` class inherit the default method from the `Person` interface? At first glance, this might seem reasonable. But how does the compiler know whether the `Person.getId` method actually does what `Identified.getId` is expected to do? After all, it might return the level of the person's Freudian id, not an ID number.

The Java designers decided in favor of safety and uniformity. It doesn't matter how two interfaces conflict; if at least one interface provides an implementation, the compiler reports an error, and it is up to the programmer to resolve the ambiguity.



NOTE: If neither interface provides a default for a shared method, then there is no conflict. An implementing class has two choices: implement the method, or leave it unimplemented and declare the class as abstract.



NOTE: If a class extends a superclass (see Chapter 4) and implements an interface, inheriting the same method from both, the rules are easier. In that case, only the superclass method matters, and any default method from the interface is simply ignored. This is actually a more common case than conflicting interfaces. See Chapter 4 for the details.

3.3 Examples of Interfaces

At first glance, interfaces don't seem to do very much. An interface is just a set of methods that a class promises to implement. To make the importance of interfaces more tangible, the following sections show you four examples of commonly used interfaces from the standard Java library.

3.3.1 The Comparable Interface

Suppose you want to sort an array of objects. A sorting algorithm repeatedly compares elements and rearranges them if they are out of order. Of course, the rules for doing the comparison are different for each class, and the sorting algorithm should just call a method supplied by the class. As long as all classes can agree on what that method is called, the sorting algorithm can do its job. That is where interfaces come in.

If a class wants to enable sorting for its objects, it should implement the `Comparable` interface. There is a technical point about this interface. We want to compare strings against strings, employees against employees, and so on. For that reason, the `Comparable` interface has a type parameter.

```
public interface Comparable<T> {  
    int compareTo(T other);  
}
```

For example, the `String` class implements `Comparable<String>` so that its `compareTo` method has the signature

```
int compareTo(String other)
```



NOTE: A type with a type parameter such as `Comparable` or `ArrayList` is a *generic* type. You will learn all about generic types in Chapter 6.

When calling `x.compareTo(y)`, the `compareTo` method returns an integer value to indicate whether `x` or `y` should come first. A positive return value (not necessarily 1) indicates that `x` should come after `y`. A negative integer (not necessarily -1) is returned when `x` should come before `y`. If `x` and `y` are considered equal, the returned value is 0.

Note that the return value can be any integer. That flexibility is useful because it allows you to return a difference of non-negative integers.

```
public class Employee implements Comparable<Employee> {  
    ...  
    public int compareTo(Employee other) {  
        return getId() - other.getId(); // Ok if IDs always ≥ 0  
    }  
}
```



CAUTION: Returning a difference of integers does not work if the integers can be negative. Then the difference can overflow for large operands of opposite sign. In that case, use the `Integer.compare` method that works correctly for all integers.

When comparing floating-point values, you cannot just return the difference. Instead, use the static `Double.compare` method. It does the right thing, even for $\pm\infty$ and NaN.

Here is how the `Employee` class can implement the `Comparable` interface, ordering employees by salary:

```
public class Employee implements Comparable<Employee> {  
    ...  
    public int compareTo(Employee other) {  
        return Double.compare(salary, other.salary);  
    }  
}
```



NOTE: It is perfectly legal for the `compare` method to access `other.salary`. In Java, a method can access private features of *any* object of its class.

The `String` class, as well as over a hundred other classes in the Java library, implements the `Comparable` interface. You can use the `Arrays.sort` method to sort an array of `Comparable` objects:

```
String[] friends = { "Peter", "Paul", "Mary" };  
Arrays.sort(friends); // friends is now ["Mary", "Paul", "Peter"]
```



NOTE: Strangely, the `Arrays.sort` method does not check at compile time whether the argument is an array of `Comparable` objects. Instead, it throws an exception if it encounters an element of a class that doesn't implement the `Comparable` interface.

3.3.2 The Comparator Interface

Now suppose we want to sort strings by increasing length, not in dictionary order. We can't have the `String` class implement the `compareTo` method in two ways—and at any rate, the `String` class isn't ours to modify.

To deal with this situation, there is a second version of the `Arrays.sort` method whose parameters are an array and a *comparator*—an instance of a class that implements the `Comparator` interface.

```
public interface Comparator<T> {  
    int compare(T first, T second);  
}
```

To compare strings by length, define a class that implements `Comparator<String>`:

```
class LengthComparator implements Comparator<String> {  
    public int compare(String first, String second) {  
        return first.length() - second.length();  
    }  
}
```

To actually do the comparison, you need to make an instance:

```
Comparator<String> comp = new LengthComparator();  
if (comp.compare(words[i], words[j]) > 0) ...
```

Contrast this call with `words[i].compareTo(words[j])`. The `compare` method is called on the comparator object, not the string itself.



NOTE: Even though the `LengthComparator` object has no state, you still need to make an instance of it. You need the instance to call the `compare` method—it is not a static method.

To sort an array, pass a `LengthComparator` object to the `Arrays.sort` method:

```
String[] friends = { "Peter", "Paul", "Mary" };  
Arrays.sort(friends, new LengthComparator());
```

Now the array is either ["Paul", "Mary", "Peter"] or ["Mary", "Paul", "Peter"].

You will see in Section 3.4.2, “Functional Interfaces,” on p. 109 how to use a `Comparator` much more easily, using a lambda expression.

3.3.3 The Runnable Interface

At a time when just about every processor has multiple cores, you want to keep those cores busy. You may want to run certain tasks in a separate thread, or give them to a thread pool for execution. To define the task, you implement the `Runnable` interface. This interface has just one method.

```
class HelloTask implements Runnable {  
    public void run() {  
        for (int i = 0; i < 1000; i++) {  
            System.out.println("Hello, World!");  
        }  
    }  
}
```

If you want to execute this task in a new thread, create the thread from the `Runnable` and start it.

```
Runnable task = new HelloTask();
Thread thread = new Thread(task);
thread.start();
```

Now the `run` method executes in a separate thread, and the current thread can proceed with other work.



NOTE: In Chapter 10, you will see other ways of executing a `Runnable`.



NOTE: There is also a `Callable<T>` interface for tasks that return a result of type `T`.

3.3.4 User Interface Callbacks

In a graphical user interface, you have to specify actions to be carried out when the user clicks a button, selects a menu option, drags a slider, and so on. These actions are often called *callbacks* because some code gets called back when a user action occurs.

In Java-based GUI libraries, interfaces are used for callbacks. For example, in JavaFX, the following interface is used for reporting events:

```
public interface EventHandler<T> {
    void handle(T event);
}
```

This too is a generic interface where `T` is the type of event that is being reported, such as an `ActionEvent` for a button click.

To specify the action, implement the interface:

```
class CancelAction implements EventHandler<ActionEvent> {
    public void handle(ActionEvent event) {
        System.out.println("Oh noes!");
    }
}
```

Then, make an object of that class and add it to the button:

```
Button cancelButton = new Button("Cancel");
cancelButton.setOnAction(new CancelAction());
```



NOTE: Since Oracle positions JavaFX as the successor to the Swing GUI toolkit, I use JavaFX in these examples. (Don't worry—you need not know any more about JavaFX than the couple of statements you just saw.) The details don't matter; in every user interface toolkit, be it Swing, JavaFX, or Android, you give a button some code that you want to run when the button is clicked.

Of course, this way of defining a button action is rather tedious. In other languages, you just give the button a function to execute, without going through the detour of making a class and instantiating it. The next section shows how you can do the same in Java.

3.4 Lambda Expressions

A “lambda expression” is a block of code that you can pass around so it can be executed later, once or multiple times. In the preceding sections, you have seen many situations where it is useful to specify such a block of code:

- To pass a comparison method to `Arrays.sort`
- To run a task in a separate thread
- To specify an action that should happen when a button is clicked

However, Java is an object-oriented language where (just about) everything is an object. There are no function types in Java. Instead, functions are expressed as objects, instances of classes that implement a particular interface. Lambda expressions give you a convenient syntax for creating such instances.

3.4.1 The Syntax of Lambda Expressions

Consider again the sorting example from Section 3.3.2, “The Comparator Interface,” on p. 104. We pass code that checks whether one string is shorter than another. We compute

```
first.length() - second.length()
```

What are `first` and `second`? They are both strings. Java is a strongly typed language, and we must specify that as well:

```
(String first, String second) -> first.length() - second.length()
```

You have just seen your first *lambda expression*. Such an expression is simply a block of code, together with the specification of any variables that must be passed to the code.

Why the name? Many years ago, before there were any computers, the logician Alonzo Church wanted to formalize what it means for a mathematical function to be effectively computable. (Curiously, there are functions that are known to exist, but nobody knows how to compute their values.) He used the Greek letter lambda (λ) to mark parameters, somewhat like

```
 $\lambda$ first.  $\lambda$ second. first.length() - second.length()
```



NOTE: Why the letter λ ? Did Church run out of letters of the alphabet? Actually, the venerable *Principia Mathematica* (see <http://plato.stanford.edu/entries/principia-mathematica>) used the $\wedge$ accent to denote function parameters, which inspired Church to use an uppercase lambda Λ . But in the end, he switched to the lowercase version. Ever since, an expression with parameter variables has been called a lambda expression.

If the body of a lambda expression carries out a computation that doesn't fit in a single expression, write it exactly like you would have written a method: enclosed in `{}` and with explicit return statements. For example,

```
(String first, String second) -> {
    int difference = first.length() - second.length();
    if (difference < 0) return -1;
    else if (difference > 0) return 1;
    else return 0;
}
```

If a lambda expression has no parameters, supply empty parentheses, just as with a parameterless method:

```
Runnable task = () -> { for (int i = 0; i < 1000; i++) doWork(); }
```

If the parameter types of a lambda expression can be inferred, you can omit them. For example,

```
Comparator<String> comp
    = (first, second) -> first.length() - second.length();
// Same as (String first, String second)
```

Here, the compiler can deduce that `first` and `second` must be strings because the lambda expression is assigned to a string comparator. (We will have a closer look at this assignment in the next section.)

If a method has a single parameter with inferred type, you can even omit the parentheses:

```
EventHandler<ActionEvent> listener = event ->
    System.out.println("Oh noes!");
// Instead of (event) -> or (ActionEvent event) ->
```

You never specify the result type of a lambda expression. However, the compiler infers it from the body and checks that it matches the expected type. For example, the expression

```
(String first, String second) -> first.length() - second.length()
```

can be used in a context where a result of type `int` is expected (or a compatible type such as `Integer`, `long`, or `double`).

3.4.2 Functional Interfaces

As you already saw, there are many interfaces in Java that express actions, such as `Runnable` or `Comparator`. Lambda expressions are compatible with these interfaces.

You can supply a lambda expression whenever an object of an interface with a *single abstract method* is expected. Such an interface is called a *functional interface*.

To demonstrate the conversion to a functional interface, consider the `Arrays.sort` method. Its second parameter requires an instance of `Comparator`, an interface with a single method. Simply supply a lambda:

```
Arrays.sort(words,  
    (first, second) -> first.length() - second.length());
```

Behind the scenes, the second parameter variable of the `Arrays.sort` method receives an object of some class that implements `Comparator<String>`. Invoking the `compare` method on that object executes the body of the lambda expression. The management of these objects and classes is completely implementation-dependent and highly optimized.

In most programming languages that support function literals, you can declare function types such as `(String, String) -> int`, declare variables of those types, put functions into those variables, and invoke them. In Java, there is *only one thing* you can do with a lambda expression: put it in a variable whose type is a functional interface, so that it is converted to an instance of that interface.



NOTE: You cannot assign a lambda expression to a variable of type `Object`, the common supertype of all classes in Java (see Chapter 4). `Object` is a class, not a functional interface.

The standard library provides a large number of functional interfaces (see Section 3.6.2, “Choosing a Functional Interface,” on p. 113). One of them is

```
public interface Predicate<T> {  
    boolean test(T t);  
    // Additional default and static methods  
}
```

The `ArrayList` class has a `removeIf` method whose parameter is a `Predicate`. It is specifically designed to pass a lambda expression. For example, the following statement removes all `null` values from an array list:

```
list.removeIf(e -> e == null);
```

3.5 Method and Constructor References

Sometimes, there is already a method that carries out exactly the action that you'd like to pass on to some other code. There is special syntax for a *method reference* that is even shorter than a lambda expression calling the method. A similar shortcut exists for constructors. You will see both in the following sections.

3.5.1 Method References

Suppose you want to sort strings regardless of letter case. You could call

```
Arrays.sort(strings, (x, y) -> x.compareToIgnoreCase(y));
```

Instead, you can pass this method expression:

```
Arrays.sort(strings, String::compareToIgnoreCase);
```

The expression `String::compareToIgnoreCase` is a *method reference* that is equivalent to the lambda expression `(x, y) -> x.compareToIgnoreCase(y)`.

Here is another example. The `Objects` class defines a method `isNull`. The call `Objects.isNull(x)` simply returns the value of `x == null`. It seems hardly worth having a method for this case, but it was designed to be passed as a method expression. The call

```
list.removeIf(Objects::isNull);
```

removes all `null` values from a list.

As another example, suppose you want to print all elements of a list. The `ArrayList` class has a method `forEach` that applies a function to each element. You could call

```
list.forEach(x -> System.out.println(x));
```

It would be nicer, however, if you could just pass the `println` method to the `forEach` method. Here is how to do that:

```
list.forEach(System.out::println);
```

As you can see from these examples, the `::` operator separates the method name from the name of a class or object. There are three variations:

1. `Class::instanceMethod`
2. `Class::staticMethod`
3. `object::instanceMethod`

In the first case, the first parameter becomes the receiver of the method, and any other parameters are passed to the method. For example, `String::compareToIgnoreCase` is the same as `(x, y) -> x.compareToIgnoreCase(y)`.

In the second case, all parameters are passed to the static method. The method expression `Objects::isNull` is equivalent to `x -> Objects.isNull(x)`.

In the third case, the method is invoked on the given object, and the parameters are passed to the instance method. Therefore, `System.out::println` is equivalent to `x -> System.out.println(x)`.



NOTE: When there are multiple overloaded methods with the same name, the compiler will try to find from the context which one you mean. For example, there are multiple versions of the `println` method. When passed to the `forEach` method of an `ArrayList<String>`, the `println(String)` method is picked.

You can capture the `this` parameter in a method reference. For example, `this::equals` is the same as `x -> this.equals(x)`.



NOTE: In an inner class, you can capture the `this` reference of an enclosing class as `EnclosingClass.this::method`. You can also capture `super`—see Chapter 4.

3.5.2 Constructor References

Constructor references are just like method references, except that the name of the method is `new`. For example, `Employee::new` is a reference to an `Employee` constructor. If the class has more than one constructor, then it depends on the context which constructor is chosen.

Here is an example for using such a constructor reference. Suppose you have a list of strings

```
List<String> names = ...;
```

You want a list of employees, one for each name. As you will see in Chapter 8, you can use streams to do this without a loop: Turn the list into a stream, and then call the `map` method. It applies a function and collects all results.

```
Stream<Employee> stream = names.stream().map(Employee::new);
```

Since `names.stream()` contains `String` objects, the compiler knows that `Employee::new` refers to the constructor `Employee(String)`.

You can form constructor references with array types. For example, `int[]::new` is a constructor reference with one parameter: the length of the array. It is equivalent to the lambda expression `n -> new int[n]`.

Array constructor references are useful to overcome a limitation of Java: It is not possible to construct an array of a generic type. (See Chapter 6 for details.) For that reason, methods such `Stream.toArray` return an `Object` array, not an array of the element type:

```
Object[] employees = stream.toArray();
```

But that is unsatisfactory. The user wants an array of employees, not objects. To solve this problem, another version of `toArray` accepts a constructor reference:

```
Employee[] buttons = stream.toArray(Employee[]::new);
```

The `toArray` method invokes this constructor to obtain an array of the correct type. Then it fills and returns the array.

3.6 Processing Lambda Expressions

Up to now, you have seen how to produce lambda expressions and pass them to a method that expects a functional interface. In the following sections, you will see how to write your own methods that can consume lambda expressions.

3.6.1 Implementing Deferred Execution

The point of using lambdas is *deferred execution*. After all, if you wanted to execute some code right now, you'd do that, without wrapping it inside a lambda. There are many reasons for executing code later, such as:

- Running the code in a separate thread
- Running the code multiple times
- Running the code at the right point in an algorithm (for example, the comparison operation in sorting)
- Running the code when something happens (a button was clicked, data has arrived, and so on)
- Running the code only when necessary

Let's look at a simple example. Suppose you want to repeat an action *n* times. The action and the count are passed to a `repeat` method:

```
repeat(10, () -> System.out.println("Hello, World!"));
```

To accept the lambda, we need to pick (or, in rare cases, provide) a functional interface. In this case, we can just use `Runnable`:

```
public static void repeat(int n, Runnable action) {  
    for (int i = 0; i < n; i++) action.run();  
}
```

Note that the body of the lambda expression is executed when `action.run()` is called.

Now let's make this example a bit more sophisticated. We want to tell the action in which iteration it occurs. For that, we need to pick a functional interface that has a method with an `int` parameter and a `void` return. Instead of rolling your own, I strongly recommend that you use one of the standard ones described in the next section. The standard interface for processing `int` values is

```
public interface IntConsumer {  
    void accept(int value);  
}
```

Here is the improved version of the `repeat` method:

```
public static void repeat(int n, IntConsumer action) {  
    for (int i = 0; i < n; i++) action.accept(i);  
}
```

And here is how you call it:

```
repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
```

3.6.2 Choosing a Functional Interface

In most functional programming languages, function types are *structural*. To specify a function that maps two strings to an integer, you use a type that looks something like `Function2<String, String, Integer>` or `(String, String) -> int`. In Java, you instead declare the intent of the function using a functional interface such as `Comparator<String>`. In the theory of programming languages this is called *nominal* typing.

Of course, there are many situations where you want to accept “any function” without particular semantics. There are a number of generic function types for that purpose (see Table 3–1), and it's a very good idea to use one of them when you can.

Table 3–1 Common Functional Interfaces

Functional Interface	Parameter types	Return type	Abstract method name	Description	Other methods
Runnable	none	void	run	Runs an action without arguments or return value	
Supplier<T>	none	T	get	Supplies a value of type T	
Consumer<T>	T	void	accept	Consumes a value of type T	andThen
BiConsumer<T, U>	T, U	void	accept	Consumes values of types T and U	andThen
Function<T, R>	T	R	apply	A function with argument of type T	compose, andThen, identity
BiFunction<T, U, R>	T, U	R	apply	A function with arguments of types T and U	andThen
UnaryOperator<T>	T	T	apply	A unary operator on the type T	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	A binary operator on the type T	andThen, maxBy, minBy
Predicate<T>	T	boolean	test	A boolean-valued function	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	A boolean-valued function with two arguments	and, or, negate

For example, suppose you write a method to process files that match a certain criterion. Should you use the descriptive `java.io.FileFilter` class or a `Predicate<File>`? I strongly recommend that you use the standard `Predicate<File>`. The only reason not to do so would be if you already have many useful methods producing `FileFilter` instances.



NOTE: Most of the standard functional interfaces have nonabstract methods for producing or combining functions. For example, `Predicate.isEqual(a)` is the same as `a::equals`, but it also works if `a` is `null`. There are default methods `and`, `or`, `negate` for combining predicates. For example, `Predicate.isEqual(a).or(Predicate.isEqual(b))` is the same as `x -> a.equals(x) || b.equals(x)`.

Table 3–2 lists the 34 available specializations for primitive types `int`, `long`, and `double`. It is a good idea to use these specializations to reduce autoboxing. For that reason, I used an `IntConsumer` instead of a `Consumer<Integer>` in the example of the preceding section.

Table 3–2 Functional Interfaces for Primitive Types

p, q is `int`, `long`, `double`; *P, Q* is `Int`, `Long`, `Double`

Functional Interface	Parameter types	Return type	Abstract method name
<code>BooleanSupplier</code>	none	<code>boolean</code>	<code>getAsBoolean</code>
<code>PSupplier</code>	none	<i>p</i>	<code>getAsP</code>
<code>PConsumer</code>	<i>p</i>	<code>void</code>	<code>accept</code>
<code>ObjPConsumer<T></code>	<i>T, p</i>	<code>void</code>	<code>accept</code>
<code>PFunction<T></code>	<i>p</i>	<i>T</i>	<code>apply</code>
<code>PToQFunction</code>	<i>p</i>	<i>q</i>	<code>applyAsQ</code>
<code>ToPFunction<T></code>	<i>T</i>	<i>p</i>	<code>applyAsP</code>
<code>ToPBifunction<T, U></code>	<i>T, U</i>	<i>p</i>	<code>applyAsP</code>
<code>PUnaryOperator</code>	<i>p</i>	<i>p</i>	<code>applyAsP</code>
<code>PBinaryOperator</code>	<i>p, p</i>	<i>p</i>	<code>applyAsP</code>
<code>PPredicate</code>	<i>p</i>	<code>boolean</code>	<code>test</code>

3.6.3 Implementing Your Own Functional Interfaces

Ever so often, you will be in a situation where none of the standard functional interfaces work for you. Then you need to roll your own.

Suppose you want to fill an image with color patterns, where the user supplies a function yielding the color for each pixel. There is no standard type for a mapping `(int, int) -> Color`. You could use `Bifunction<Integer, Integer, Color>`, but that involves autoboxing.

In this case, it makes sense to define a new interface

```
@FunctionalInterface
public interface PixelFunction {
    Color apply(int x, int y);
}
```



NOTE: You should tag functional interfaces with the `@FunctionalInterface` annotation. This has two advantages. First, the compiler checks that the annotated entity is an interface with a single abstract method. Second, the javadoc page includes a statement that your interface is a functional interface.

Now you are ready to implement a method:

```
BufferedImage createImage(int width, int height, PixelFunction f) {
    BufferedImage image = new BufferedImage(width, height,
        BufferedImage.TYPE_INT_RGB);

    for (int x = 0; x < width; x++)
        for (int y = 0; y < height; y++) {
            Color color = f.apply(x, y);
            image.setRGB(x, y, color.getRGB());
        }
    return image;
}
```

To call it, supply a lambda expression that yields a color value for two integers:

```
BufferedImage frenchFlag = createImage(150, 100,
    (x, y) -> x < 50 ? Color.BLUE : x < 100 ? Color.WHITE : Color.RED);
```

3.7 Lambda Expressions and Variable Scope

In the following sections, you will learn how variables work inside lambda expressions. This information is somewhat technical but essential for working with lambda expressions.

3.7.1 Scope of a Lambda Expression

The body of a lambda expression has *the same scope as a nested block*. The same rules for name conflicts and shadowing apply. It is illegal to declare a parameter or a local variable in the lambda that has the same name as a local variable.

```
int first = 0;
Comparator<String> comp = (first, second) -> first.length() - second.length();
// Error: Variable first already defined
```

Inside a method, you can't have two local variables with the same name, therefore you can't introduce such variables in a lambda expression either.

As another consequence of the "same scope" rule, the `this` keyword in a lambda expression denotes the `this` parameter of the method that creates the lambda. For example, consider

```
public class Application() {
    public void doWork() {
        Runnable runner = () -> { ...; System.out.println(this.toString()); ... };
        ...
    }
}
```

The expression `this.toString()` calls the `toString` method of the `Application` object, *not* the `Runnable` instance. There is nothing special about the use of `this` in a lambda expression. The scope of the lambda expression is nested inside the `doWork` method, and `this` has the same meaning anywhere in that method.

3.7.2 Accessing Variables from the Enclosing Scope

Often, you want to access variables from an enclosing method or class in a lambda expression. Consider this example:

```
public static void repeatMessage(String text, int count) {
    Runnable r = () -> {
        for (int i = 0; i < count; i++) {
            System.out.println(text);
        }
    };
    new Thread(r).start();
}
```

Note that the lambda expression accesses the parameter variables defined in the enclosing scope, not in the lambda expression itself.

Consider a call

```
repeatMessage("Hello", 1000); // Prints Hello 1000 times in a separate thread
```

Now look at the variables `count` and `text` inside the lambda expression. If you think about it, something nonobvious is going on here. The code of the lambda expression may run long after the call to `repeatMessage` has returned and the

parameter variables are gone. How do the text and count variables stay around when the lambda expression is ready to execute?

To understand what is happening, we need to refine our understanding of a lambda expression. A lambda expression has three ingredients:

1. A block of code
2. Parameters
3. Values for the *free* variables—that is, the variables that are not parameters and not defined inside the code

In our example, the lambda expression has two free variables, text and count. The data structure representing the lambda expression must store the values for these variables—in our case, "Hello" and 1000. We say that these values have been *captured* by the lambda expression. (It's an implementation detail how that is done. For example, one can translate a lambda expression into an object with a single method, so that the values of the free variables are copied into instance variables of that object.)



NOTE: The technical term for a block of code together with the values of free variables is a *closure*. In Java, lambda expressions are closures.

As you have seen, a lambda expression can capture the value of a variable in the enclosing scope. To ensure that the captured value is well defined, there is an important restriction. In a lambda expression, you can only reference variables whose value doesn't change. This is sometimes described by saying that lambda expressions capture values, not variables. For example, the following is a compile-time error:

```
for (int i = 0; i < n; i++) {  
    new Thread(() -> System.out.println(i)).start();  
    // Error—cannot capture i  
}
```

The lambda expression tries to capture *i*, but this is not legal because *i* changes. There is no single value to capture. The rule is that a lambda expression can only access local variables from an enclosing scope that are *effectively final*. An *effectively final* variable is never modified—it either is or could be declared as *final*.



NOTE: The same rule applies to variables captured by local inner classes (see Section 3.9, "Local Inner Classes," on p. 122). In the past, the rule was more draconian and required captured variables to actually be declared *final*. This is no longer the case.



NOTE: The variable of an enhanced for loop is effectively final since its scope is a single iteration. The following is perfectly legal:

```
for (String arg : args) {  
    new Thread(() -> System.out.println(arg)).start();  
    // OK to capture arg  
}
```

A new variable `arg` is created in each iteration and assigned the next value from the `args` array. In contrast, the scope of the variable `i` in the preceding example was the entire loop.

As a consequence of the “effectively final” rule, a lambda expression cannot mutate any captured variables. For example,

```
public static void repeatMessage(String text, int count, int threads) {  
    Runnable r = () -> {  
        while (count > 0) {  
            count--; // Error: Can't mutate captured variable  
            System.out.println(text);  
        }  
    };  
    for (int i = 0; i < threads; i++) new Thread(r).start();  
}
```

This is actually a good thing. As you will see in Chapter 10, if two threads update `count` at the same time, its value is undefined.



NOTE: Don't count on the compiler to catch all concurrent access errors. The prohibition against mutation only holds for local variables. If `count` is an instance variable or static variable of an enclosing class, then no error is reported even though the result is just as undefined.



CAUTION: One can circumvent the check for inappropriate mutations by using an array of length 1:

```
int[] counter = new int[1];  
button.setOnAction(event -> counter[0]++);
```

The `counter` variable is effectively final—it is never changed since it always refers to the same array, so you can access it in the lambda expression.

Of course, code like this is not threadsafe. Except possibly for a callback in a single-threaded UI, this is a terrible idea. You will see how to implement a threadsafe shared counter in Chapter 10.

3.8 Higher-Order Functions

In a functional programming language, functions are first-class citizens. Just like you can pass numbers to methods and have methods that produce numbers, you can have arguments and return values that are functions. Functions that process or return functions are called *higher-order functions*. This sounds abstract, but it is very useful in practice. Java is not quite a functional language because it uses functional interfaces, but the principle is the same. In the following sections, we will look at some examples and examine the higher-order functions in the `Comparator` interface.

3.8.1 Methods that Return Functions

Suppose sometimes we want to sort an array of strings in ascending order and other times in descending order. We can make a method that produces the correct comparator:

```
public static Comparator<String> compareInDirection(int direction) {  
    return (x, y) -> direction * x.compareTo(y);  
}
```

The call `compareInDirection(1)` yields an ascending comparator, and the call `compareInDirection(-1)` a descending comparator.

The result can be passed to another method (such as `Arrays.sort`) that expects such an interface.

```
Arrays.sort(friends, compareInDirection(-1));
```

In general, don't be shy to write methods that produce functions (or, technically, instances of classes that implement a functional interface). This is useful to generate custom functions that you pass to methods with functional interfaces.

3.8.2 Methods That Modify Functions

In the preceding section, you saw a method that yields an increasing or decreasing string comparator. We can generalize this idea by reversing any comparator:

```
public static Comparator<String> reverse(Comparator<String> comp) {  
    return (x, y) -> comp.compare(x, y);  
}
```

This method operates on functions. It receives a function and returns a modified function. To get case-insensitive descending order, use

```
reverse(String::compareToIgnoreCase)
```



NOTE: The `Comparator` interface has a default method `reversed` that produces the reverse of a given comparator in just this way.

3.8.3 Comparator Methods

The `Comparator` interface has a number of useful static methods that are higher-order functions generating comparators.

The `comparing` method takes a “key extractor” function that maps a type `T` to a comparable type (such as `String`). The function is applied to the objects to be compared, and the comparison is then made on the returned keys. For example, suppose you have an array of `Person` objects. Here is how you can sort them by name:

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

You can chain comparators with the `thenComparing` method for breaking ties. For example,

```
Arrays.sort(people, Comparator
    .comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

If two people have the same last name, then the second comparator is used.

There are a few variations of these methods. You can specify a comparator to be used for the keys that the `comparing` and `thenComparing` methods extract. For example, here we sort people by the length of their names:

```
Arrays.sort(people, Comparator.comparing(Person::getName,
    (s, t) -> s.length() - t.length()));
```

Moreover, both the `comparing` and `thenComparing` methods have variants that avoid boxing of `int`, `long`, or `double` values. An easier way of sorting by name length would be

```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

If your key function can return `null`, you will like the `nullsFirst` and `nullsLast` adapters. These static methods take an existing comparator and modify it so that it doesn’t throw an exception when encountering `null` values but ranks them as smaller or larger than regular values. For example, suppose `getMiddleName` returns a `null` when a person has no middle name. Then you can use `Comparator.comparing(Person::getMiddleName(), Comparator.nullsFirst(...))`.

The `nullsFirst` method needs a comparator—in this case, one that compares two strings. The `naturalOrder` method makes a comparator for any class implementing `Comparable`. Here is the complete call for sorting by potentially null middle names.

I use a static import of `java.util.Comparator.*` to make the expression more legible. Note that the type for `naturalOrder` is inferred.

```
Arrays.sort(people, comparing(Person::getMiddleName,
    nullsFirst(naturalOrder())));
```

The static `reverseOrder` method gives the reverse of the natural order.

3.9 Local Inner Classes

Long before there were lambda expressions, Java had a mechanism for concisely defining classes that implement an interface (functional or not). For functional interfaces, you should definitely use lambda expressions, but once in a while, you may want a concise form for an interface that isn't functional. You will also encounter the classic constructs in legacy code.

3.9.1 Local Classes

You can define a class inside a method. Such a class is called a *local class*. You would do this for classes that are just tactical. This occurs often when a class implements an interface and the caller of the method only cares about the interface, not the class.

For example, consider a method

```
public static IntSequence randomInts(int low, int high)
```

that generates an infinite sequence of random integers with the given bounds.

Since `IntSequence` is an interface, the method must return an object of some class implementing that interface. The caller doesn't care about the class, so it can be declared inside the method:

```
private static Random generator = new Random();

public static IntSequence randomInts(int low, int high) {
    class RandomSequence implements IntSequence {
        public int next() { return low + generator.nextInt(high - low + 1); }
        public boolean hasNext() { return true; }
    }

    return new RandomSequence();
}
```



NOTE: A local class is not declared as `public` or `private` since it is never accessible outside the method.

There are two advantages of making a class local. First, its name is hidden in the scope of the method. Second, the methods of the class can access variables from the enclosing scope, just like the variables of a lambda expression.

In our example, the next method captures three variables: `low`, `high`, and `generator`. If you turned `RandomInt` into a nested class, you would have to provide an explicit constructor that receives these values and stores them in instance variables (see Exercise 15).

3.9.2 Anonymous Classes

In the example of the preceding section, the name `RandomSequence` was used exactly once: to construct the return value. In this case, you can make the class *anonymous*:

```
public static IntSequence randomInts(int low, int high) {
    return new IntSequence() {
        public int next() { return low + generator.nextInt(high - low + 1); }
        public boolean hasNext() { return true; }
    }
}
```

The expression

```
new Interface() { methods }
```

means: Define a class implementing the interface that has the given methods, and construct one object of that class.



NOTE: As always, the `()` in the `new` expression indicate the construction arguments. A default constructor of the anonymous class is invoked.

Before Java had lambda expressions, anonymous inner classes were the most concise syntax available for providing runnables, comparators, and other functional objects. You will often see them in legacy code.

Nowadays, they are only necessary when you need to provide two or more methods, as in the preceding example. If the `IntSequence` interface has a default `hasNext` method, as in Exercise 15, you can simply use a lambda expression:

```
public static IntSequence randomInts(int low, int high) {
    return () -> low + generator.nextInt(high - low + 1);
}
```

Exercises

1. Provide an interface `Measurable` with a method `double getMeasure()` that measures an object in some way. Make `Employee` implement `Measurable`. Provide a method `double average(Measurable[] objects)` that computes the average measure. Use it to compute the average salary of an array of employees.
2. Continue with the preceding exercise and provide a method `Measurable largest(Measurable[] objects)`. Use it to find the name of the employee with the largest salary. Why do you need a cast?
3. What are all the supertypes of `String`? Of `Scanner`? Of `ImageOutputStream`? Note that each type is its own supertype. A class or interface without declared supertype has supertype `Object`.
4. Implement a static method of the `IntSequence` class that yields a sequence with the arguments. For example, `IntSequence.of(3, 1, 4, 1, 5, 9)` yields a sequence with six values. Extra credit if you return an instance of an anonymous inner class.
5. Implement a static constant method of the `IntSequence` class that yields an infinite constant sequence. For example, `IntSequence.constant(1)` yields values 1 1 1 . . . , ad infinitum. Extra credit if you do this with a lambda expression.
6. In this exercise, you will try out what happens when a method is added to an interface. In Java 7, implement a class `DigitSequence` that implements `Iterator<Integer>`, not `IntSequence`. Provide methods `hasNext`, `next`, and a do-nothing `remove`. Write a program that prints the elements of an instance. In Java 8, the `Iterator` class gained another method, `forEachRemaining`. Does your code still compile when you switch to Java 8? If you put your Java 7 class in a JAR file and don't recompile, does it work in Java 8? What if you call the `forEachRemaining` method? Also, the `remove` method has become a default method in Java 8, throwing an `UnsupportedOperationException`. What happens when `remove` is called on an instance of your class?
7. Implement the method `void luckySort(ArrayList<String> strings, Comparator<String> comp)` that keeps calling `Collections.shuffle` on the array list until the elements are in increasing order, as determined by the comparator.
8. Implement a class `Greeter` that implements `Runnable` and whose `run` method prints `n` copies of "Hello, " + `target`, where `n` and `target` are set in the constructor. Construct two instances with different messages and execute them concurrently in two threads.
9. Implement methods

```
public static void runTogether(Runnable... tasks)
public static void runInOrder(Runnable... tasks)
```

The first method should run each task in a separate thread and then return. The second method should run all methods in the current thread and return when the last one has completed.

10. Using the `listFiles(FileFilter)` and `isDirectory` methods of the `java.io.File` class, write a method that returns all subdirectories of a given directory. Use a lambda expression instead of a `FileFilter` object. Repeat with a method expression and an anonymous inner class.
11. Using the `list(FilenameFilter)` method of the `java.io.File` class, write a method that returns all files in a given directory with a given extension. Use a lambda expression, not a `FilenameFilter`. Which variable from the enclosing scope does it capture?
12. Given an array of `File` objects, sort it so that directories come before files, and within each group, elements are sorted by path name. Use a lambda expression to specify the `Comparator`.
13. Write a method that takes an array of `Runnable` instances and returns a `Runnable` whose `run` method executes them in order. Return a lambda expression.
14. Write a call to `Arrays.sort` that sorts employees by salary, breaking ties by name. Use `Comparator.thenComparing`. Then do this in reverse order.
15. Implement the `RandomSequence` in Section 3.9.1, “Local Classes,” on p. 122 as a nested class, outside the `randomInts` method.

Index

Symbols and Numbers

- (minus sign)
 - flag (for output), 30
 - in dates, 390
 - in regular expressions, 294
 - operator, 13–14
- - in shell scripts, 439
 - operator, 13, 15
- >, in lambda expressions, 107, 110–111
- ∞, in string templates, 409
- _ (underscore)
 - in number literals, 8
 - in variable names, 11, 61
- , (comma)
 - flag (for output), 30
 - in numbers, 398, 403, 408
- ; (semicolon)
 - in Java vs. JavaScript, 426
 - path separator (Windows), 75, 240
- : (colon)
 - in assertions, 185–186
 - in dates, 390
 - in switch statement, 31
 - path separator (Unix), 75, 240
 - :: operator, 110–111, 137
- ! (exclamation sign)
 - comments, in property files, 239
 - operator, 13, 18
- != operator, 13, 18
 - for wrapper classes, 40
- ? (quotation mark)
 - in regular expressions, 293–294, 296
 - replacement character, 281, 413
 - wildcard, for types, 204–208, 219
- ? : operator, 13, 18
- / (slash)
 - file separator (Unix), 239, 284
 - in javac path segments, 4
 - operator, 13–14
 - root component, 284
- //, /*...*/ comments, 3
- /**...*/ comments, 84–85
- /= operator, 13
- . (period)
 - in method calls, 5
 - in numbers, 398, 403, 408
 - in package names, 4, 73
 - in regular expressions, 293–294, 301
 - operator, 13

- ..., parent directory, 285
- ... (ellipsis), for varargs, 48
- `...` (back quotes), in shell scripts, 438–439
- ^ (caret)
 - for function parameters, 108
 - in regular expressions, 293–297, 300
 - operator, 13, 18
- = operator, 13
- ~ operator, 13, 18
- '...' (single quotes)
 - for character literals, 9–10
 - in JavaScript, 426
 - in string templates, 409
- "..." (double quotes)
 - for strings, 6
 - in javadoc hyperlinks, 88
 - in shell scripts, 438
- "" (empty string), 22–23, 139
- ((left parenthesis), in formatted output, 30
- () (empty parentheses), for anonymous classes, 123
- (...) (parentheses)
 - for anonymous functions (JavaScript), 435
 - for casts, 17, 97
 - in regular expressions, 293–296, 298–299
 - operator, 13
- [] (empty square brackets), for arrays, 37–38
- [...] (square brackets)
 - for arrays, 37, 44
 - in JavaScript, 430, 433–434
 - in regular expressions, 293–295
 - operator, 13
- {...} (curly braces)
 - in annotation elements, 357
 - in lambda expressions, 108
 - in regular expressions, 293–296, 300
 - in string templates, 408
 - with arrays, 38
- {{...}}, double brace initialization, 136
- @ (at), in javadoc comments, 85
- \$ (dollar sign)
 - currency symbol, 408
 - flag (for output), 30
 - in JavaScript function calls, 432, 437
 - in regular expressions, 293–294, 297, 300
 - in variable names, 11
- \${...}, in shell scripts, 438–440
- € currency symbol, 403, 408
- * (asterisk)
 - for annotation processors, 372
 - in documentation comments, 86
 - in regular expressions, 293–296, 299
 - operator, 13–14
 - wildcard:
 - in class path, 75
 - in imported classes, 77–78
- *= operator, 13
- \ (backslash)
 - character literal, 10
 - file separator (Windows), 239, 284
 - in regular expressions, 293–294, 300
- & (ampersand), operator, 13, 18–19
- && (double ampersand)
 - in regular expressions, 295
 - operator, 13, 18
- &= operator, 13
- # (number sign)
 - comments, in property files, 239
 - flag (for output), 30
 - in javadoc hyperlinks, 87
 - in string templates, 409
- #!, in shell scripts, 440
- % (percent sign)
 - conversion character, 28–29
 - operator, 13–14
- %% pattern variable, 193
- %= operator, 13
- + (plus sign)
 - flag (for output), 30
 - in regular expressions, 293–296
 - operator, 13–14
 - for strings, 20–21, 23, 139
- ++ operator, 13, 15
- += operator, 13
- < (left angle bracket)
 - flag (for output), 30
 - in shell syntax, 28
 - in string templates, 409
 - operator, 18, 432

- << operator, 13, 18–19
- <= operator, 13
- <= operator, 13, 18
- <%...%>, <%=...%> delimiters (JSP), 440
- ≤, in string templates, 409
- ◊ (diamond syntax)
 - for array lists, 39
 - for constructors of generic classes, 201
- <...> (angle brackets)
 - for type parameters, 103, 200
 - in javadoc hyperlinks, 88
 - in regular expressions, 296
- =, -= operators, 13–14
- == operator, 13, 18, 141
 - for class objects, 153
 - for enumerations, 147
 - for strings, 22
 - for wrapper classes, 40
- > (right angle bracket)
 - in shell syntax, 28
 - operator, 18
- >=, >>, >>> operators, 13, 18
- >>=, >>>= operators, 13
- | (vertical bar)
 - in regular expressions, 293–295
 - in string templates, 409
 - operator, 13, 18–19
- |= operator, 13
- || operator, 13, 18
- 0 (zero)
 - as default value, 65, 68
 - flag (for output), 30
 - formatting symbol (date/time), 393
 - prefix, for octal literals, 8
- \0, in regular expressions, 294
- 0b prefix, 8
- 0x prefix, 8
 - in formatted output, 30
- 0xFEFF byte order mark, 277

A

- a formatting symbol (date/time), 392
- a, A conversion characters, 29
- \a, \A, in regular expressions, 294, 297
- abstract classes, 133–134
- abstract methods, of an interface, 109

- abstract modifier, 97, 133–134
- AbstractCollection class, 100
- AbstractMethodError, 101
- AbstractProcessor class, 372
- accept methods (Consumer, XXXConsumer), 114–115
- acceptEither method (CompletableFuture), 344–345
- AccessibleObject class, 163
 - setAccessible method, 162–163
- accessors, 56
- accumulate method (LongAccumulator), 331
- accumulateAndGet method (AtomicXXX), 330
- accumulator functions, 265
- add method
 - of ArrayDeque, 242
 - of ArrayList, 39, 56
 - of BlockingQueue, 327
 - of Collection, 228
 - of List, 229
 - of ListIterator, 233
 - of LongAdder, 330
- addAll method
 - of Collection, 206, 228
 - of Collections, 231
- addExact method (Math), 16
- addHandler method (Logger), 191
- addition, 14
 - identity for, 265
- addSuppressed method (IOException), 181
- allMatch method (Stream), 256
- allOf method
 - of CompletableFuture, 344–345
 - of EnumSet, 241
- and, andNot methods (BitSet), 241
- and, andThen methods (functional interfaces), 114
- Android, 342
- AnnotatedConstruct interface, 373
- AnnotatedElement interface, 369–371
- annotation interfaces, 361–364
- annotation processors, 372
- annotations
 - accessing, 362
 - and modifiers, 360
 - container, 368, 370
 - declaration, 358–359

- annotations (*cont.*)
 - documented, 367
 - generating source code with, 373–376
 - inherited, 367, 370
 - key/value pairs in. *See* elements
 - meta, 362–368
 - multiple, 358
 - processing:
 - at runtime, 368–371
 - source-level, 371–376
 - repeatable, 358, 368, 370
 - standard, 364–368
 - type use, 359–360
- anonymous classes, 123
- anyMatch method (Stream), 256
- anyOf method (CompletableFuture), 344–345
- Applet class, 155
- applications. *See* programs
- apply, applyAsXXX methods (functional interfaces), 114–115
- applyToEither method (CompletableFuture), 344–345
- \$ARG, in shell scripts, 440
- arguments array (jjs), 440
- arithmetic operations, 13–19
- Array class, 165–167
- array list variables, 39
- array lists, 39–40
 - accessing elements in, 40
 - adding elements to, 39
 - anonymous, 136
 - checking for nulls, 207
 - constructing, 39
 - converting between, 204
 - copying, 42
 - filling, 42
 - instantiating with type variables, 214
 - removing elements from, 40
 - size of, 40
 - sorting, 43
 - visiting all elements of, 41
- array variables
 - assigning values to, 38
 - copying, 41
 - declaring, 37–38
 - initializing, 37
- ArrayBlockingQueue class, 328
- ArrayDeque class, 242
- ArrayIndexOutOfBoundsException, 38
- ArrayList class, 39–40, 230
 - add method, 39, 56
 - clone method, 146–147
 - forEach method, 110
 - get, set methods, 40
 - remove method, 40
 - removeIf method, 110
 - size method, 40
- arrays, 37–39
 - accessing nonexistent elements in, 38
 - allocating, 214
 - annotating, 359
 - casting, 166
 - checking, 165
 - comparing, 141
 - computing values of, 324
 - constructing, 37–38
 - constructor references with, 112
 - converting:
 - to a reference of type Object, 138
 - to/from streams, 259, 268, 324
 - copying, 42
 - covariant, 203
 - filling, 38, 42
 - generating Class objects for, 153
 - growing, 165–167
 - hash codes of, 144
 - in JavaScript, 433–434
 - length of, 38–39, 119
 - multidimensional, 44–46, 140
 - of bytes, 274–275
 - of generic types, 112, 215
 - of objects, 38, 324
 - of primitive types, 324
 - of strings, 299
 - passing into methods, 47
 - printing, 43, 46, 140
 - serializable, 301
 - sorting, 43, 103–104, 324
 - superclass assignment in, 132
 - using class literals with, 152
 - visiting all elements of, 41

- Arrays class
 - asList method, 42, 244
 - copyOf method, 42
 - deepToString method, 140
 - equals method, 141
 - fill method, 42
 - hashCode method, 144
 - parallelXXX methods, 43, 324
 - sort method, 43, 104–105, 109–110
 - stream method, 252, 266
 - toString method, 43, 140
 - ArrayStoreException, 132, 203, 215
 - ASCII, 25, 276, 300
 - for property files, 412
 - for source files, 413
 - asList method (Arrays), 42, 244
 - ASM tool, 376
 - assert statement, 185–186
 - AssertionError, 186
 - assertions, 185–187
 - checking, 359
 - enabling/disabling, 186–187
 - assignment operators, 14
 - associative operations, 265
 - asSubclass method (Class), 219
 - asynchronous computations, 341–344
 - AsyncTask class (Android), 342
 - atomic methods, 326
 - atomic operations, 319, 321, 329–331
 - and performance, 330
 - AtomicXXX classes, 329–330
 - atZone method (LocalDateTime), 387
 - @author tag (javadoc), 85, 89
 - autoboxing, 40, 115
 - AutoCloseable interface, 180, 202
 - availableCharsets method (Charset), 278
 - availableProcessors method (Runtime), 313
 - average method (XXXStream), 267
- B**
- b, B conversion characters, 29
 - \b (backspace), 10
 - \b, \B, in regular expressions, 297
 - bash scripts, 438
 - bash scripts (Unix), 437
 - BasicFileAttributes class, 289
 - batch files (Windows), 437–438
 - BeanInfo class, 165
 - between method (Duration), 381
 - BiConsumer interface, 114
 - BiFunction interface, 114–115
 - BigDecimal, BigInteger classes, 19
 - big-endian format, 277, 282–283
 - binary data, reading/writing, 282
 - binary numbers, 8–9
 - binary trees, 234
 - BinaryOperator interface, 114
 - binarySearch method (Collections), 231
 - Bindings interface, 425
 - BiPredicate interface, 114
 - BitSet class, 240–241
 - collecting streams into, 266
 - methods of, 240–241
 - bitwise operators, 18–19
 - block statement, labeled, 35
 - blocking queues, 326–328
 - BlockingQueue interface, 327–328
 - Boolean class, 40
 - boolean type, 10
 - default value of, 65, 68
 - formatting for output, 29
 - reading/writing, 282
 - streams of, 266
 - BooleanSupplier interface, 115
 - bootstrap class loader, 156
 - boxed method (XXXStream), 267
 - branches, 30–32
 - break statement, 31–32, 34–35
 - labeled, 35
 - bridge methods, 210–211
 - clashes of, 217
 - BufferedReader class, 280
 - bulk operations, 326
 - Byte class, 40
 - MIN_VALUE, MAX_VALUE constants, 7
 - toUnsignedInt method, 8
 - byte codes, 4
 - writing to memory, 422–423
 - byte order mark, 277
 - byte type, 7–8, 275
 - streams of, 266
 - type conversions of, 17

- ByteArrayClass class, 422
- ByteArrayClassLoader class, 423
- ByteArrayInputStream class, 274
- ByteArrayOutputStream class, 274–275
- ByteBuffer class, 283
- bytes
 - arrays of, 274–275
 - converting to strings, 278
 - reading, 275
 - writing, 276
- C**
- c, C conversion characters, 29
- C:\ root component, 284
- C/C++ programming languages
 - #include directive in, 78
 - allocating memory in, 321
 - integer types in, 7
 - pointers in, 57
- C# programming language, type
 - parameters in, 207
- \c, in regular expressions, 294
- caching, 318
- calculators, 149–150
- Calendar class, 379
 - weekends in, 384
- calendars, 54
- call method (CompilationTask), 421
- call by reference, 63
- Callable interface, 106
 - call method, 314
 - extending, 421
- callbacks, 106–107
- camel case, 11
- cancel method (Future), 315
- cancellation requests, 338
- cardinality method (BitSet), 241
- carriage return, 10
- case label, 31–32
- cast method (Class), 219
- cast insertion, 209–210
- casts, 17, 97–98, 132
 - and generic types, 212
 - annotating, 360
- catch statement, 178–179
 - annotating parameters of, 358
 - in JavaScript, 437
 - in try-with-resources, 181
 - no type variables in, 217
- ceiling method (NavigableSet), 235
- Channel interface, 98
- channels, 283
- char type, 9–10
 - streams of, 266
 - type conversions of, 17
- Character class, 40
- character classes, 294
- character encodings, 276–279
 - detecting, 278
 - in PrintWriter constructor, 281
 - localizing, 413
 - partial, 277, 281
 - platform, 278, 413
- character literals, 9–10
- characters, 274
 - combined, 407
 - formatting for output, 29
 - normalized, 408
 - reading/writing, 282
- charAt method (String), 26
- CharSequence interface, 24
 - chars, codePoints methods, 266
 - splitting by regular expressions, 252
- Charset class
 - availableCharsets method, 278
 - defaultCharset method, 278, 413
 - displayName method, 413
 - forName method, 278
- checked exceptions, 176–178
 - and generic types, 218
 - combining in a superclass, 177
 - declaring, 177–178
 - documenting, 178
 - in lambda expressions, 178
 - not allowed in a method, 183
 - rethrowing, 182
- checked views, 213, 245
- checkedXXX methods (Collections), 232, 245
- Checker Framework, 359
- childrenNames method (Preferences), 415
- choice indicator, in string templates, 409
- Church, Alonzo, 108, 382

- Class class, 152–155, 220
 - asSubclass method, 219
 - cast method, 219
 - comparing objects of, 153
 - forName method, 152–153, 157–158, 176, 184, 423
 - generic, 219
 - getCanonicalName method, 153
 - getClassLoader method, 154, 156
 - getComponentType method, 154, 165
 - getConstructor(s) methods, 154, 161, 163, 219
 - getDeclaredConstructor(s) methods, 154, 161, 219
 - getDeclaredField(s) methods, 154
 - getDeclaredMethod(s) methods, 154, 162
 - getDeclaringClass method, 154
 - getEnclosingXXX methods, 154
 - getEnumConstants method, 219
 - getField(s) methods, 154, 161
 - getInterfaces method, 153
 - getMethod(s) methods, 154, 161–162
 - getModifiers method, 153
 - getName method, 152–153
 - getPackage method, 153
 - getResource method, 155, 410
 - getResourceAsStream method, 154–155
 - getSimpleName method, 153
 - getSuperClass method, 153, 219
 - getTypeName method, 153
 - getTypeParameters method, 220
 - isXXX methods, 153–154, 165
 - newInstance method, 154, 163, 219
 - toString, toGenericString methods, 153
- class comments, 85–86
- class declarations
 - annotations in, 358, 367
 - initialization blocks in, 66–67
- class files, 4, 155
 - compressing, 74
 - paths of, 73
 - processing annotations in, 376
- class literals, 152
 - no annotations for, 360
 - no type variables in, 213
- class loaders, 155–157
- class objects, 152
 - comparing, 153
- class path, 74–76, 156
- .class suffix, 152–153
- ClassCastException, 98, 212
- classes, 2, 54
 - abstract, 97, 102, 133–134
 - adding functionality to, 71
 - adding to packages, 77
 - anonymous, 123
 - companion, 100
 - compiling on the fly, 422
 - constructing objects of, 10
 - deprecated, 87
 - evolving, 306
 - extending, 128–137
 - in JavaScript, 435–437
 - fields of, 127
 - final, 133
 - generic, 39
 - immutable, 24, 322
 - implementing, 58–63, 145
 - importing, 77–78
 - inner, 81–83
 - instances of, 5, 59, 72
 - loading, 161
 - local, 122–123
 - members of, 127, 160–161
 - names of, 11, 72, 152
 - nested, 79–84, 359
 - nested enumerations in, 151
 - not known at compile time, 152, 167
 - protected, 134–135
 - public, 76
 - static initialization of, 157
 - super- and sub-, 128–129
 - system, 186
 - testing, 76
 - utility, 76, 157
 - wrapper, 40
- classes win rule, 144
- classifier functions, 262
- ClassLoader class, 77
 - extending, 423
 - findClass, loadClass methods, 157
 - setXXXAssertionStatus methods, 187

- classloader inversion, 158
- ClassNotFoundException, 176
- CLASSPATH environment variable, 75
- clear method
 - of BitSet, 240
 - of Collection, 228
 - of Map, 237
- clone method
 - of ArrayList, 146–147
 - of Enum, 148
 - of Message, 146–147
 - of Object, 135, 138, 144–147, 163
 - protected, 144
- Cloneable interface, 145
- CloneNotSupportedException, 146–148
- cloning, 144–147
- close method (PrintWriter), 179–180
- Closeable interface, 98
 - close method, 180
- closures, 118
- code element (HTML), in documentation
 - comments, 85
- code generator tools, 365–366
- code points, 26, 276
- code units, 9, 26, 266
 - in regular expressions, 294
- codePoints method (CharSequence), 266
- codePoints, codePointXXX methods (String), 26
- Collator class, 23
 - getInstance, setDecomposition, setStrength methods, 407
- collect method (Stream), 259–260, 266
- Collection interface, 100, 228
 - add method, 228
 - addAll method, 206, 228
 - clear method, 228
 - contains, containsAll, isEmpty methods, 229
 - iterator, spliterator methods, 229
 - parallelStream method, 229, 250–251, 267, 323
 - remove, removeXXX, retainAll methods, 228
 - size method, 228
 - stream method, 229, 250–251
 - toArray method, 229
- collections, 227–246
 - generic, 245
 - iterating over elements of, 250–251
 - processing, 230
 - serializable, 301
 - threadsafe, 328
 - unmodifiable views of, 245
 - vs. streams, 251
- Collections class, 100, 230–231
 - addAll method, 231
 - binarySearch method, 231
 - checkedXXX, emptyXXX methods, 232, 245
 - copy method, 231
 - disjoint method, 231
 - fill method, 42, 231
 - frequency method, 231
 - indexOfSubList, lastIndexOfSubList methods, 231
 - nCopies method, 230–231
 - replaceAll method, 231
 - reverse, shuffle method, 43, 232
 - rotate method, 232
 - singleton, singletonXXX methods, 232, 245
 - sort method, 43, 206–207, 231
 - static methods, 244
 - swap method, 231
 - synchronizedXXX, unmodifiableXXX methods, 232
- Collector interface, 259
- Collectors class, 79
 - counting method, 263
 - groupingBy method, 262–264
 - groupingByConcurrent method, 262, 269
 - joining method, 259–260
 - mapping method, 263–264
 - maxBy, minBy methods, 263
 - partitioningBy method, 262, 264
 - reducing method, 264
 - summarizingXXX methods, 260, 264
 - summingXXX methods, 263
 - toCollection method, 259
 - toConcurrentMap method, 261
 - toList method, 259
 - toMap method, 260–261
 - toSet method, 259, 263
- com global object (JavaScript), 431
- command-line arguments, 43–44
- comments, 3
 - documentation, 84–89

- companion classes, 100
- Comparable interface, 103–104, 148, 207, 234
 - compareTo method, 103
 - streams of, 254
 - with priority queues, 243
- Comparator interface, 79, 104–105, 120–122, 234
 - comparing, comparingXXX, thenComparing methods, 121
 - naturalOrder method, 121–122
 - nullsFirst, nullsLast methods, 121
 - reversed method, 121
 - reverseOrder method, 122
 - streams of, 254–255
 - with priority queues, 243
- compare method (Integer, Double), 104
- compareTo method
 - of Enum, 148
 - of Instant, 381
 - of String, 22–23, 103, 406
- compareToIgnoreCase method (String), 110
- compareUnsigned method (Integer, Long), 16
- compatibility, drawbacks of, 208
- Compilable interface, 428
- compilation, 4
- CompilationTask interface, 420
 - call method, 421
- compile method (Pattern), 298, 300
- compiler, 420
- compile-time errors, 11
- CompletableFuture class, 342–344
 - acceptEither, applyToEither methods, 344–345
 - allOf, anyOf methods, 344–345
 - get method, 344
 - handle method, 344
 - runAfterXXX methods, 344–345
 - thenAccept, thenAcceptBoth, thenCombine, thenRun, whenComplete methods, 344
 - thenApply, thenApplyAsync, thenCompose methods, 343–344
- CompletionStage interface, 344
- compose method (functional interfaces), 114
- compute, computeIfXXX methods (Map), 236
- concat method (Stream), 254
- concatenation, 20–21
 - objects with strings, 139
- concurrent access errors, 119
- concurrent programming, 311–348
 - for scripts, 425
 - strategies for, 321
- ConcurrentHashMap class, 325–326
 - compute method, 325–326
 - computeIfXXX, forEachXXX, merge, putIfAbsent, reduceXXX, searchXXX methods, 326
 - keySet method, 329
 - newKeySet method, 328
 - no null values in, 238
- ConcurrentModificationException, 233, 325
- ConcurrentSkipListXXX classes, 328
- conditional operator, 18
- configuration files, 413–415
 - editing, 190–191
 - locating, 155
 - resolving paths for, 285
- confinement, 321
- connect method (URLConnection), 292
- Console class, 27
- console, displaying fonts on, 413
- ConsoleHandler class, 191, 194
- constants, 12–13, 99
 - names of, 12
 - static, 69–70
 - using in another class, 12
- Constructor interface, 160–161
 - getModifiers, getName methods, 160
 - newInstance method, 163–164
- constructor references, 111–112
 - annotating, 360
- constructors, 63–68
 - annotating, 358–359
 - executing, 64
 - for subclasses, 131
 - implementing, 63–64
 - in abstract classes, 134
 - invoking another constructor from, 65
 - no-argument, 131, 163
 - overloading, 64–65
 - public, 64, 161
 - this references in, 322–323
 - with no arguments, 68

- Consumer interface, 114
 - contains method (String), 24
 - contains, containsAll methods (Collection), 229
 - containsXXX methods (Map), 237
 - Content-Type header, 278
 - context class loaders, 157–159
 - continue statement, 34–35
 - labeled, 35
 - control flow, 30–37
 - conversion characters, 29
 - cooperative cancellation, 338
 - copy method
 - of Collections, 231
 - of Files, 275–276, 287–288, 291
 - copyOf method (Arrays), 42
 - CopyOnWriteArrayXXX classes, 328
 - count method (Stream), 251, 255
 - CountdownLatch class, 337
 - counters, de/incrementing, 181
 - counting method (Collectors), 263
 - country codes, 262, 399–400
 - covariance, 203
 - createBindings method (ScriptEngine), 425
 - createInstance method (Util), 158
 - createTempXXX methods (Files), 287
 - createXXX methods (Files), 286
 - critical sections, 332, 338
 - Crockford, Douglas, 427
 - currencies, 403–404
 - formatting, 408
 - Currency class, 404
 - currency indicator, in string templates, 409
 - CyclicBarrier class, 337
- D**
- d
 - conversion character, 29
 - formatting symbol (date/time), 392
 - D suffix, 9
 - \d, \D, in regular expressions, 295
 - daemon threads, 341
 - databases, 355
 - annotating access to, 366
 - DataInput/Output interfaces, 281–282
 - read/writeXXX methods, 282–283, 304
 - DataXXXStream classes, 282
 - Date class, 379, 393–394
 - formatting values of, 409
 - date indicator, in string templates, 409
 - DateFormat class, 405
 - dates
 - computing, 385–386
 - formatting, 390–393, 398, 404–406
 - local, 382–384
 - nonexistent, 384, 388, 406
 - parsing, 393
 - DateTimeFormat class, 404–406
 - DateTimeFormatter class, 390–393
 - and legacy classes, 394
 - format method, 390, 405
 - ofLocalizedXXX methods, 391, 405
 - ofPattern method, 392
 - parse method, 393
 - toFormat method, 392
 - withLocale method, 392, 405
 - DateTimeParseException, 405
 - daylight savings time, 387–390
 - DayOfWeek enumeration, 55, 383–384, 389
 - getDisplayName method, 392, 406
 - dayOfWeekInMonth method (TemporalAdjusters), 385
 - deadlocks, 321, 332, 336, 338
 - debugging
 - messages for, 175
 - overriding methods for, 133
 - primary arrays for, 43
 - streams, 255
 - using anonymous subclasses for, 135–136
 - with assertions, 185
 - DecimalFormat class, 72
 - number format patterns of, 409
 - declaration-site variance, 207
 - decomposition (for classes), 46–48
 - decomposition modes (for characters), 407
 - decrement operator, 15
 - decrementExact method (Math), 16
 - deep copies, 145
 - deepToString method (Arrays), 140
 - default label (switch statement), 31–32, 151
 - default methods, 100–102
 - in interfaces, 144

- resolving conflicts of, 101–102, 136–137
 - default modifier, 100, 363
 - defaultCharset method (Charset), 278, 413
 - defaultXXXObject methods (ObjectXXXStream), 304
 - defensive programming, 185
 - deferred execution, 112–113
 - delete, deleteIfExists methods (Files), 288
 - delimiters, 280
 - @Deprecated annotation, 87, 364–365
 - @deprecated tag (javadoc), 87, 365
 - Deque interface, 230, 242
 - destroy, destroyForcibly methods (Process), 348
 - DiagnosticCollector class, 423
 - DiagnosticListener interface, 423–424
 - diamond syntax
 - for array lists, 39
 - for constructors of generic classes, 201
 - directories, 284
 - checking for existence, 286, 288
 - creating, 286–288
 - deleting, 288, 290–291
 - moving, 287
 - temporary, 287
 - user, 285
 - visiting, 288–291
 - working, 346
 - directory method (ProcessBuilder), 346
 - disjoint method (Collections), 231
 - displayName method (Charset), 413
 - distinct method (Stream), 254, 268
 - dividedBy method (Duration), 381–382
 - divideUnsigned method (Integer, Long), 16
 - division, 14
 - do statement, 33
 - doc-files directory, 85
 - documentation comments, 84–89
 - @Documented annotation, 365, 367
 - domain names, using for package names, 73
 - dot notation, 5, 12
 - double brace initialization, 136
 - Double class, 40
 - compare method, 104
 - equals method, 141
 - isFinite, isInfinite methods, 9
 - NaN, NEGATIVE_INFINITY, POSITIVE_INFINITY values, 9
 - parseDouble method, 23
 - toString method, 23
 - double type, 8–9
 - atomic operations on, 331
 - functional interfaces for, 115
 - streams of, 266
 - type conversions of, 16–17
 - DoubleAccumulator, DoubleAdder classes, 331
 - DoubleConsumer, DoubleXXXOperator, DoublePredicate, DoubleSupplier, DoubleToXXXFunction interfaces, 115
 - DoubleFunction interface, 115, 212
 - doubles method (Random), 267
 - DoubleStream class, 266–267
 - DoubleSummaryStatistics class, 260, 267
 - doubleValue method (Number), 403
 - downstream collectors, 262–264, 269
 - Duration class
 - between method, 381
 - dividedBy, isZero, isNegative, minus, minusXXX, multipliedBy, negated, plus, plusXXX methods, 381–382
 - ofDays method, 384, 388
 - toXXX methods, 381
 - dynamic method lookup, 131–132, 210–211
 - dynamically typed languages, 432
- ## E
- E constant (Math), 16
 - e, E
 - conversion characters, 29
 - formatting symbols (date/time), 392
 - \e, \E, in regular expressions, 294–295
 - Eclipse, 4
 - ECMAScript standard, 435
 - edu global object (JavaScript), 431
 - effectively final variables, 118–119
 - efficiency, and final modifier, 133
 - Element interface, 372–373
 - element method (BlockingQueue), 327
 - elements (in annotations), 356–357
 - values of, 357, 363
 - else statement, 30

- em element (HTML), in documentation
 - comments, 85
- Emacs text editor, running jjs inside, 430
- empty method
 - of `Optional`, 258
 - of `Stream`, 252
- empty string, 22, 139
 - concatenating, 23
- emptyXXX methods (`Collections`), 232, 245
- encapsulation, 54
- encodings. *See* character encodings
- <<END, in shell scripts, 439
- endsWith method (`String`), 24
- engine scope, 425
- enhanced for loop, 41, 46, 119
 - for collections, 232
 - for enumerations, 148
 - for iterators, 160
 - for paths, 286
- entering, exiting methods (`Logger`), 189
- Entry class, 209
- entrySet method (`Map`), 237
- Enum class, 147–148
- enum instances
 - adding methods to, 149–150
 - construction, 149
 - referred by name, 151
- enum keyword, 13, 147
- enumeration sets, 241
- enumerations, 13, 147–151
 - adding constructors, methods, and fields to, 149
 - annotating, 358
 - comparing, 147–148
 - immutable empty, 244
 - nested inside classes, 151
 - serialization of, 305
 - static members of, 150–151
 - traversing instances of, 148
 - using in switch, 151
- EnumSet, EnumMap classes, 241
- \$ENV, in shell scripts, 440
- environment variables, modifying, 347
- epoch, definition of, 380
- equality, testing for, 18
- equals method
 - final, 142
 - for subclasses, 141
 - for values from different classes, 142
 - null-safe, 141
 - of `Arrays`, 141
 - of `Double`, 141
 - of `Instant`, 381
 - of `Object`, 138, 140–143
 - of `Objects`, 141
 - of `String`, 21–22
 - of wrapper classes, 40
 - overriding, 140–142
 - symmetric, 142
- equalsIgnoreCase method (`String`), 22
- \$ERR, in shell scripts, 438
- Error class, 175
- error messages, for generic methods, 202
- eval method (`ScriptEngine`), 425–427
- even numbers, 14
- EventHandler interface, 106
- Exception class, 176
- exceptions, 174–185
 - and generic types, 217–218
 - annotating, 360
 - catching, 178–182
 - in JavaScript, 437
 - chaining, 182–183–183
 - combining in a superclass, 177
 - creating, 176
 - documenting, 178
 - hierarchy of, 175–177
 - logging, 189
 - rethrowing, 180–183
 - suppressed, 180
 - throwing, 174–175
 - uncaught, 184
 - unchecked, 176
- exec method (`Runtime`), 345
- Executable class
 - getModifiers method, 164
 - getParameters method, 161
- ExecutableElement interface, 373
- ExecutionException, 315, 344
- Executor interface, 343
 - execute method, 313

- ExecutorCompletionService class, 316
- Executors class, 313–314
- ExecutorService interface, 314, 421
 - invokeAll method, 315
 - invokeAny method, 316
- exists method (Files), 286, 288
- exit function (shell scripts), 440
- \$EXIT, in shell scripts, 438
- exitValue method (Process), 348
- exportSubtree method (Preferences), 415
- expression closures, 435
- extends keyword, 98, 128, 202–206
- extension class loader, 156
- Externalizable interface, 304

F

- f conversion character, 29
- F suffix, 9
- \f, in regular expressions, 294
- factory methods, 64, 72
- failures, 344
 - logging, 182
- falling through, 32
- false value (boolean), 10
 - as default value, 65, 68
- Field interface, 160–161
 - get, getXXX, set, setXXX methods, 162–163
 - getModifiers, getName method, 160, 163
 - getType method, 160
- fields (instance and static variables), 127
 - final, 319
 - provided, 135
 - public, 161
 - retrieving values of, 161–162
 - setting, 162
 - transient, 303
- File class, 286
- file attributes
 - copying, 287
 - filtering paths by, 289
- file handlers
 - configuring, 192–193
 - default, 191
- file pointers, 282
- file.encoding system property, 278
- file.separator system property, 239

- FileChannel class
 - get, getXXX, put, putXXX methods, 283
 - lock, tryLock methods, 284
 - open method, 283
- FileFilter class, 114
- FileHandler class, 191–194
- FileNotFoundException, 176
- files
 - archiving, 291
 - channels to, 283
 - checking for existence, 176, 286–288
 - closing, 179
 - copying/moving, 287–288
 - creating, 285–288
 - deleting, 288
 - empty, 286
 - encoding of, 276–277
 - locking, 283–284
 - memory-mapped, 283
 - missing, 424
 - random-access, 282–283
 - reading from/writing to, 28, 176, 275
 - temporary, 287
- Files class
 - copy method, 275–276, 287–288, 291
 - createTempXXX methods, 287
 - createXXX methods, 286
 - delete, deleteIfExists methods, 288
 - exists method, 286, 288
 - find method, 288–289
 - isXXX methods, 286, 288
 - lines method, 252, 279
 - list method, 288–289
 - move method, 287–288
 - newBufferedReader method, 280, 424
 - newBufferedWriter method, 280, 287
 - newXXXStream methods, 274, 287, 302
 - readAllBytes method, 275, 279
 - readAllLines method, 279
 - walk method, 288–291
 - walkFileTree method, 288, 290
 - write method, 281, 287
- FileSystem, FileSystems classes, 291
- FileTime class, 394
- FileVisitor interface, 290

- fill method
 - of Arrays, 42
 - of Collections, 42, 231
- filter method (Stream), 251–253, 255
- Filter interface, 194
- final fields, 319
- final methods, 322
- final modifier, 12, 67, 133
- final variables, 318, 322
- finalize method
 - of Enum, 148
 - of Object, 138
- finally statement, 181–182
 - for locks, 332
- financial calculations, 9
- find method (Files), 288–289
- findAny method (Stream), 256
- findClass method (ClassLoader), 157
- findFirst method (Stream), 255
- fine method (Logger), 188
- first method (SortedSet), 234
- firstDayOfXXX methods (TemporalAdjusters), 385
- flag bits, sequences of, 240
- flatMap method
 - general concept of, 254
 - of Optional, 258–259
 - of Stream, 253
- flip method (BitSet), 240
- Float class, 40
- float type, 8–9
 - streams of, 266
 - type conversions of, 16–17
- floating-point types, 8–9
 - and binary number system, 9
 - comparing, 104
 - division of, 14
 - formatting for output, 29
 - in hexadecimal notation, 9
 - type conversions of, 16–17
- floor method (NavigableSet), 235
- floorMod method (Math), 15
- fonts, missing, 413
- for statement, 33–34
 - continuing, 34
 - declaring variables for, 36
 - enhanced, 41, 46, 119, 148, 232, 286
 - multiple variables in, 34
- for each loop (JavaScript), 434
- forEach method
 - of ArrayList, 110
 - of Map, 237
- forEach, forEachOrdered methods (Stream), 259
- forEachXXX methods (ConcurrentHashMap), 326
- ForkJoinPool class, 343
- forLanguageTag method (Locale), 402
- format method
 - of DateTimeFormatter, 390, 405
 - of MessageFormat, 408–410
- Format class, 393
- format specifiers, 28
- formatted output, 28–30
- Formatter class, 194
- formatters, for date/time values
 - custom, 392
 - locale-specific, 391
 - predefined, 390
- forName method
 - of Charset, 278
 - of Class, 152–153, 157–158, 176, 184, 423
- frequency method (Collections), 231
- from method (Instant, ZonedDateTime), 393
- full indicator, in string templates, 409
- Function interface, 114, 260
- function keyword (JavaScript), 435
- function types, 107
 - structural, 113
- functional interfaces, 109–110
 - as method parameters, 205–206
 - common, 114
 - contravariant in parameter types, 205
 - for primitive types, 115
 - implementing, 115–116
- @FunctionalInterface annotation, 116, 364, 366–367
- functions, 54
 - higher-order, 120–122
- Future interface, 315
- futures, 315
 - combining, 344–345
 - completable, 342–344
 - in order of completion, 316

G

- G formatting symbol (date/time), 392
- g, G conversion characters, 29
- \G, in regular expressions, 297
- %g pattern variable, 193
- garbage collector, 243
- generate method (Stream), 252, 266
- @Generated annotation, 365–366
- generators, converting to streams, 268
- generic classes, 39, 200–201
 - constructing objects of, 201
 - information available at runtime, 220
 - instantiating, 200
- generic collections, 245
- generic constructors, 220
- generic methods, 201–202
 - calling, 201
 - declaring, 201
 - information available at runtime, 220
- generic type declarations, 220–221
- generic types, 103
 - and exceptions, 217–218
 - and lambda expressions, 205
 - and reflection, 218–221
 - annotating, 359
 - arrays of, 112
 - casting, 212
 - in JVM, 208–211
 - invariant, 203, 205
 - restrictions on, 211–218
- GenericArrayType interface, 220
- get method
 - of Array, 166
 - of ArrayList, 40
 - of BitSet, 240
 - of CompletableFuture, 344
 - of Field, 162–163
 - of FileChannel, 283
 - of Future, 315
 - of List, 229
 - of LongAccumulator, 331
 - of Map, 235–236
 - of Optional, 257
 - of Path, 284–286
 - of Preferences, 414–415
 - of Supplier, 114
 - of ThreadLocal, 340
- getAndXXX methods (AtomicXXX), 330
- getAnnotation, getAnnotationsByType methods (AnnotatedConstruct), 373
- getAnnotationXXX methods (AnnotatedElement), 369–371
- getAsXXX methods
 - of OptionalXXX, 267
 - of XXXSupplier, 115
- getAudioClip method (Applet), 155
- getAvailableCurrencies method (Currency), 404
- getAvailableIds method (ZoneId), 387
- getAvailableLocales method (Locale), 401
- getAverage method (XXXSummaryStatistics), 260
- getBundle method (ResourceBundle), 411–412
- getCanonicalName method (Class), 153
- getClass method (Object), 133, 138, 141, 152, 213, 219
- getClassLoader method (Class), 154, 156
- getComponentType method (Class), 154, 165
- getConstructor(s) methods (Class), 154, 161, 163, 219
- getContents method (ListResourceBundle), 412
- getContextClassLoader method (Thread), 158
- getCountry method (Locale), 262
- getCurrencyInstance method (NumberFormat), 403
- getDayXXX, getMonthXXX, getYear methods
 - of LocalDate, 55, 383–384
 - of LocalTime, 386
 - of ZonedDateTime, 389
- getDeclaredAnnotationXXX methods (AnnotatedElement), 369–371
- getDeclaredConstructor(s) methods (Class), 154, 161, 219
- getDeclaredField(s) methods (Class), 154
- getDeclaredMethod(s) methods (Class), 154, 162
- getDeclaringClass method
 - of Class, 154
 - of Enum, 148
- getDefault method (Locale), 401–402
- getDisplayDefault method (Locale), 411
- getDisplayName method
 - of Currency, 404
 - of DayOfWeek, Month, 392, 406
 - of Locale, 402

- getElementsAnnotatedWith method
 - (RoundEnvironment), 373
- getEnclosedElements method (TypeElement), 373
- getEnclosingXXX methods (Class), 154
- getEngineXXX methods (ScriptEngineManager), 424
- getEnumConstants method (Class), 219
- getErrorStream method (Process), 347
- getField(s) methods (Class), 154, 161
- getFileName method (Path), 286
- getFilePointer method (RandomAccessFile), 283
- getGlobal method (Logger), 187
- getHead method (Formatter), 194
- getHeaderFields, getInputStream methods
 - (URLConnection), 292
- getInstance method
 - of Collator, 407
 - of Currency, 404
- getInterfaces method (Class), 153
- getLength method (Array), 166
- getLogger method (Logger), 188
- getMax method (XXXSummaryStatistics), 260
- getMethod(s) methods (Class), 154, 161–162
- getMethodCallSyntax method
 - (ScriptEngineFactory), 427
- getModifiers method
 - of Class, 153
 - of Constructor, 160
 - of Executable, 164
 - of Field, 160, 163
 - of Method, 160
- getName method
 - of Class, 152–153
 - of Constructor, 160
 - of Field, 160, 163
 - of Method, 160
 - of Parameter, 164
 - of Path, 286
 - of PropertyDescriptor, 165
- getNumberInstance method (NumberFormat), 403
- getObject method (ResourceBundle), 412
- getOrDefault method (Map), 235–236
- getOrElse method (Optional), 255
- getOutputStream method (URLConnection), 292
- getPackage method (Class), 153
- getParameters method (Executable), 161
- getParent method (Path), 286
- getPath method (FileSystem), 291
- getPercentInstance method (NumberFormat), 403
- getProperties method (System), 239
- getPropertyDescriptors method (BeanInfo), 165
- getPropertyType method (PropertyDescriptor), 165
- getQualifiedName method (TypeElement), 373
- getReadMethod method (PropertyDescriptor), 165
- getResource method (Class), 155, 410
- getResourceAsStream method (Class), 154–155
- getRoot method (Path), 286
- getSimpleName method
 - of Class, 153
 - of Element, 373
- getStackTrace method (Throwable), 184
- getString method (ResourceBundle), 411
- getSuperClass method (Class), 153, 219
- getSuppressed method (IOException), 181
- getSymbol method (Currency), 404
- getSystemJavaCompiler method (ToolProvider), 420
- getTail method (Formatter), 194
- getTask method (JavaCompiler), 420–421
- getType method (Field), 160
- getTypeName method (Class), 153
- getTypeParameters method (Class), 220
- getURLs method (URLClassLoader), 156
- getValue method (LocalDate), 55
- getWriteMethod method (PropertyDescriptor), 165
- getXXX methods (Array), 166
- getXXX methods (Field), 162–163
- getXXX methods (FileChannel), 283
- getXXX methods (Preferences), 415
- getXXXInstance methods (NumberFormat), 72
- getXXXStream methods (Process), 346
- GlassFish administration tool, 439
- Goetz, Brian, 311
- Gregorian calendar reform, 383
- GregorianCalendar class, 393–394
 - toZonedDateTime method, 393–394
- grouping, 262
 - classifier functions of, 262
 - reducing to numbers, 263
- groupingBy method (Collectors), 262–264
- groupingByConcurrent method (Collectors), 262, 269

GUI (graphical user interface)
 callbacks in, 106–107
 long-running tasks in, 341–342
 missing fonts in, 413

H

H formatting symbol (date/time), 392
h, H conversion characters, 29
\\h, \\H, in regular expressions, 295
%h pattern variable, 193
handle method (CompletableFuture), 344
Handler class, 194
Hansen, Per Brinch, 334
hash method (Object), 144
hash codes, 143–144
 computing in String class, 143
 formatting for output, 29
hash functions, 143–144, 234
hash maps
 concurrent, 325–326
 weak, 243
hash tables, 234
hashCode method
 of Arrays, 144
 of Enum, 148
 of Object, 138, 140, 143–144
HashMap class, 235
 null values in, 238
HashSet class, 234
Hashtable class, 334
hasNext method (Iterator), 232
hasNext, hasNextXXX methods (Scanner), 27, 279
headMap method (SortedMap), 244
headSet method
 of NavigableSet, 235
 of SortedSet, 234, 244
heap pollution, 212–213, 245
HelloWorld class, 2
helper methods, 208
here documents, 439
hexadecimal numbers, 8–9
 formatting for output, 29
higher method (NavigableSet), 235
higher-order functions, 120–122
hn, hr elements (HTML), in documentation
 comments, 85

Hoare, Tony, 334
HTML documentation, generating,
 376
URLConnection class, 292
hyperlinks, 87–88, 293

I

[I prefix, 140, 153
IANA (Internet Assigned Numbers
 Authority), 387
IDE (integrated development
 environment), 3–4
identity method
 of Function, 114, 260
 of UnaryOperator, 114
identity values, 265
if statement, 30–31
ifPresent, isPresent methods (Optional), 257
IllegalArgumentException, 185
IllegalStateException, 260, 327
ImageIcon class, 155
images, locating, 155
img element (HTML), in documentation
 comments, 85
immutability, 321
immutable classes, 322
implements keyword, 95–96
import statement, 6, 77–78
 no annotations for, 360
 static, 78–79
import static statement, 151
importPreferences method (Preferences), 415
increment method (LongAdder), 330
increment operator, 15
incrementAndGet method (AtomicXXX), 329
incrementExact method (Math), 16
indexOf method
 of List, 229
 of String, 24
indexOfSubList method (Collections), 231
info method (Logger), 187
inheritance, 128–147
 and default methods, 136–137
 classes win rule, 137, 144
@Inherited annotation, 365, 367
initCause method (Throwable), 183

- initialization blocks, 66–67
 - static, 70–71
- inlining, 133
- inner classes, 81–83
 - capturing this references in, 111
 - invoking methods of outer classes, 82
 - local, 122–123
 - syntax for, 83
- input
 - reading, 27–28
 - redirecting, 426
- input prompts, 28
- input streams, 274
 - copying into output streams, 276
 - obtaining, 274
 - reading from, 275
- InputStream class, 275
- InputStreamReader class, 279
- INSTANCE instance (enum types), 305
- instance methods, 5, 60–61
- instance variables, 59, 61–62
 - annotating, 358
 - comparing, 141
 - default values of, 65–66
 - final, 67
 - in abstract classes, 134
 - in JavaScript, 436
 - initializing, 66–67, 131
 - not accessible from static methods, 72
 - of deserialized objects, 304–306
 - protected, 135
 - setting, 64
 - transient, 303
 - vs. local, 66
- instanceof operator, 98, 132, 141–142
 - annotating, 360
- instances, 5
- Instant class, 380
 - and legacy classes, 394
 - compareTo, equals methods, 381
 - from method, 393
 - minus, minusXXX, plus, plusXXX methods, 381–382
 - now method, 381
- instruction reordering, 318
- int type, 7–8
 - functional interfaces for, 115
 - processing values of, 113
 - random number generator for, 6, 32
 - streams of, 266
 - type conversions of, 16–17
 - using class literals with, 152
- IntBinaryOperator interface, 115
- IntConsumer interface, 113, 115
- Integer class, 40
 - compare method, 104
 - MIN_VALUE, MAX_VALUE constants, 7
 - parseInt method, 23, 176
 - toString method, 23
 - xxxUnsigned methods, 16
- integer indicator, in string templates, 409
- integer types, 7–8
 - comparing, 104
 - computations of, 16
 - division of, 14
 - even or odd, 14
 - formatting for output, 29
 - in hexadecimal notation, 8
 - reading/writing, 282–283
 - type conversions of, 16–17
- @interface declaration, 361–363
- interface keyword, 95
- interface methods, 100–102
- interfaces, 94–99
 - annotating, 358–359
 - compatibility of, 100–101
 - declarations of, 94–95
 - defining variables in, 99
 - evolution of, 100
 - extending, 98
 - functional, 109–110
 - implementing, 95–97
 - in JavaScript, 435–437
 - in scripting engines, 427
 - multiple, 98–99
 - methods of, 95–96
 - no instance variables in, 99
 - no redefining methods of the Object class in, 144
 - views of, 244
- Internet Engineering Task Force, 399

- interrupted method (Thread), 338
 - interrupted status, 338
 - InterruptedException, 337, 339
 - interruption requests, 315
 - intersects method (BitSet), 241
 - IntFunction interface, 115, 212
 - IntPredicate interface, 115
 - intrinsic locks, 333–334
 - ints method (Random), 267
 - IntSequence interface, 122
 - IntStream class, 266–267
 - parallel method, 267, 323
 - IntSummaryStatistics class, 260, 267
 - IntSupplier, IntToXXXFunction, IntUnaryOperator
 - interfaces, 115
 - InvalidClassException, 306
 - InvalidPathException, 284
 - Invocable interface, 426–427
 - InvocationHandler interface, 167
 - invoke method (Method), 162, 164
 - invokeAll method (ExecutorService), 315
 - invokeAny method (ExecutorService), 316
 - IOException, 176, 279
 - addSuppressed, getSuppressed methods, 181
 - isAfter, isBefore methods
 - of LocalDate, 383
 - of LocalTime, 386
 - of ZonedDateTime, 389
 - isAlive method (Process), 348
 - isCancelled, isDone methods (Future), 315
 - isEmpty method
 - of BitSet, 241
 - of Collection, 229
 - of Map, 237
 - isEqual method (Predicate), 114–115
 - isFinite, isInfinite methods (Double), 9
 - isInterrupted method (Thread), 315, 338
 - isLoggable method (Filter), 194
 - isNamePresent method (Parameter), 164
 - isNull method (Objects), 110
 - ISO 8601 format, 366
 - ISO 8859-1 encoding, 277, 281
 - isXXX methods (Class), 153–154, 165
 - isXXX methods (Files), 286, 288
 - isXXX methods (Modifier), 155, 160–161
 - isZero, isNegative methods (Duration), 381
 - Iterable interface, 232–233, 286, 434
 - iterator method, 232
 - iterate method (Stream), 252, 255, 259, 266
 - iterator method
 - of Collection, 229
 - of ServiceLoader, 160
 - Iterator interface
 - next, hasNext methods, 232
 - remove, removeIf methods, 233
 - iterators, 232–233, 259
 - converting to streams, 268
 - for random numbers, 435
 - immutable empty, 244
 - invalid, 233
 - traversing, 160
 - weakly consistent, 325
- ## J
- JAR files, 74–76
 - resources in, 155, 410
 - sealed, 77
 - jar program, 74, 77
 - Java Persistence Architecture, 355
 - java program, 4
 - classpath (-cp) option, 75
 - disableassertions (-da) option, 186
 - enableassertions (-ea) option, 186
 - enablesystemassertions (-esa) option, 186
 - specifying locales in, 402
 - Java programming language
 - compatibility with older versions of, 137, 208
 - online API documentation on, 24
 - portability of, 15
 - strongly typed, 10
 - Unicode support in, 25–26
 - uniformity of, 3, 102
 - java, javax, javafx global objects (JavaScript), 431
 - java.awt package, 76
 - java.class.path, java.home, java.io.tmpdir system properties, 239
 - Java.extend function (JavaScript), 435–436
 - Java.from function (JavaScript), 434
 - java.lang, java.lang.annotation packages, 364
 - java.lang.reflect package, 160

- java.sql package, 393
 - Java.super function (JavaScript), 436–437
 - java.time package, 379–394
 - Java.to function (JavaScript), 434
 - Java.type function (JavaScript), 431–432
 - java.util package, 6, 325
 - java.util.concurrent package, 325, 328
 - java.util.concurrent.atomic package, 329
 - java.version system property, 239
 - JavaBeans, 164–165
 - javac program, 4
 - author option, 89
 - classpath (-cp) option, 75
 - d option, 74, 89
 - encoding option, 413
 - link, -linksource options, 89
 - parameters option, 161
 - processor option, 372
 - version option, 89
 - Xlint option, 32
 - XprintRounds option, 375
 - JavaCompiler interface, 420–421
 - javadoc program, 84–89
 - including annotations in, 367
 - JavaFileObject interface, 421
 - JavaFX platform, 106–107
 - and threads, 342
 - java.n.log file, 192
 - JavaScript programming language
 - accessing classes of, from Java, 428
 - anonymous functions in, 435
 - anonymous subclasses in, 436
 - arrays in, 433–434
 - bracket notation in, 430, 433–434
 - calling static methods in, 431
 - catching Java exceptions in, 437
 - constructing Java objects in, 431–432
 - delimiters in, 426
 - extending Java classes in, 435–437
 - implementing Java interfaces in, 435–437
 - inner classes in, 432
 - instance variables in, 436
 - lists and maps in, 434
 - methods in, 430
 - numbers in, 432–433
 - objects in, 432
 - REPL in, 429–430
 - semicolons in, 426
 - strings in, 432
 - superclasses in, 436
 - JavaServer Faces framework, 238
 - javax.annotation package, 364
 - jconsole program, 191
 - JDK (Java Development Kit), 3
 - jjs tool, 429–430
 - command-line arguments in, 439
 - executing commands in, 438
 - job scheduling, 242
 - join method
 - of String, 20
 - of Thread, 337
 - joining method (Collectors), 259–260
 - jre/lib/ext directory, 76
 - JSP (JavaServer Pages), 440
 - JUnit, 355–356
- K**
- ␣ formatting symbol (date/time), 392
 - \k, in regular expressions, 296
 - key/value pairs
 - adding new keys to, 235
 - in annotations. *See* elements
 - removed by garbage collector, 243
 - values of, 235
 - keys method (Preferences), 415
 - keySet method
 - of ConcurrentHashMap, 329
 - of Map, 237, 244
 - keywords, 11
- L**
- l suffix, 8
 - [l prefix, 153
 - labeled statements, 35
 - lambda expressions, 107–110
 - and generic types, 205
 - annotating targets for, 366
 - capturing variables in, 117–119
 - executing, 113
 - for loggers, 187
 - parameters of, 108

- processing, 112–116
- return type of, 109
- scope of, 116–117
- this reference in, 117
- throwing exceptions in, 178
- using with streams, 253
- vs. anonymous functions (JavaScript), 435
- with parallel streams, 323
- language codes, 262, 399–400
- language model API, 372–373
- last method (SortedSet), 234
- lastIndexOf method
 - of List, 229
 - of String, 24
- lastIndexOfSubList method (Collections), 231
- lastXXX methods (TemporalAdjusters), 385
- lazy operations, 251, 255, 269, 299
- leap seconds, 380
- leap years, 383–384
- legacy code, 393–394
- length method, 38
 - of arrays, 38
 - of RandomAccessFile, 283
 - of String, 6, 26
- .level suffix, 190
- limit method (Stream), 254, 268
- line feed, 10
 - formatting for output, 29
 - in regular expressions, 297
- line.separator system property, 240
- lines method
 - of BufferedReader, 280
 - of Files, 252, 279
- @link tag (javadoc), 87–88
- linked lists, 230, 233
- LinkedBlockingQueue class, 328
- LinkedHashMap class, 238
- LinkedList class, 230
- list method (Files), 288–289
- List interface, 206–207, 229–230
 - add, get, indexOf, lastIndexOf, listIterator, remove, replaceAll, set, sort methods, 229
 - subList method, 229, 244
- ListIterator interface, 233
- ListResourceBundle class, 412
- lists
 - converting to streams, 268
 - immutable empty, 244
 - in Nashorn, 434
 - printing elements of, 110
 - removing null values from, 110
 - sublists of, 244
 - unmodifiable views of, 245
- little-endian format, 277
- load method (ServiceLoader), 160
- load balancing, 301
- loadClass method (ClassLoader), 157
- local classes, 122–123
- local date/time, 382–387
- local variables, 36–37
 - annotating, 358–359
 - vs. instance, 66
- LocalDate class, 55
 - and legacy classes, 394
 - getXXX methods, 55, 383–384
 - isXXX methods, 383
 - minus, minusXXX methods, 383–384
 - now method, 64, 71, 382–383
 - of method, 55, 64, 382–383
 - parse method, 405
 - plus, plusXXX methods, 55–56, 58, 383–384
 - until method, 383
 - withXXX methods, 383
- LocalDateTime class, 387
 - and legacy classes, 394
 - atZone method, 387
 - parse method, 405
- Locale class, 261
 - forLanguageTag method, 402
 - get/setDefault methods, 401–402
 - getAvailableLocales method, 401
 - getCountry method, 262
 - getDisplayDefault methods, 411
 - getDisplayName method, 402
 - predefined fields, 401
- locales, 260–264, 398–402
 - date/time formatting for, 404–406
 - default, 392, 401–402, 404–405, 411
 - displaying names of, 402
 - for template strings, 408–410

- locales (*cont.*)
 - formatting styles for, 391, 405
 - sorting words for, 406–408
 - specifying, 399–401
 - weekdays and months in, 392
 - LocalTime class, 386–387
 - and legacy classes, 394
 - final, 133
 - getXXX, isXXX, minus, minusXXX, now, of, plus, plusXXX, toXXXOfDay, withXXX methods, 386
 - parse method, 405
 - lock method
 - of FileChannel, 284
 - of ReentrantLock, 332
 - locks, 321
 - intrinsic, 333–334
 - reentrant, 331–332
 - releasing, 181, 318
 - log handlers, 191–193
 - default, 188, 191
 - filtering/formatting, 194
 - installing custom, 191
 - levels of, 191
 - suppressing messages in, 188
 - Logger class
 - addHandler method, 191
 - entering, exiting methods, 189
 - fine method, 188
 - getGlobal method, 187
 - getLogger method, 188
 - info method, 187
 - log method, 188–189
 - logp, logrb methods, 190
 - setFilter method, 194
 - setLevel method, 187–188, 191
 - setUseParentHandlers method, 191
 - throwing method, 189–190
 - warning method, 188
 - loggers
 - defining, 187–188
 - filtering/formatting, 194
 - hierarchy of, 188
 - logging, 187–194
 - configuring, 188–191
 - enabling/disabling, 188
 - failures, 182
 - levels of, 188–191
 - localizing, 190
 - overriding methods for, 133
 - using for unexpected exceptions, 189
 - Long class, 40
 - MIN_VALUE, MAX_VALUE constants, 7
 - xxxUnsigned methods, 16
 - long indicator, in string templates, 409
 - long type, 7–8
 - atomic operations on, 330–331
 - functional interfaces for, 115
 - streams of, 266
 - type conversions of, 16–17
 - LongAccumulator class, 330
 - accumulate, get methods, 331
 - LongAdder class, 330–331
 - add, increment, sum methods, 330
 - LongConsumer, LongXXXOperator, LongPredicate, LongSupplier, LongToXXXFunction interfaces, 115
 - LongFunction interface, 115, 212
 - longs method (Random), 267
 - LongStream class, 266–267
 - LongSummaryStatistics class, 260, 267
 - long-term persistence, 306
 - loops, 32–34
 - exiting, 34–35
 - infinite, 34
- ## M
- m, M formatting symbols (date/time), 392–393
 - main method, 2, 5
 - decomposing, 46–48
 - string array parameter of, 43
 - map method
 - of Optional, 257
 - of Stream, 253
 - Map interface, 230
 - clear method, 237
 - compute, computeIfXXX methods, 236
 - containsXXX methods, 237
 - entrySet method, 237
 - forEach method, 237
 - get, getOrDefault methods, 235–236

- isEmpty method, 237
- keySet method, 237, 244
- merge method, 235–236
- put method, 235–236
- putAll, putIfAbsent methods, 236
- remove method, 236
- replace method, 236
- replaceAll method, 237
- size method, 237
- values method, 237, 244
- mapping method (Collectors), 263–264
- maps, 235–238
 - concurrent, 236, 261
 - empty, 237
 - immutable empty, 244
 - in Nashorn, 434
 - iterating over, 237
 - of stream elements, 260–261, 269
 - order of elements in, 238
 - unmodifiable views of, 245
 - views of keys, values, and entries of, 237
- mapToInt method (Stream), 265
- mapToXXX methods (XXXStream), 267
- marker interfaces, 145
- Matcher class, 298
 - replaceAll method, 300
- matcher, matches methods (Pattern), 298
- Math class
 - E constant, 16
 - floorMod method, 15
 - max, min methods, 16
 - PI constant, 16, 69
 - pow method, 15, 71
 - round method, 17
 - sqrt method, 15
 - xxxExact methods, 16–17
- max method
 - of Stream, 255
 - of XXXStream, 267
- MAX_VALUE constant (integer classes), 7
- maxBy method
 - of BinaryOperator, 114
 - of Collectors, 263
- medium indicator, in string templates, 409
- members (fields, methods, nested classes/interfaces), 127
 - enumerating, 160–161
- memory allocation, 321
- memory-mapped files, 283
- merge method
 - of ConcurrentHashMap, 326
 - of Map, 235–236
- Message class, 146–147
- MessageFormat class, 408–410
- meta-annotations, 362–368
- Method interface, 160–161
 - getModifiers method, 160
 - getName method, 160
 - invoke method, 162, 164
- method calls, 5
 - receiver of, 61
- method comments, 86
- method expressions, 110, 137
- method references, 110–111, 213
 - annotating, 360
- methods, 2
 - abstract, 109, 133–134
 - accessor, 56
 - annotating, 358
 - atomic, 326
 - body of, 60
 - chaining calls of, 56
 - clashes of, 216–217
 - compatible, 143
 - declarations of, 59
 - default, 100–102
 - deprecated, 87
 - factory, 64, 72
 - final, 133, 322
 - header of, 59
 - inlining, 133
 - instance, 60–61
 - invoking, 162–163
 - modifying functions, 120
 - mutator, 56
 - names of, 11
 - native, 70
 - overloading, 65, 111
 - overriding, 100, 129–130, 133, 177–178, 365

- methods (*cont.*)
 - parameters of, 161
 - null checks for, 185
 - passing arrays into, 47
 - proxied, 168
 - public, 95–96, 161
 - restricted to subclasses, 134–135
 - return value of, 2, 60
 - returning functions, 120
 - static, 47, 71–72, 78, 99–100
 - storing in variables, 6
 - symmetric, 142
 - synchronized, 333–336
 - utility, 76
 - variable number of arguments of, 48
- Microsoft Notepad, 277
- Microsoft Windows
 - batch files, 437–438
 - path separator, 75, 240
 - registry, 414
- min method
 - of Math, 16
 - of Stream, 255
 - of XXXStream, 267
- MIN_VALUE constant (integer classes), 7
- minBy method
 - of BinaryOperator, 114
 - of Collectors, 263
- minus, minusXXX methods
 - of Instant, Duration, 381–382
 - of LocalDate, 383–384
 - of LocalTime, 386
 - of ZonedDateTime, 389
- Modifier interface
 - isXXX methods, 155, 160–161
 - toString method, 155
- modifiers, checking, 160
- monads, 254
- monitors (classes), 334
- Month enumeration, 382–383, 389
 - getDisplayName method, 392, 406
- MonthDay class, 384
- move method (Files), 287–288
- Mozilla JavaScript implementation, 435
- multiplication, 14
- multipliedBy method (Duration), 381–382
- mutators, 56
 - and unmodifiable views, 245
- N**
 - n**
 - conversion character, 29
 - formatting symbol (date/time), 393
 - `\n`
 - for character literals, 10
 - in regular expressions, 294–296, 301
 - newline, in property files, 239–240
 - name method (Enum), 148
 - NaN (not a number), 9
 - Nashorn engine, 424, 428–437
 - anonymous subclasses in, 436
 - arrays in, 433–434
 - catching Java exceptions in, 437
 - class objects in, 431
 - extending Java classes in, 435–437
 - getters/setters in, 430
 - implementing Java interfaces in, 435–437
 - instance variables in, 436
 - lists and maps in, 434
 - methods in, 430
 - no standard input source in, 426
 - numbers in, 432–433
 - running from command line, 429
 - shell scripting in, 437–440
 - strings in, 432
 - superclasses in, 436
 - native methods, 70
 - native2ascii tool, 412
 - naturalOrder method (Comparator), 121–122
 - navigable maps/sets
 - immutable empty, 244
 - unmodifiable views of, 245
 - NavigableMap interface, 328
 - NavigableSet interface, 230, 234, 244
 - methods of, 235
 - nCopies method (Collections), 230–231
 - negate method (Predicate, BiPredicate), 114
 - negated method (Duration), 381–382
 - negateExact method (Math), 16
 - NEGATIVE_INFINITY value (Double), 9
 - negative values, 7

- nested classes, 79–84
 - annotating, 359
 - inner, 81–83
 - public, 80
 - static, 79–80
 - new operator, 6, 10, 13, 64
 - as constructor reference, 111
 - for anonymous classes, 123
 - for arrays, 37–38, 44
 - in JavaScript, 431–433, 436
 - newBufferedReader method (Files), 280, 424
 - newBufferedWriter method (Files), 280, 287
 - newFileSystem method (FileSystems), 291
 - newInputStream method (Files), 274, 287, 302
 - newInstance method
 - of Array, 166
 - of Class, 154, 163, 219
 - of Constructor, 163–164
 - newKeySet method (ConcurrentHashMap), 328
 - newline. *See* line feed
 - newOutputStream method (Files), 274, 287, 302
 - newProxyInstance method (Proxy), 167
 - newXXXThreadPool methods (Executors), 313–314
 - next method (Iterator), 232
 - next, nextOrSame methods (TemporalAdjusters), 385
 - next, nextXXX methods (Scanner), 27, 279
 - nextInt method (Random), 6, 32
 - nextXXXBit methods (BitSet), 241
 - nominal typing, 113
 - noneMatch method (Stream), 256
 - noneOf method (EnumSet), 241
 - noninterference, of stream operations, 269
 - @NonNull annotation, 359
 - normalize method (Path), 285
 - Normalizer class, 408
 - NoSuchElementException, 257, 327
 - notify, notifyAll methods (Object), 138, 336–337
 - NotSerializableException, 303
 - now method
 - of Instant, 381
 - of LocalDate, 64, 71, 382–383
 - of LocalTime, 386
 - of ZonedDateTime, 389
 - null value, 22, 58
 - as default value, 65, 68
 - checking parameters for, 185
 - comparing against, 141
 - converting to strings, 139
 - NullPointerException, 22, 39, 58, 66, 176, 185, 235
 - vs. Optional, 255
 - nullsFirst, nullsLast methods (Comparator), 121
 - Number class, 403
 - number indicator, in string templates, 409
 - Number type (JavaScript), 432–433
 - NumberFormat class
 - getXXXInstance methods, 72, 403
 - not threadsafe, 339–340
 - parse method, 403
 - setCurrency method, 404
 - NumberFormatException, 176
 - numbers
 - big, 19
 - comparing, 104
 - converting to strings, 23
 - default value of, 65, 68
 - even or odd, 14
 - formatting, 29, 398, 403, 408
 - from grouped elements, 263
 - in regular expressions, 295
 - non-negative, 186, 240
 - printing, 28
 - random, 6, 32, 252, 254, 267
 - reading/writing, 279, 282–283
 - rounding, 9, 17
 - type conversions of, 16–17
 - unsigned, 8, 16
 - with fractional parts, 8–9
- ## O
- o conversion character, 29
 - Object class, 137–147
 - clone method, 135, 138, 144–147, 163
 - equals method, 138, 140–143
 - finalize method, 138
 - getClass method, 133, 138, 141, 152, 213, 219
 - hashCode method, 138, 140, 143–144
 - notify, notifyAll methods, 138, 336–337

- Object class (*cont.*)
 - toString method, 138–140
 - wait method, 138, 335–337
- object references, 56–58
 - and serialization, 302
 - attempting to change, 63
 - comparing, 140
 - default value of, 65, 68
 - null, 58
 - passed by value, 63
- ObjectInputStream class, 302–303
 - defaultReadObject method, 304
 - readFields method, 307
 - readObject method, 302–304, 306–307
- object-oriented programming, 53–91
- ObjectOutputStream class, 302
 - defaultWriteObject method, 304
 - writeObject method, 302–304
- objects, 2, 54–58
 - calling methods on, 6
 - casting, 97–98
 - cloning, 144–147
 - comparing, 40, 140–143
 - constructing, 6, 63–68, 163–164
 - in JavaScript, 431–432
 - converting to strings, 138–140
 - deep/shallow copies of, 144–146
 - deserialized, 304–306
 - externalizable, 304
 - immutable, 56
 - initializing variables with, 10
 - inspecting, 161–162
 - invoking static methods on, 71
 - mutable, 67
 - serializable, 301–303
 - sorting, 103–104
 - state of, 54
- Objects class
 - equals method, 141
 - hash method, 144
 - isNull method, 110
 - requireNonNull method, 185
- ObjXXXConsumer interfaces, 115
- octal numbers, 8
 - formatting for output, 29
- octonions, 26
- odd numbers, 14
- of method
 - of EnumSet, 241
 - of IntStream, 266
 - of LocalDate, 55, 64, 382–383
 - of LocalTime, 386
 - of Optional, 258
 - of Stream, 251–252
 - of ZonedDateTime, 387–389
- ofDateAdjuster method (TemporalAdjusters), 385
- ofDays method (Duration, Period), 384, 388
- offer method (BlockingQueue), 327
- offsetByCodePoints method (String), 26
- OffsetDateTime class, 390
- ofInstant method (ZonedDateTime), 389
- ofLocalizedXXX methods (DateTimeFormatter), 391, 405
- ofNullable method (Optional), 258
- ofPattern method (DateTimeFormatter), 392
- open method (FileChannel), 283
- openConnection method (URL), 292
- openStream method (URL), 274
- Operation interface, 150
- operations
 - associative, 265
 - atomic, 319, 321, 329–331
 - bulk, 326
 - lazy, 251, 255, 269, 299
 - performed optimistically, 330
 - stateless, 267
 - threadsafe, 324–329
- operators, 13–19
 - precedence of, 14
- Optional class, 255–259
 - creating values of, 258
 - empty method, 258
 - flatMap method, 258–259
 - for empty streams, 264–265
 - get method, 257
 - getOrElse method, 255
 - ifPresent, isPresent methods, 257
 - map method, 257
 - of, ofNullable methods, 258
 - orElse, orElseXXX methods, 256
- OptionalXXX classes, 267

- or method
 - of BitSet, 241
 - of Predicate, BiPredicate, 114
- order method (ByteBuffer), 283
- ordinal method (Enum), 148
- org global object (JavaScript), 431
- os.arch, os.name, os.version system properties, 239
- \$OUT, in shell scripts, 438
- output
 - formatted, 28–30
 - redirecting, 426
- output streams, 274
 - closing, 276
 - copying from input streams, 276
 - obtaining, 274
 - writing to, 276
- OutputStream class, 302
 - write method, 276
- OutputStreamWriter class, 280
- @Override annotation, 130, 364–365
- overriding, 129–130
 - for logging/debugging, 133
- overview.html file, 88
- P**
 - \p, \P, in regular expressions, 295
 - package statement, 73
 - package comments, 88
 - package declarations, 73–74
 - Package object (JavaScript), 431
 - package-info.java file, 88, 358
 - packages, 3, 72–79
 - accessing, 135
 - adding classes to, 77
 - annotating, 358–359
 - default, 73
 - names of, 73
 - not nesting, 73
 - scope of, 76–77
 - parallel method (XXXStream), 267, 323
 - parallel streams, 323
 - parallelStream method (Collection), 229, 250–251, 267, 323
 - parallelXXX methods (Arrays), 43, 324
 - @param tag (javadoc), 86
 - Parameter class, 164
 - parameter variables, 62
 - annotating, 358
 - scope of, 36
 - ParameterizedType interface, 220
 - parse method
 - of DateTimeFormatter, 393
 - of LocalXXX, ZonedDateTime, 405
 - of NumberFormat, 403
 - parseDouble method (Double), 23
 - ParseException, 403
 - parseInt method (Integer), 23, 176
 - partitioning, 322
 - partitioningBy method (Collectors), 262, 264
 - Pascal triangle, 45
 - passwords, 27
 - Path interface, 100, 284–286
 - get method, 284–286
 - getXXX methods, 286
 - normalize, relativize methods, 285
 - resolve, resolveSibling methods, 285
 - subpath method, 286
 - toAbsolutePath, toFile methods, 285
 - path separators, 284
 - path.separator system property, 240
 - paths, 284
 - absolute vs. relative, 284–285
 - filtering, 289
 - resolving, 285
 - taking apart/combining, 286
 - Paths class, 100
 - Pattern class
 - compile method, 298, 300
 - flags, 300–301
 - matcher, matches methods, 298
 - split method, 299
 - splitAsStream method, 252, 299
 - pattern variables, 193
 - PECS (producer extends, consumer super), 206
 - peek method
 - of BlockingQueue, 327
 - of Stream, 255
 - percent indicator, in string templates, 409
 - performance, and atomic operations, 330

- Period class, 383–384
 - ofDays method, 388
- @Persistent annotation, 368
- PI constant (Math), 16, 69
- placeholders, 408–410
- platform encoding, 278, 413
- plugins, loading, 156–160
- plus, plusXXX methods
 - of Instant, Duration, 381–382
 - of LocalDate, 55–56, 58, 383–384
 - of LocalTime, 386
 - of ZonedDateTime, 388–389
- Point class, 138–140
- Point2D class (JavaFX), 303
- poll method (BlockingQueue), 327–328
- pollXXX methods (NavigableSet), 235
- pop method (ArrayDeque), 242
- POSITIVE_INFINITY value (Double), 9
- @PostConstruct annotation, 364, 366
- pow method (Math), 15, 71
- predefined character classes, 294–295, 297
- @PreDestroy annotation, 364, 366
- predicate functions, 262
- Predicate interface, 109–110, 114
 - and, or, negate methods, 114
 - isEqual method, 114–115
 - test method, 114, 205
- Preferences class, 413–415
- previous method (ListIterator), 233
- previous, previousOrSame methods
 - (TemporalAdjusters), 385
- previousXXXBit methods (BitSet), 241
- preVisitDirectory, postVisitDirectory methods
 - (FileVisitor), 290
- primitive types, 7–10
 - and type parameters, 212
 - attempting to update parameters of, 62
 - comparing, 141
 - converting to strings, 139
 - functions interfaces for, 115
 - passed by value, 63
 - streams of, 265–267
 - wrapper classes for, 40
- printStackTrace method (Throwable), 184
- PrintStream class, 5, 139, 281
 - print method, 5, 28, 187, 280–281
 - printf method, 28–29, 48, 280–281
 - println method, 5–6, 27–28, 43, 110, 280–281
- PrintWriter class, 280
 - close method, 179–180
- priority queues, 242
- private modifier, 2, 76
 - for enum constructors, 149
- Process class, 345–348
 - destroy, destroyForcibly methods, 348
 - exitValue method, 348
 - getErrorStream method, 347
 - getXXXStream methods, 346
 - isAlive method, 348
 - waitFor method, 348
- ProcessBuilder class, 345–348
 - directory method, 346
 - redirectXXX methods, 346–347
 - start method, 347
- processes, 345–348
 - building, 345–347
 - killing, 348
- Processor interface, 372
- Programmer's Day, 383
- programming languages
 - dynamically typed, 432
 - functional, 93
 - object-oriented, 2
 - scripting, 424
- programs
 - compiling, running, 3
 - configuration options for, 238
 - localizing, 397–415
 - responsive, 341
 - testing, 185
- properties, 164–165
 - encoding, 239
 - loading from file, 239
 - names of, 164
 - read-only/write-only, 164
 - testing for, 205
- Properties class, 238–240
- .properties extension, 410
- property files
 - encoding, 412
 - generating, 376

- localizing, 410–412
- PropertyDescriptor class, 165
- protected modifier, 134–135
- Proxy class, 167–168
 - newProxyInstance method, 167
- public modifier, 2, 76
 - and method overriding, 130
 - for interface methods, 95–96
- push method (ArrayDeque), 242
- put method
 - of BlockingQueue, 327
 - of FileChannel, 283
 - of Map, 235–236
 - of Preferences, 415
- putAll method (Map), 236
- putIfAbsent method
 - of ConcurrentHashMap, 326
 - of Map, 236
- putXXX methods (FileChannel), 283
- putXXX methods (Preferences), 415

Q

- \Q, in regular expressions, 295
- Queue interface, 230, 242
 - synchronizing methods in, 335
 - using ArrayDeque with, 242

R

- \r carriage return, 10, 240
- \r, \R, in regular expressions, 294, 297
- race conditions, 268, 319–321
- Random class, 6
 - ints, longs, doubles methods, 267
 - nextInt method, 6, 32
- random numbers, 6, 32, 435
 - streams of, 252, 254, 267
- RandomAccess interface, 230
- RandomAccessFile class, 282–283
 - getFilePointer method, 283
 - length method, 283
 - seek method, 282–283
- RandomNumbers class, 71–72
- range method (EnumSet), 241
- range, rangeClosed methods (XXXStream), 266
- ranges, 244
 - converting to streams, 268

- raw types, 209, 212–213
- read method
 - of InputStream, 275
 - of InputStreamReader, 279
- readAllBytes method (Files), 275, 279
- readAllLines method (Files), 279
- Reader class, 279
- readers, 274
- readExternal method (Externalizable), 304
- readFields method (ObjectInputStream), 307
- readLine function (shell scripts), 440
- readLine method
 - of BufferedReader, 280
 - of Console, 27
- readObject method (ObjectInputStream), 302–304, 306–307
- readPassword method (Console), 27
- readResolve method (SimpleType), 304–306
- readXXX methods (DataInput), 282–283, 304
- receiver parameters, 61, 361
- redirection syntax, 28
- redirectXXX methods (ProcessBuilder), 346–347
- reduce method (Stream), 264–266
- reduceXXX methods (ConcurrentHashMap), 326
- reducing method (Collectors), 264
- reductions, 255, 264–266
- ReentrantLock class, 331–332
 - lock, unlock methods, 332
- reflection, 160–168
 - and generic types, 214, 218–221
 - processing annotations with, 368–371
- ReflectiveOperationException, 152
- regular expressions, 293–301
 - finding matches of, 298
 - flags for, 300–301
 - groups in, 298–299
- relational operators, 18
- relativize method (Path), 285
- remainderUnsigned method (Integer, Long), 16
- remove method
 - of ArrayDeque, 242
 - of ArrayList, 40
 - of BlockingQueue, 327
 - of List, 229
 - of Map, 236
- remove, removeIf methods (Iterator), 233

- remove, removeNode methods (Preferences), 415
 - remove, removeXXX methods (Collection), 228
 - removeIf method (ArrayList), 110
 - @Repeatable annotation, 365, 368
 - REPL (“read-eval-print” loop), 429–430
 - replace method
 - of Map, 236
 - of String, 24
 - replaceAll method
 - of Collections, 231
 - of List, 229
 - of Map, 237
 - of Matcher, 300
 - of String, 300
 - requireNonNull method (Objects), 185
 - resolve, resolveSibling methods (Path), 285
 - @Resource annotation, 364, 366
 - resource bundles, 410–412
 - resource injections, 366
 - ResourceBundle class, 190
 - extending, 412
 - getBundle method, 411–412
 - getObject method, 412
 - getString method, 411
 - resources, 151–160
 - loading, 155
 - managing, 179
 - @Resources annotation, 364
 - resume method (Thread, deprecated), 338
 - retainAll method (Collection), 228
 - @Retention annotation, 362, 365
 - return statement, 32, 47, 60
 - in lambda expressions, 108
 - not in finally, 181
 - @return tag (javadoc), 86
 - return types, covariant, 130, 211
 - return values
 - as arrays, 47
 - missing, 255
 - providing type of, 47
 - reverse method (Collections), 43, 232
 - reversed method (Comparator), 121
 - reverseOrder method (Comparator), 122
 - RFC 822, RFC 1123 formats, 391
 - r1wrap tool, 430
 - rotate method (Collections), 232
 - round method (Math), 17
 - RoundEnvironment interface, 373
 - roundoff errors, 9
 - runAfterXXX methods (CompletableFuture), 344–345
 - Runnable interface, 105–106, 114, 313
 - run method, 114, 312, 337, 339
 - using class literals with, 152
 - runtime
 - raw types at, 212–213
 - safety checks at, 210
 - Runtime class
 - availableProcessors method, 313
 - exec method, 345
 - RuntimeException, 176
- S**
- s formatting symbol (date/time), 393
 - s, S conversion characters, 29
 - \s, \S, in regular expressions, 295
 - safety checks, as runtime, 210
 - @SafeVarargs annotation, 216, 364–365
 - Scala programming language
 - REPL in, 430
 - type parameters in, 207
 - Scanner class, 27
 - hasNext, hasNextXXX, next, nextXXX methods, 27, 279
 - scheduling applications
 - and time zones, 382, 387
 - computing dates for, 385–386
 - ScriptContext interface, 426
 - ScriptEngine interface
 - createBindings method, 425
 - eval method, 425–427
 - ScriptEngineFactory interface, 427
 - ScriptEngineManager class
 - getEngineXXX methods, 424
 - visibility of bindings in, 425
 - scripting engines, 424–425
 - compiling code in, 428
 - implementing Java interfaces in, 427
 - scripting languages, 424
 - invoking functions in, 426
 - searchXXX methods (ConcurrentHashMap), 326
 - security, 77

- @see tag (javadoc), 87–88
- seek method (RandomAccessFile), 282
- serial numbers, 303
- Serializable interface, 301–303
- serialization, 301–307
- serialVersionUID instance variable, 306
- server-side software, 301
- ServiceLoader class, 159–160
 - iterator, load method, 160
- ServletException class, 183
- Set interface, 230, 328
 - working with EnumSet, 241
- set method
 - of Array, 166
 - of ArrayList, 40
 - of BitSet, 240
 - of Field, 163
 - of List, 229
 - of ListIterator, 233
- setAccessible method (AccessibleObject), 162–163
- setContextClassLoader method (Thread), 158
- setCurrency method (NumberFormat), 404
- setDaemon method (Thread), 341
- setDecomposition method (Collator), 407
- setDefault method (Locale), 401–402
- setDefaultUncaughtExceptionHandler method (Thread), 184
- setDoOutput method (URLConnection), 292
- setFilter methods (Handler, Logger), 194
- setFormatter method (Handler), 194
- setLevel method (Logger), 187–188, 191
- setOut method (System), 70
- setReader method (ScriptContext), 426
- setRequestProperty method (URLConnection), 292
- sets, 233–235
 - immutable, 322
 - empty, 244
 - threadsafe, 328
 - unmodifiable views of, 245
- setStrength method (Collator), 407
- setUncaughtExceptionHandler method (Thread), 337
- setUseParentHandlers method (Logger), 191
- setWriter method (ScriptContext), 426
- setXXX methods (Array), 166
- setXXX methods (Field), 162–163
- setXXXAssertionStatus methods (ClassLoader), 187
- shallow copies, 144–146
- shared variables, 318–321
 - atomic mutations of, 329–331
 - locking, 331–332
- shebang, 440
- shell scripts, 437–440
 - command-line arguments in, 439
 - environment variables in, 440
 - executing, 438
 - generating, 376
 - string interpolation in, 438–439
- shell, redirection syntax of, 28
- shift operators, 18–19
- Shift-JIS encoding, 277
- short circuit evaluation, 18
- Short class, 40
 - MIN_VALUE, MAX_VALUE constants, 7
- short indicator, in string templates, 409
- short type, 7–8
 - streams of, 266
 - type conversions of, 17
- short-term persistence, 306
- shuffle method (Collections), 43, 232
- SimpleFileVisitor class, 290
- SimpleJavaFileObject class, 422
- @since tag (javadoc), 87
- singleton, singletonXXX methods (Collections), 232, 245
- singletons, 232, 305
- size method
 - of ArrayList, 40
 - of Collection, 228
 - of Map, 237
- skip method (Stream), 254
- sleep method (Thread), 337, 339
- SocketHandler class, 191
- sort method
 - of Arrays, 43, 104–105, 109–110
 - of Collections, 43, 206–207, 231
 - of List, 229
- sorted maps, 244
 - immutable empty, 244
 - unmodifiable views of, 245

- sorted method (*Stream*), 254–255
- sorted sets, 230, 244
 - immutable empty, 244
 - traversing, 234
 - unmodifiable views of, 245
- sorted streams, 268
- SortedMap* interface, 244
- SortedSet* interface, 230, 234
 - first, last methods, 234
 - headSet*, *subSet*, *tailSet* methods, 234, 244
- sorting
 - array lists, 43
 - arrays, 43, 103–104
 - chaining comparators for, 121
 - changing order of, 120
 - streams, 254–255
 - strings, 22–23, 110, 406–408
- source code, generating, 373–376
- source files
 - encoding of, 413
 - reading from memory, 421
- space flag (for output), 30
- spaces
 - in regular expressions, 295
 - removing, 24
- split* method
 - of *Pattern*, 299
 - of *String*, 21, 300
- splitAsStream* method (*Pattern*), 252, 299
- splitIterator* method (*Collection*), 229
- sqrt* method (*Math*), 15
- square root, computing, 258
- Stack* class, 242
- stack trace, 184–185
- StackTraceElement* class, 184
- standard output, 3
- StandardCharsets* class, 278
- StandardJavaFileManager* interface, 421–422, 424
- start* method
 - of *ProcessBuilder*, 347
 - of *Thread*, 337
- startsWith* method (*String*), 24
- stateless operations, 267
- statements, combining, 37
- static constants, 69–70
- static imports, 78–79
- static initialization, 157
- static methods, 47, 71–72
 - accessing static variables from, 72
 - importing, 78
 - in interfaces, 99–100
- static modifier, 2, 12, 47, 68–72, 150
- static nested classes, 79–80
- static variables, 68–69
 - accessing from static methods, 72
 - importing, 78
 - visibility of, 318
- stop*, *suspend* methods (*Thread*, deprecated), 338
- Stream* interface
 - collect* method, 259–260, 266
 - concat* method, 254
 - count* method, 251, 255
 - distinct* method, 254, 268
 - empty* method, 252
 - filter* method, 251–253, 255
 - findAny* method, 256
 - findFirst* method, 255
 - flatMap* method, 253
 - forEach*, *forEachOrdered* methods, 259
 - generate* method, 252, 266
 - iterate* method, 252, 255, 259, 266
 - limit* method, 254, 268
 - map* method, 253
 - mapToInt* method, 265
 - max*, *min* methods, 255
 - of method, 251–252
 - peek* method, 255
 - reduce* method, 264–266
 - skip* method, 254
 - sorted* method, 254–255
 - toArray* method, 112, 259
 - unordered* method, 268
 - xxxMatch* methods, 256
- stream* method
 - of *Arrays*, 252, 266
 - of *BitSet*, 241
 - of *Collection*, 229, 250–251
- streams, 249–269
 - collecting elements of, 259–261
 - computing values from, 264–266

- converting to/from arrays, 252, 259, 268, 324
 - creating, 251–252
 - debugging, 255
 - empty, 252, 255, 264–265
 - flattening, 253
 - infinite, 251–252, 254–255
 - intermediate operations for, 251
 - noninterference of, 269
 - of primitive type values, 265–267
 - of random numbers, 267
 - ordered, 268
 - parallel, 250, 256, 259, 261–262, 265, 267–269, 323
 - processed lazily, 251, 255, 269
 - reductions of, 255
 - removing duplicates from, 254
 - sorting, 254–255
 - splitting/combining, 254
 - terminal operation for, 251, 255
 - transformations of, 252–254, 267
 - vs. collections, 251
- strictfp modifier, 15
- StrictMath class, 16
- String class, 6, 24
- charAt, codePoints, codePointXXX methods, 26
 - compareTo method, 22–23, 103, 406
 - compareToIgnoreCase method, 110
 - contains, endsWith, startsWith methods, 24
 - equals method, 21–22
 - equalsIgnoreCase method, 22
 - final, 133
 - hash codes, 143
 - immutable, 24
 - indexOf, lastIndexOf methods, 24
 - join method, 20
 - length method, 6, 26
 - offsetByCodePoints method, 26
 - replace method, 24
 - replaceAll method, 300
 - split method, 21, 300
 - substring method, 21
 - toLowerCase method, 24, 253
 - toUpperCase method, 24
 - trim method, 24, 403
 - string interpolation, in scripts, 438–440
- StringBuilder class, 21
- strings, 6, 20–26
- comparing, 21–23
 - concatenating, 20–21, 139
 - converting:
 - from objects, 138–140
 - to numbers, 23
 - empty, 22–23, 139
 - formatting for output, 29
 - from byte arrays, 278
 - normalized, 408
 - sorting, 22–23, 110, 406–408
 - splitting, 21, 252
 - templates for, 408–410
 - transforming to lowercase, 253
 - traversing, 26
- StringSource class, 421
- StringWriter class, 281
- strong element (HTML), in documentation
- comments, 85
- subclasses, 128–129
- anonymous, 135–136, 150
 - calling toString method in, 139
 - constructors for, 131
 - initializing instance variables in, 131
 - methods in, 129
 - preventing, 133
 - public, 130
 - superclass assignments in, 131
- subList method (List), 229, 244
- subMap method (SortedMap), 244
- subpath method (Path), 286
- subSet method
- of NavigableSet, 235
 - of SortedSet, 234, 244
- substring method (String), 21
- subtractExact method (Math), 16
- subtraction, 14
- accurate, 19
 - not associative, 265
- subtypes, 97
- wildcards for, 204
- sum method
- of LongAdder, 330
 - of XXXStream, 267

- summarizingXXX methods (Collectors), 260, 264
 - summaryStatistics method (XXXStream), 267
 - summingXXX methods (Collectors), 263
 - super keyword, 102, 129–131, 137, 205–207
 - superclasses, 128–129
 - annotating, 359
 - calling equals method, 141
 - default methods of, 136–137
 - in JavaScript, 436
 - methods of, 129–130
 - public, 130
 - supertypes, 97–99
 - wildcards for, 205–206
 - Supplier interface, 114
 - @SuppressWarnings annotation, 32, 212, 364–365, 367
 - swap method (Collections), 231
 - Swing GUI toolkit, 107, 342
 - SwingConstants interface, 99
 - SwingWorker class (Swing), 342
 - switch statement, 31–32
 - using enumerations in, 151
 - symbolic links, 288–289
 - synchronized keyword, 332–336
 - synchronized views, 246
 - synchronizedXXX methods (Collections), 232
 - System class
 - getProperties method, 239
 - setOut method, 70
 - system class loader, 156, 158
 - system classes, enabling/disabling
 - assertions for, 186
 - system properties, 239–240
 - System.err constant, 184, 191, 340, 420
 - System.in constant, 27
 - System.out constant, 5–6, 12, 27–29, 43, 48, 70, 110, 187, 280, 420
 - systemXXX methods (Preferences), 414
- T**
- T, in dates, 390
 - t, T conversion characters, 29
 - \t
 - for character literals, 10
 - in regular expressions, 294
 - %t pattern variable, 193
 - tab, 10
 - tagging interfaces, 145
 - tailMap method (SortedMap), 244
 - tailSet method
 - of NavigableSet, 235
 - of SortedSet, 234, 244
 - take method (BlockingQueue), 327
 - @Target annotation, 362–363, 365
 - Task class (JavaFX), 342
 - tasks, 312–316
 - cancelling, 315–316
 - combining results from, 314–316
 - computationally intensive, 313
 - coordinating work between, 326–328
 - defining, 105
 - executing, 106, 313
 - groups of, 340
 - long-running, 341–342
 - running, 312–314
 - short-lived, 313
 - submitting, 315
 - vs. threads, 313
 - Temporal interface, 385
 - TemporalAdjusters class, 385
 - terminal window, 3–4
 - test method
 - of BiPredicate, 114
 - of Predicate, 114, 205
 - of XXXPredicate, 115
 - @Test annotation, 356–357, 361–362
 - text
 - input, 279–280
 - output, 280–281
 - TextStyle enumeration, 406
 - thenAccept, thenAcceptBoth, thenCombine methods (CompletableFuture), 344
 - thenApply, thenApplyAsync methods (CompletableFuture), 343–344
 - thenComparing method (Comparator), 121
 - thenCompose method (CompletableFuture), 343–344
 - thenRun method (CompletableFuture), 344
 - this reference, 61–62
 - annotating, 361
 - capturing, 111
 - in constructors, 322–323

- in lambda expressions, 117
- this syntax, with constructors, 65
- Thread class
 - get/setContextClassLoader methods, 158
 - interrupted method, 338
 - isInterrupted method, 315, 338
 - join method, 337
 - properties, 340–341
 - resume, stop, suspend methods (deprecated), 338
 - setDaemon method, 341
 - setDefaultUncaughtExceptionHandler method, 184
 - setUncaughtExceptionHandler method, 337
 - sleep method, 337, 339
 - start method, 337
- ThreadLocal class, 339–340
 - get, withInitial methods, 340
- threads, 312, 337–341
 - and visibility, 317–319, 334
 - atomic mutations in, 329–331
 - creating, 106
 - groups of, 340
 - interrupting, 315, 338–339
 - local variables in, 339–340
 - locking, 331–332
 - priorities of, 340
 - race conditions in, 268, 319–321
 - running tasks in, 105
 - starting, 337–338
 - states of, 340
 - temporarily inactive, 338
 - terminating, 313–314
 - vs. tasks, 313
 - waiting on conditions, 335
 - worker, 341–342
- throw statement, 175
- Throwable class, 175
 - getStackTrace, printStackTrace methods, 184
 - in assertions, 186
 - initCause method, 183
 - no generic subtypes for, 217
- throwing method (Logger), 189–190
- throws keyword, 177
 - type variables in, 217–218
- @throws tag (javadoc), 86, 178
- time
 - current, 380
 - formatting, 390–393, 404–406
 - measuring, 381
 - parsing, 393
- Time class, 393–394
- time indicator, in string templates, 409
- time zones, 387–390
- Timestamp class, 142, 393–394
- timestamps, 391
 - using instants as, 381
- TimeZone class, 394
- ™ (trademark symbol), 408
- toAbsolutePath method (Path), 285
- toArray method
 - of Collection, 229
 - of Stream, 112, 259
 - of XXXStream, 267
- toByteArray method
 - of BitSet, 241
 - of ByteArrayOutputStream, 274–275
- toCollection method (Collectors), 259
- toConcurrentMap method (Collectors), 261
- toFile method (Path), 286
- toFormat method (DateTimeFormatter), 392
- toGenericString method (Class), 153
- toInstant method
 - of Date, 393
 - of ZonedDateTime, 387
- toIntExact method (Math), 17
- toList method (Collectors), 259
- toLowerCase method (String), 24, 253
- toMap method (Collectors), 260–261
- ToolProvider class, 420
- toPath method (File), 286
- toSet method (Collectors), 259, 263
- toString method
 - calling from subclasses, 139
 - of Arrays, 43, 140
 - of BitSet, 241
 - of Class, 153
 - of Double, Integer, 23
 - of Enum, 148
 - of Modifier, 155
 - of Object, 138–140
 - of Point, 138–140

- toUnsignedInt method (Byte), 8
- toUpperCase method (String), 24
- toXXX methods (Duration), 381
- ToXXXBiFunction interfaces, 115
- ToXXXFunction interfaces, 115, 212
- toXXXOfDay methods
 - of LocalTime, 386
 - of ZonedDateTime, 389
- toZonedDateTime method (GregorianCalendar), 393–394
- transient modifier, 303
- TreeMap class, 235, 261
- TreeSet class, 234
- trim method (String), 24, 403
- true value (boolean), 10
- try statement, 178–182
 - for visiting directories, 288
- tryLock method (FileChannel), 284
- try-with-resources statement, 179–181
 - closing output streams with, 276
 - for file locking, 284
- type bounds, 202–203, 221
 - annotating, 360
- type erasure, 208–211, 216
 - clashes after, 216–217
- Type interface, 220
- type parameters, 103, 200–201
 - and primitive types, 201, 212
 - annotating, 358
- type variables
 - and exceptions, 217–218
 - in static context, 216
 - no instantiating of, 213–215
 - wildcards with, 206–207
- TypeElement interface, 373
- TypeVariable interface, 220

U

- \u
 - for character literals, 9–10, 412–413
 - in regular expressions, 294
- %u pattern variable, 193
- UnaryOperator interface, 114
- uncaught exception handlers, 337, 340

- unchecked exceptions, 176
 - and generic types, 218
 - documenting, 178
- UncheckedIOException, 279
- Unicode, 25–26, 266, 276
 - escapes in, 239
 - normalization forms in, 408
 - replacement character in, 281
- unit tests, 355
- Unix operating system
 - bash scripts, 437
 - path separator, 75, 240
 - specifying locales in, 402
 - wildcard in classpath in, 75
- unlock method (ReentrantLock), 332
- unmodifiableXXX methods (Collections), 232
- unordered method (Stream), 268
- until method (LocalDate), 383–384
- updateAndGet method (AtomicXXX), 329
- URL class
 - final, 133
 - openConnection method, 292
 - openStream method, 274
- URLClassLoader class, 156
- URLConnection class, 292
- URLs, reading from, 274, 292
- user directory, 285
- user interface. *See* GUI
- user preferences, 413–415
- user.dir, user.home, user.name system properties, 239
- userXXX methods (Preferences), 414
- UTC (coordinated universal time), 388
- UTF-8 encoding, 276–277
 - for source files, 413
 - modified, 282
- UTF-16 encoding, 9, 26, 266, 277
 - in regular expressions, 294
- Util class, 158

V

- V formatting symbol (date/time), 393
- \v, \V, in regular expressions, 295

- valueOf method
 - of BitSet, 241
 - of Enum, 147–148
- values method
 - of Enum, 148
 - of Map, 237, 244
- varargs parameters
 - corrupted, 365
 - declaring, 48
- variable comments, 86–87
- VariableElement interface, 373
- variables, 6, 10–13
 - atomic mutations of, 329–331
 - declaring, 10–11
 - defined in interfaces, 99
 - deprecated, 87
 - effectively final, 118–119
 - final, 322
 - holding object references, 56–58
 - in lambda expressions, 117–119
 - initializing, 10–12
 - local, 36–37
 - names of, 11
 - parameter, 62
 - private, 59, 76
 - public static final, 99
 - redefining, 37
 - scope of, 36, 76
 - shared, 318–321, 331–332
 - static final. *See* constants
 - static, 68–69, 72, 78
 - thread-local, 339–340
 - using an abstract class as type of, 134
 - visibility of, 317–319, 334
 - volatile, 318–319
- @version tag (javadoc), 85, 89
- versioning, 306
- views, 244–246
 - checked, 245
 - synchronized, 246
 - unmodifiable, 245
- virtual machine, 4
- visibility, 317–319
 - guaranteed with locks, 334

- visitFileXXX methods (FileVisitor), 290
- void keyword, 2, 47
 - using class literals with, 152
- volatile modifier, 318–319

W

- \w, \W, in regular expressions, 295
- wait method (Object), 138, 335–337
- waitFor method (Process), 348
- waiting on a condition, 336
- walk method (Files), 288–291
- walkFileTree method (Files), 288, 290
- warning method (Logger), 188
- warnings
 - for switch statements, 151
 - suppressing, 212, 216, 365
- weak references, 243
- weaker access privilege, 130
- WeakHashMap class, 243
- weakly consistent iterators, 325
- WeakReference class, 243
- web pages
 - extracting links from, 342
 - reading, 341, 343
- whenComplete method (CompletableFuture), 344
- while statement, 32–34
 - breaking, 34
 - continuing, 35
 - declaring variables for, 36
- white space
 - in regular expressions, 295
 - removing, 24
- wildcards
 - annotating, 360
 - capturing, 208
 - for annotation processors, 372
 - for types, 204–206
 - in class path, 75
 - unbounded, 207
 - with imported classes, 77–78
 - with type variables, 206–207
- WildcardType interface, 220
- Window class, 76
- WindowAdapter class, 100
- WindowListener interface, 100

- with method (Temporal), 385
- withInitial method (ThreadLocal), 340
- withLocale method (DateTimeFormatter), 392, 405
- withXXX methods
 - of LocalDate, 383
 - of LocalTime, 386
 - of ZonedDateTime, 389
- words
 - in regular expressions, 295
 - reading from a file, 279
 - sorting alphabetically, 406–408
- working directory, 346
- wrapper classes, 40
- write method
 - of Files, 281, 287
 - of OutputStream, 276
- writeExternal method (Externalizable), 304
- writeObject method (ObjectOutputStream), 302–304
- Writer class, 280–281
 - write method, 280
- writeReplace method, 304–306
- writers, 274
- writeXXX methods (DataOutput), 282–283, 304

X

- x formatting symbol (date/time), 393
- x, X conversion characters, 29
- \x, in regular expressions, 294
- XML descriptors, generating, 376
- xor method (BitSet), 241

Y

- y formatting symbol (date/time), 392
- Year, YearMonth classes, 384

Z

- z, Z formatting symbols (date/time), 390, 393
- \z, \Z, in regular expressions, 297
- ZIP file systems, 74, 291
- ZipInputStream, ZipOutputStream classes, 292
- zoned time, 382–384, 387–390
- ZonedDateTime class, 387–390
 - and legacy classes, 394
 - getXXX, isXXX, minus, minusXXX, now, ofInstant, toXXXOfDay, withXXX methods, 389
 - of method, 387–389
 - parse method, 405
 - plus, plusXXX methods, 388–389
- ZoneId class, 387