



Core Java

for the

Impatient

Fourth Edition

Cay S. Horstmann



FREE SAMPLE CHAPTER



This page intentionally left blank

Core Java for the Impatient

Fourth Edition

Cay S. Horstmann

◆ Addison-Wesley

Hoboken, New Jersey

This page intentionally left blank

Cover illustration by Morphart Creation / Shutterstock

Figures 1.1, 1.3: Microsoft Corporation

Figure 1.2: Eclipse Foundation

Figures 1.4, 1.5, 1.11: Oracle Corporation

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

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

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

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

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

Please contact us with concerns about any potential bias at pearson.com/report-bias.html.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2024947133

Copyright © 2025 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit pearson.com/global-permission-granting.html.

ISBN-13: 978-0-13-540454-6

ISBN-10: 0-13-540454-1

This page intentionally left blank

To Chi—the most patient person in my life.

This page intentionally left blank

Table of Contents

Preface.....	xvii
Acknowledgments.....	xix
1. Fundamental Programming Structures.....	1
1.1. Our First Program.....	1
1.1.1. Dissecting the “Hello, World” Program.....	1
1.1.2. Compiling and Running a Java Program	3
1.1.3. Object Instances and Method Calls.....	6
1.1.4. JShell	8
1.2. Primitive Types	12
1.2.1. Signed Integer Types	12
1.2.2. Floating-Point Types.....	13
1.2.3. The <code>char</code> Type	14
1.2.4. The <code>boolean</code> Type.....	15
1.3. Variables	15
1.3.1. Variable Declarations	15
1.3.2. Identifiers	16
1.3.3. Initialization	16
1.3.4. Constants	17
1.4. Arithmetic Operations.....	18
1.4.1. Assignment.....	19
1.4.2. Basic Arithmetic	19
1.4.3. Mathematical Methods.....	21
1.4.4. Number Type Conversions	21
1.4.5. Relational and Logical Operators.....	23
1.4.6. Big Numbers	24
1.5. Strings.....	25
1.5.1. Concatenation	25
1.5.2. Substrings	26
1.5.3. String Comparison	27
1.5.4. Converting Between Numbers and Strings.....	28
1.5.5. The String API.....	29
1.5.6. Code Points and Code Units	30
1.5.7. Text Blocks	33
1.6. Input and Output	35
1.6.1. Reading Input.....	35
1.6.2. Formatted Output.....	37
1.7. Control Flow	39
1.7.1. Branches	39
1.7.2. Switches	40
1.7.3. Loops	43
1.7.4. Breaking and Continuing	44
1.7.5. Local Variable Scope	46
1.8. Arrays and Array Lists	48

1.8.1. Working with Arrays.....	48
1.8.2. Array Construction.....	49
1.8.3. Array Lists	50
1.8.4. Wrapper Classes for Primitive Types	51
1.8.5. The Enhanced for Loop	52
1.8.6. Copying Arrays and Array Lists	52
1.8.7. Array Algorithms	53
1.8.8. Command-Line Arguments.....	54
1.8.9. Multidimensional Arrays	55
1.9. Functional Decomposition	58
1.9.1. Declaring and Calling Static Methods	58
1.9.2. Array Parameters and Return Values	58
1.9.3. Variable Arguments	59
1.10. Exercises	60
2. Object-Oriented Programming	63
2.1. Working with Objects.....	63
2.1.1. Accessor and Mutator Methods	66
2.1.2. Object References	66
2.2. Implementing Classes.....	68
2.2.1. Instance Variables	68
2.2.2. Method Headers.....	69
2.2.3. Method Bodies	69
2.2.4. Instance Method Invocations	70
2.2.5. The this Reference	71
2.2.6. Call by Value.....	71
2.3. Object Construction	73
2.3.1. Implementing Constructors	73
2.3.2. Overloading	74
2.3.3. Calling One Constructor from Another	75
2.3.4. Default Initialization.....	75
2.3.5. Instance Variable Initialization.....	76
2.3.6. Final Instance Variables	77
2.3.7. The Constructor with No Arguments	77
2.4. Records	78
2.4.1. The Record Concept.....	79
2.4.2. Constructors: Canonical, Custom, and Compact	81
2.5. Static Variables and Methods	82
2.5.1. Static Variables	82
2.5.2. Static Constants	83
2.5.3. Static Initialization Blocks	84
2.5.4. Static Methods	84
2.5.5. Factory Methods.....	85
2.6. Packages	86
2.6.1. Package Declarations	86
2.6.2. The jar Command.....	87
2.6.3. The Class Path.....	88
2.6.4. Package Access	90
2.6.5. Importing Classes.....	91

2.6.6. Static Imports.....	92
2.7. Nested Classes.....	92
2.7.1. Static Nested Classes.....	93
2.7.2. Inner Classes.....	94
2.7.3. Special Syntax Rules for Inner Classes.....	97
2.8. Documentation Comments.....	98
2.8.1. Comment Insertion.....	98
2.8.2. Class Comments	99
2.8.3. Method Comments	99
2.8.4. Variable Comments	100
2.8.5. General Comments	100
2.8.6. Links	101
2.8.7. Package, Module, and Overview Comments	102
2.8.8. Comment Extraction	102
2.9. Exercises.....	103
3. Interfaces and Lambda Expressions.....	105
3.1. Interfaces	106
3.1.1. Using Interfaces	106
3.1.2. Declaring an Interface	106
3.1.3. Implementing an Interface.....	107
3.1.4. Converting to an Interface Type	109
3.1.5. Casts and the instanceof Operator	109
3.1.6. The “Pattern-Matching” Form of instanceof	110
3.1.7. Extending Interfaces	112
3.1.8. Implementing Multiple Interfaces.....	112
3.1.9. Constants	112
3.2. Static, Default, and Private Methods.....	113
3.2.1. Static Methods	113
3.2.2. Default Methods	114
3.2.3. Resolving Default Method Conflicts	115
3.2.4. Private Methods	116
3.3. Examples of Interfaces	117
3.3.1. The Comparable Interface	117
3.3.2. The Comparator Interface	118
3.3.3. The Runnable Interface	119
3.3.4. User Interface Callbacks.....	120
3.4. Lambda Expressions	121
3.4.1. The Syntax of Lambda Expressions	121
3.4.2. Functional Interfaces	123
3.5. Method and Constructor References	124
3.5.1. Method References	124
3.5.2. Constructor References	125
3.6. Processing Lambda Expressions.....	126
3.6.1. Implementing Deferred Execution	126
3.6.2. Choosing a Functional Interface	127
3.6.3. Implementing Your Own Functional Interfaces	130
3.7. Lambda Expressions and Variable Scope	131
3.7.1. Scope of a Lambda Expression	131

3.7.2. Accessing Variables from the Enclosing Scope.....	132
3.8. Higher-Order Functions.....	134
3.8.1. Methods That Return Functions	134
3.8.2. Methods That Modify Functions	135
3.8.3. Comparator Methods.....	135
3.9. Local and Anonymous Classes	136
3.9.1. Local Classes	136
3.9.2. Anonymous Classes	137
3.10. Exercises	138
4. Inheritance and Reflection	141
4.1. Extending a Class	142
4.1.1. Super- and Subclasses	142
4.1.2. Defining and Inheriting Subclass Methods	142
4.1.3. Method Overriding	143
4.1.4. Subclass Construction.....	144
4.1.5. Superclass Assignments.....	145
4.1.6. Casts.....	146
4.1.7. Anonymous Subclasses	146
4.1.8. Method Expressions with <code>super</code>	147
4.2. Inheritance Hierarchies	147
4.2.1. Final Methods and Classes.....	148
4.2.2. Abstract Methods and Classes	148
4.2.3. Protected Access	149
4.2.4. Sealed Types	150
4.2.5. Inheritance and Default Methods	154
4.3. <code>Object</code> : The Cosmic Superclass	154
4.3.1. The <code>toString</code> Method.....	155
4.3.2. The <code>equals</code> Method.....	157
4.3.3. The <code>hashCode</code> Method.....	159
4.3.4. Cloning Objects	161
4.4. Enumerations	164
4.4.1. Methods of Enumerations	164
4.4.2. Constructors, Methods, and Fields	165
4.4.3. Bodies of Instances	166
4.4.4. Static Members	167
4.4.5. Switching on an Enumeration	167
4.5. Pattern Matching	168
4.5.1. Record Patterns.....	169
4.5.2. Guards	170
4.5.3. Null Handling	170
4.5.4. Exhaustiveness	171
4.5.5. Dominance.....	172
4.5.6. Patterns and Constants	173
4.6. Runtime Type Information and Resources	174
4.6.1. The <code>Class</code> Class	174
4.6.2. Loading Resources	178
4.6.3. Class Loaders	178
4.6.4. The Context Class Loader	180

4.6.5. Service Loaders.....	181
4.7. Reflection.....	183
4.7.1. Enumerating Class Members.....	183
4.7.2. Inspecting Objects.....	184
4.7.3. Invoking Methods.....	185
4.7.4. Constructing Objects.....	186
4.7.5. JavaBeans	187
4.7.6. Working with Arrays.....	188
4.7.7. Proxies.....	190
4.8. Exercises.....	191
5. Exceptions, Assertions, and Logging.....	193
5.1. Exception Handling.....	193
5.1.1. Throwing Exceptions.....	194
5.1.2. The Exception Hierarchy	194
5.1.3. Declaring Checked Exceptions.....	196
5.1.4. Catching Exceptions.....	197
5.1.5. The Try-with-Resources Statement.....	198
5.1.6. The finally Clause.....	200
5.1.7. Rethrowing and Chaining Exceptions	202
5.1.8. Uncaught Exceptions and the Stack Trace	203
5.1.9. API Methods for Throwing Exceptions.....	204
5.2. Assertions	205
5.2.1. Using Assertions	206
5.2.2. Enabling and Disabling Assertions	206
5.3. Logging	207
5.3.1. Should You Use the Java Logging Framework?	207
5.3.2. Logging 101	208
5.3.3. The Platform Logging API	209
5.3.4. Logging Configuration	211
5.3.5. Log Handlers	213
5.3.6. Filters and Formatters	215
5.4. Exercises.....	216
6. Generic Programming	219
6.1. Generic Classes.....	220
6.2. Generic Methods.....	220
6.3. Type Bounds	221
6.4. Type Variance and Wildcards.....	222
6.4.1. Subtype Wildcards	223
6.4.2. Supertype Wildcards	224
6.4.3. Wildcards with Type Variables	226
6.4.4. Unbounded Wildcards	227
6.4.5. Wildcard Capture	227
6.5. Generics in the Java Virtual Machine	228
6.5.1. Type Erasure	228
6.5.2. Cast Insertion.....	229
6.5.3. Bridge Methods	230
6.6. Restrictions on Generics	231
6.6.1. No Primitive Type Arguments	231

6.6.2. At Runtime, All Types Are Raw	232
6.6.3. You Cannot Instantiate Type Variables	233
6.6.4. You Cannot Construct Arrays of Parameterized Types	235
6.6.5. Class Type Variables Are Not Valid in Static Contexts.....	236
6.6.6. Methods May Not Clash after Erasure	236
6.6.7. Exceptions and Generics.....	237
6.7. Reflection and Generics	238
6.7.1. The <code>Class<T></code> Class	238
6.7.2. Generic Type Information in the Virtual Machine	239
6.8. Exercises	241
7. Collections	245
7.1. An Overview of the Collections Framework	245
7.2. Iterators	251
7.3. Sets	252
7.4. Maps	254
7.4.1. Basic Map Operations	254
7.4.2. Entries and Traversal.....	258
7.5. Other Collections	260
7.5.1. Properties	260
7.5.2. Bit Sets	262
7.5.3. Small Collections.....	264
7.5.4. Enumeration Sets and Maps	265
7.5.5. Stacks, Queues, Deques, and Priority Queues	265
7.5.6. Weak Hash Maps	267
7.6. Views.....	267
7.6.1. Ranges.....	267
7.6.2. Unmodifiable Views.....	268
7.6.3. Reversed Views	269
7.7. Exercises	269
8. Streams.....	271
8.1. From Iterating to Stream Operations	271
8.2. Stream Creation.....	273
8.3. The <code>filter</code> , <code>map</code> , and <code>flatMap</code> Methods	276
8.4. Extracting Substreams and Combining Streams	278
8.5. Other Stream Transformations	279
8.6. Simple Reductions	280
8.7. The <code>Optional</code> Type.....	281
8.7.1. Producing an Alternative	281
8.7.2. Consuming the Value If Present	281
8.7.3. Pipelining <code>Optional</code> Values	282
8.7.4. How Not to Work with <code>Optional</code> Values	283
8.7.5. Creating <code>Optional</code> Values	284
8.7.6. Composing <code>Optional</code> Value Functions with <code>flatMap</code>	284
8.7.7. Turning an <code>Optional</code> into a Stream	285
8.8. Collecting Results	286
8.9. Collecting into Maps	287
8.10. Grouping and Partitioning	289
8.11. Downstream Collectors.....	290

8.12. Reduction Operations	292
8.13. Primitive Type Streams.....	294
8.14. Parallel Streams.....	295
8.15. Exercises.....	298
9. Processing Input and Output.....	301
9.1. Input/Output Streams, Readers, and Writers	301
9.1.1. Obtaining Streams	302
9.1.2. Reading Bytes	302
9.1.3. Writing Bytes.....	303
9.1.4. Character Encodings.....	304
9.1.5. Text Input	306
9.1.6. Text Output.....	307
9.1.7. Reading Character Input.....	309
9.1.8. Reading and Writing Binary Data	310
9.1.9. Random-Access Files.....	311
9.1.10. Memory-Mapped Files	311
9.1.11. File Locking.....	312
9.2. Paths, Files, and Directories	313
9.2.1. Paths.....	313
9.2.2. Creating Files and Directories	315
9.2.3. Copying, Moving, and Deleting Files	315
9.2.4. Visiting Directory Entries.....	317
9.2.5. ZIP File Systems.....	320
9.3. HTTP Connections	321
9.3.1. The URLConnection and HttpURLConnection Classes	321
9.3.2. The HTTP Client API	322
9.4. Regular Expressions	325
9.4.1. The Regular Expression Syntax	325
9.4.2. Testing a Match.....	330
9.4.3. Finding All Matches	331
9.4.4. Groups.....	332
9.4.5. Splitting along Delimiters	333
9.4.6. Replacing Matches	334
9.4.7. Flags.....	335
9.5. Serialization	336
9.5.1. The Serializable Interface	336
9.5.2. Transient Instance Variables.....	338
9.5.3. The readObject and writeObject Methods	338
9.5.4. The readExternal and writeExternal Methods	340
9.5.5. The readResolve and writeReplace Methods	341
9.5.6. Versioning.....	342
9.5.7. Deserialization and Security	344
9.6. Exercises	346
10. Concurrent Programming.....	349
10.1. Concurrent Tasks	350
10.1.1. Running Tasks	350
10.1.2. Futures	353
10.1.3. Thread Interruption	355

10.2. Thread Safety.....	357
10.2.1. Visibility.....	357
10.2.2. Race Conditions	359
10.2.3. Strategies for Safe Concurrency	361
10.2.4. Immutable Classes	362
10.3. Threadsafe Data Structures.....	363
10.3.1. Concurrent Hash Maps	363
10.3.2. Blocking Queues.....	365
10.3.3. Other Threadsafe Data Structures.....	367
10.4. Parallel Algorithms	368
10.4.1. Parallel Streams	368
10.4.2. Parallel Array Operations.....	368
10.5. Asynchronous Computations	369
10.5.1. CompletableFuture.....	369
10.5.2. Composing CompletableFuture	371
10.5.3. Long-Running Tasks in User Interface Callbacks	375
10.6. Atomic Counters and Accumulators	376
10.7. Locks and Conditions.....	379
10.7.1. Locks	379
10.7.2. The synchronized Keyword.....	380
10.7.3. Waiting on Conditions	382
10.8. Threads	384
10.8.1. Running a Thread.....	385
10.8.2. Thread-Local Variables.....	386
10.8.3. Miscellaneous Thread Properties.....	387
10.9. Processes	388
10.9.1. Building a Process	388
10.9.2. Running a Process.....	390
10.9.3. Process Handles	392
10.10. Exercises	393
11. Annotations.....	399
11.1. Using Annotations.....	400
11.1.1. Annotation Elements	400
11.1.2. Multiple and Repeated Annotations	401
11.1.3. Annotating Declarations.....	402
11.1.4. Annotating Type Uses	403
11.1.5. Making Receivers Explicit.....	404
11.2. Defining Annotations	405
11.3. Annotations in the Java API	408
11.3.1. Annotations for Compilation	409
11.3.2. Meta-Annotations	410
11.4. Processing Annotations at Runtime	412
11.5. Source-Level Annotation Processing	415
11.5.1. Annotation Processors.....	416
11.5.2. The Language Model API	416
11.5.3. Using Annotations to Generate Source Code.....	417
11.6. Exercises.....	420
12. The Java Platform Module System.....	423

12.1. The Module Concept	424
12.2. Naming Modules	425
12.3. The Modular “Hello, World!” Program	426
12.4. Requiring Modules	427
12.5. Exporting Packages	429
12.6. Modules and Reflective Access	433
12.7. Modular JARs	436
12.8. Automatic Modules	437
12.9. The Unnamed Module	438
12.10. Command-Line Flags for Migration	439
12.11. Transitive and Static Requirements	440
12.12. Qualified Exporting and Opening	442
12.13. Service Loading	443
12.14. Tools for Working with Modules	444
12.15. Exercises	446
Index	449

This page intentionally left blank

Preface

Java has seen many changes since its initial release in 1996. 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 over 2,000 pages. However, if you just want to be productive with modern Java, there is a much faster, easier pathway for learning the language and core libraries. In this book, I don't retrace history and don't dwell on features of past versions. I show you the good parts of Java as it exists today, so you can put your knowledge to work quickly.

As with my previous "Impatient" books, I quickly cut to the chase, showing you what you need to know to solve 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 Python, C++, JavaScript, Swift, 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 today, including the powerful concepts of lambda expressions and streams, as well as modern constructs such as records and pattern matching.

This book is fully updated to Java 21. It uses modern features and does not dwell on historical or obsolete constructs. Preview features that may make it to the language in the future are not covered either.

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.

Traditionally, books on Java have focused on user interface programming, but nowadays, few developers produce user interfaces on desktop computers. You will be able to use this book effectively without being distracted by lengthy 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/bugs.html> and leave a comment.



Tip: Download the runnable code examples that complement this book at <http://horstmann.com/javaimpatient/bugs.html>.

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. My special gratitude goes to the excellent team of reviewers for this and previous editions who spotted many errors and gave thoughtful suggestions for improvement. They are: Andres Almiray, Gail Anderson, Paul Anderson, Marcus Biel, Jean-Claude Brantschen, Brian Goetz, Mark Lawrence, Doug Lea, Ron Mak, Simon Ritter, Yoshiki Shibata, Clovis Tondo, and Christian Ullenboom.

I wrote the book using HTML and CSS, and Prince (<https://princexml.com>) turned it into PDF—a workflow that I highly recommend.

*Cay Horstmann
Düsseldorf, Germany
August 2024*

This page intentionally left blank

Processing Input and Output

In this chapter, you will learn how to work with files, directories, and web pages, and how to read and write data in binary and text format. You will also find a discussion of regular expressions, which can be useful for processing input. (I couldn't think of a better place to handle that topic, and apparently neither could the Java developers—when the regular expression API specification was proposed, it was attached to the specification request for “new I/O” features.) Finally, this chapter shows you the object serialization mechanism that lets you store objects as easily as you can store text or numeric data.

The key points of this chapter are:

1. An `InputStream` is a source of bytes, and an `OutputStream` is a destination for bytes.
2. A `Reader` reads characters, and a `Writer` writes them. Be sure to specify a character encoding.
3. The `Files` class has convenience methods for reading all bytes or lines of a file.
4. The `DataInput` and `DataOutput` interfaces have methods for writing numbers in binary format.
5. Use a `RandomAccessFile` or a memory-mapped file for random access.
6. A `Path` is an absolute or relative sequence of path components in a file system. `Paths` can be combined (or “resolved”).
7. Use the methods of the `Files` class to copy, move, or delete files and to recursively walk through a directory tree.
8. To read or update a ZIP file, use a ZIP file system.
9. You can read the contents of a web page with the `URL` class. To read metadata or write data, use the `URLConnection` class.
10. With the `Pattern` and `Matcher` classes, you can find all matches of a regular expression in a string, as well as the captured groups for each match.
11. The serialization mechanism can save and restore any object implementing the `Serializable` interface, provided its instance variables are also serializable.

9.1. Input/Output Streams, Readers, and Writers

In the Java API, a source from which one can read bytes is called an *input stream*. The bytes can come from a file, a network connection, or an array in memory. (These streams are unrelated to the streams of [Chapter 8](#).) Similarly, a destination for bytes is an *output stream*. In contrast, *readers* and *writers* consume and produce sequences of *characters*. In the following sections, you will learn how to read and write bytes and characters.

9.1.1. Obtaining Streams

The easiest way to obtain a stream from a file is with the static methods

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
```

Here, `path` is an instance of the `Path` class that is covered in [Section 9.2.1](#). It describes a path in a file system.

If you have an `URL` object, you can read its contents from the input stream returned by the `openStream` method. (The `URL` constructors are deprecated, and you should create an `URL` instance as shown here.)

```
var url = URI.create("https://horstmann.com/index.html").toURL();
InputStream in = url.openStream();
```

[Section 9.3](#) shows how to send data to a web server.

The `ByteArrayInputStream` class lets you read from an array of bytes.

```
byte[] bytes = ...;
var in = new ByteArrayInputStream(bytes);
Read from in
```

Conversely, to send output to a byte array, use a `ByteArrayOutputStream`:

```
var out = new ByteArrayOutputStream();
Write to out
byte[] bytes = out.toByteArray();
```

9.1.2. Reading Bytes

The `InputStream` class has a method to read a single byte:

```
InputStream in = ...;
int b = in.read();
```

This method either returns the byte as an integer between 0 and 255, or returns -1 if the end of input has been reached.



Caution: The Java byte type has values between -128 and 127. You can cast the returned value into a byte *after* you have checked that it is not -1.

More commonly, you will want to read the bytes in bulk. The most convenient method is the `readAllBytes` method that simply reads all bytes from the stream into a byte array:

```
byte[] bytes = in.readAllBytes();
```



Tip: If you want to read all bytes from a file, call the convenience method

```
byte[] bytes = Files.readAllBytes(path);
```

If you want to read some, but not all bytes, provide a byte array and call the `readNBytes` method:

```
var bytes = new byte[len];
int bytesRead = in.readNBytes(bytes, offset, n);
```

The method reads until either `n` bytes are read or no further input is available, and returns the actual number of bytes read. If no input was available at all, the methods return `-1`.



Note: There is also a `read(byte[], int, int)` method whose description seems exactly like `readNBytes`. The difference is that the `read` method only attempts to read the bytes and returns immediately with a lower count if it fails. The `readNBytes` method keeps calling `read` until all requested bytes have been obtained or `read` returns `-1`.

Finally, you can skip bytes:

```
long bytesToSkip = ...;
in.skipNBytes(bytesToSkip);
```

9.1.3. Writing Bytes

The `write` methods of an `OutputStream` can write individual bytes and byte arrays.

```
OutputStream out = ...;
int b = ...;
out.write(b);
byte[] bytes = ...;
out.write(bytes);
out.write(bytes, start, length);
```

When you are done writing a stream, you must *close* it in order to commit any buffered output. This is best done with a `try-with-resources` statement:

```
try (OutputStream out = ...) {
    out.write(bytes);
}
```

If you need to copy an input stream to an output stream, use the `InputStream.transferTo` method:

```
try (InputStream in = ...; OutputStream out = ...) {
    in.transferTo(out);
}
```

Both streams need to be closed after the call to `transferTo`. It is best to use a `try-with-resources` statement, as in the code example.

To write a file to an `OutputStream`, call

```
Files.copy(path, out);
```

Conversely, to save an `InputStream` to a file, call

```
Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
```

9.1.4. Character Encodings

Input and output streams are for sequences of bytes, but in many cases you will work with text—that is, sequences of characters. It then matters how characters are encoded into bytes.

Java uses the Unicode standard for characters. Each character or “code point” has a 21-bit integer number. There are different *character encodings*—methods for packaging those 21-bit numbers into bytes.

The most common encoding is UTF-8, which encodes each Unicode code point into a sequence of one to four bytes (see [Table 9.1](#)). UTF-8 has the advantage that the characters of the traditional ASCII character set, which contains all characters used in English, only take up one byte each.

Table 9.1: UTF-8 Encoding

Character range	Encoding
0...7F	0a6a5a4a3a2a1a0
80...7FF	110a10a9a8a7a6 10a5a4a3a2a1a0
800...FFFF	1110a15a14a13a12 10a11a10a9a8a7a6 10a5a4a3a2a1a0
10000...10FFFF	11110a20a19a18 10a17a16a15a14a13a12 10a11a10a9a8a7a6 10a5a4a3a2a1a0

A less common encoding is UTF-16, which encodes each Unicode code point into one or two 16-bit values (see [Table 9.2](#)). This is the encoding used in Java strings. Actually, there are two forms of UTF-16, called “big-endian” and “little-endian.” Consider the 16-bit value `0x2122`. In big-endian format, the more significant byte comes first: `0x21` followed by `0x22`. In little-endian format, it is the other way around: `0x22 0x21`. To indicate which of the two is used, a file can start with the “byte order mark,” the 16-bit quantity `0xFEFF`. A reader can use this value to determine the byte order and discard it.

Table 9.2: UTF-16 Encoding

Character range	Encoding
<code>0...FFFF</code>	<code>a15a14a13a12a11a10a9a8a7a6a5a4a3a2a1a0</code>
<code>10000...10FFFF</code>	<code>110110b19b18b17b16a15a14a13a12a11a10 110111a9a8a7a6a5a4a3a2a1a0</code> where $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$



Caution: Some programs, including Microsoft Notepad, add a byte order mark at the beginning of UTF-8 encoded files. Clearly, this is unnecessary since there are no byte ordering issues in UTF-8. But the Unicode standard allows it, and even suggests that it’s a pretty good idea since it leaves little doubt about the encoding. It is supposed to be removed when reading a UTF-8 encoded file. Sadly, Java does not do that, and bug reports against this issue are closed as “will not fix.” Your best bet is to strip out any leading `\uFEFF` that you find in your input.

In addition to the UTF encodings, there are partial encodings that cover a character range suitable for a given user population. For example, ISO 8859-1 is a one-byte code that includes accented characters used in Western European languages. Shift_JIS is a variable-length code for Japanese characters. A large number of these encodings are still in widespread use.

Because UTF-8 is so common, it has become the default encoding since Java 18. Previously, the default encoding was the *native encoding*—the character encoding that is preferred by the operating system of the computer running your program. On Windows, that is generally not UTF-8. If you are using an older version of Java, or if you are working with text in an encoding other than UTF-8, you need to explicitly specify the encoding.



Note: The native encoding is returned by the static method `Charset.defaultCharset`. The static method `Charset.availableCharsets` returns all available `Charset` instances, as a map from canonical names to `Charset` objects.

The `StandardCharsets` class has static variables of type `Charset` for the character encodings that every Java virtual machine must support:

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

To obtain the Charset for another encoding, use the static `forName` method:

```
Charset shiftJIS = Charset.forName("Shift_JIS");
```

You use the Charset object to specify a character encoding. For example, you can turn an array of bytes into a string as

```
var contents = new String(bytes, StandardCharsets.ISO_8859_1);
```

9.1.5. Text Input

To read text input, use a Reader. You can obtain a Reader from any input stream with the `InputStreamReader` adapter:

```
InputStream inStream = ...;
var in = new InputStreamReader(inStream, charset);
```

If you want to process the input one UTF-16 code unit at a time, you can call the `read` method:

```
int ch = in.read();
```

The method returns a code unit between 0 and 65536, or -1 at the end of input.

That is not very convenient. Here are several alternatives.

With a short text file, you can read it into a string like this:

```
String content = Files.readString(path, charset);
```

But if you want the file as a sequence of lines, call

```
List<String> lines = Files.readAllLines(path, charset);
```

If the file is large, process them lazily as a `Stream<String>`:

```
try (Stream<String> lines = Files.lines(path, charset)) {
    ...
}
```



Note: If an `IOException` occurs as the stream fetches the lines, that exception is wrapped into an `UncheckedIOException` which is thrown out of the stream operation. This subterfuge is necessary because stream operations are not declared to throw any checked exceptions.

To read numbers or words from a file, use a `Scanner`, as you have seen in [Chapter 1](#). For example,

```
var in = new Scanner(path);
while (in.hasNextDouble()) {
    double value = in.nextDouble();
    ...
}
```



Tip: To read alphabetic words, set the scanner's delimiter to a regular expression that is the complement of what you want to accept as a token. For example, after calling

```
in.useDelimiter("\\PL+");
```

the scanner reads in letters since any sequence of nonletters is a delimiter. See [Section 9.4.1](#) for the regular expression syntax.

You can then obtain a stream of all words as

```
Stream<String> words = in.tokens();
```

If your input does not come from a file, wrap the `InputStream` into a `BufferedReader`:

```
try (var reader = new BufferedReader(new InputStreamReader(url.openStream()))) {
    Stream<String> lines = reader.lines();
    ...
}
```

A `BufferedReader` reads input in chunks for efficiency. (Oddly, this is not an option for basic readers.) It has methods `readLine` to read a single line and `lines` to yield a stream of lines.

If a method asks for a `Reader` and you want it to read from a file, call `Files.newBufferedReader(path, charset)`.

9.1.6. Text Output

To write text, use a `Writer`. With the `write` method, you can write strings. You can turn any output stream into a `Writer`:

```
OutputStream outStream = ...;
var out = new OutputStreamWriter(outStream, charset);
out.write(str);
```

To get a writer for a file, use

```
Writer out = Files.newBufferedWriter(path, charset);
```

It is more convenient to use a `PrintWriter`, which has the `print`, `println`, and `printf` that you have always used with `System.out`. Using those methods, you can print numbers and use formatted output.

If you write to a file, construct a `PrintWriter` like this:

```
var out = new PrintWriter(Files.newBufferedWriter(path, charset));
```

If you write to another stream, use

```
var out = new PrintWriter(new OutputStreamWriter(outStream, charset));
```



Note: `System.out` is an instance of `PrintStream`, not `PrintWriter`. This is a relic from the earliest days of Java. However, the `print`, `println`, and `printf` methods work the same way for the `PrintStream` and `PrintWriter` classes, using a character encoding for turning characters into bytes.

If you already have the text to write in a string, call

```
String content = ...;
Files.writeString(path, content, charset);
```

or

```
Files.write(path, lines, charset);
```

Here, `lines` can be a `Collection<String>`, or even more generally, an `Iterable<? extends CharSequence>`.

To append to a file, use

```
Files.writeString(path, charset, StandardOpenOption.APPEND);
Files.write(path, lines, charset, StandardOpenOption.APPEND);
```



Caution: When writing text with a partial character set such as ISO 8859-1, any unmappable characters are silently changed to a “replacement”—in most cases, either the ? character or the Unicode replacement character U+FFFD.

Sometimes, a library method wants a Writer to write output. If you want to capture that output in a string, hand it a StringWriter. Or, if it wants a PrintWriter, wrap the StringWriter like this:

```
var writer = new StringWriter();
throwable.printStackTrace(new PrintWriter(writer));
String stackTrace = writer.toString();
```

9.1.7. Reading Character Input

If you read a file with a structured format such as JSON or XML, you will use a parser that someone wrote who understands the fiddly details of that format. Such a parser typically reads a character at a time.

In the uncommon case that you need to write such a parser, use a BufferedReader for efficiency. Keep calling its read method, which yields a char value or -1 at the end of input. The reader converts the encoding of the input stream into UTF-16.

If you want to process Unicode code points, you need to handle the UTF-16 encoding. Here is how to read one code point:

```
int ch = reader.read();
if (ch != -1)
{
    int codePoint;
    if (Character.isHighSurrogate((char) ch))
    {
        int ch2 = reader.read();
        if (Character.isLowSurrogate((char) ch2))
            codePoint = Character.toCodePoint(ch, ch2);
        else
            throw new MalformedInputException();
    }
    else
        codePoint = ch;
}
```

The Character class contains methods to tell whether a particular code point has a given property. For example,

```
Character.isLetter(codePoint)
```

returns true if codePoint is a letter in some language. Here are some other classification methods:

```
isUpperCase
isLowerCase
isDigit
isSpaceChar
isEmoji
```

These methods use the rules of the Unicode standard. Others refer to the rules of the Java language:

```
isJavaIdentifierStart
isJavaIdentifierPart
isWhitespace
```

After analyzing the code points, you often need to store them in strings, converting them back to UTF-16. The `appendCodePoint` method of the `StringBuilder` class turns a code point into one or two `char` values which are appended to the builder.

9.1.8. Reading and Writing Binary Data

The `DataInput` interface declares the following methods for reading a number, a character, a `boolean` value, or a string in binary format:

```
byte readByte()
int readUnsignedByte()
char readChar()
short readShort()
int readUnsignedShort()
int readInt()
long readLong()
float readFloat()
double readDouble()
void readFully(byte[] b)
```

The `DataOutput` interface declares corresponding `write` methods.



Note: These methods read and write numbers in big-endian format.



Caution: There are also `readUTF/writeUTF` methods that use a “modified UTF-8” format. These methods are *not* compatible with regular UTF-8, and are only useful for JVM internals.

The advantage of binary I/O is that it is fixed width and efficient. For example, `writeInt` always writes an integer as a big-endian 4-byte binary quantity regardless of the number of digits. The space needed is the same for each value of a given type, which speeds up random access. Also, reading binary data is faster than parsing text. The main drawback is that the resulting files cannot be easily inspected in a text editor.

You can use the `DataInputStream` and `DataOutputStream` adapters with any stream. For example,

```
DataInput in = new DataInputStream(Files.newInputStream(path));
DataOutput out = new DataOutputStream(Files.newOutputStream(path));
```

9.1.9. Random-Access Files

The `RandomAccessFile` class lets you read or write data anywhere in a file. You can open a random-access file either for reading only or for both reading and writing; specify the option by using the string "r" (for read access) or "rw" (for read/write access) as the second argument in the constructor. For example,

```
var file = new RandomAccessFile(path.toString(), "rw");
```

A random-access file has a *file pointer* that indicates the position of the next byte to be read or written. The `seek` method sets the file pointer to an arbitrary byte position within the file. The argument to `seek` is a long integer between zero and the length of the file (which you can obtain with the `length` method). The `getFilePointer` method returns the current position of the file pointer.

The `RandomAccessFile` class implements both the `DataInput` and `DataOutput` interfaces. To read and write numbers from a random-access file, use methods such as `readInt/writeInt` that you saw in the preceding section. For example,

```
int value = file.readInt();
file.seek(file.getFilePointer() - 4);
file.writeInt(value + 1);
```

9.1.10. Memory-Mapped Files

Memory-mapped files provide another, very efficient approach for random access that works well for very large files. However, the API for data access is completely different from that of input/output streams. First, get a *channel* to the file:

```
FileChannel channel = FileChannel.open(path,
    StandardOpenOption.READ, StandardOpenOption.WRITE)
```

Then, map an area of the file (or, if it is not too large, the entire file) into memory:

```
ByteBuffer buffer = channel.map(FileChannel.MapMode.READ_WRITE,  
    0, channel.size());
```

Use methods `get`, `getInt`, `getDouble`, and so on to read values, and the equivalent `put` methods to write values.

```
int offset = ...;  
int value = buffer.getInt(offset);  
buffer.put(offset, value + 1);
```

At some point, and certainly when the channel is closed, these changes are written back to the file.



Note: By default, the methods for reading and writing numbers use big-endian byte order. You can change the byte order with the command

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

9.1.11. File Locking

When multiple simultaneously executing programs modify the same file, they need to communicate in some way, or the file can easily become damaged. File locks can solve this problem.

Suppose your application saves a configuration file with user preferences. If a user invokes two instances of the application, it could happen that both of them want to write the configuration file at the same time. In that situation, the first instance should lock the file. When the second instance finds the file locked, it can decide to wait until the file is unlocked or simply skip the writing process. To lock a file, call either the `lock` or `tryLock` methods of the `FileChannel` class.

```
FileChannel channel = FileChannel.open(path, StandardOpenOption.WRITE);  
FileLock lock = channel.lock();
```

or

```
FileLock lock = channel.tryLock();
```

The first call blocks until the lock becomes available. The second call returns immediately, either with the lock or with `null` if the lock is not available. The file remains locked until the lock or the channel is closed. It is best to use a try-with-resources statement:

```
try (FileLock lock = channel.lock()) {  
    ...  
}
```

9.2. Paths, Files, and Directories

You have already seen `Path` objects for specifying file paths. In the following sections, you will see how to manipulate these objects and how to work with files and directories.

9.2.1. Paths

A `Path` is a sequence of directory names, optionally followed by a file name. The first component of a path may be a root component, such as `/` or `C:\`. The permissible root components depend on the file system. A path that starts with a root component is *absolute*. Otherwise, it is *relative*. For example, here we construct an absolute and a relative path. For the absolute path, we assume we are running on a Unix-like file system.

```
Path absolute = Path.of("/", "home", "cay");
Path relative = Path.of("myapp", "conf", "user.properties");
```

The static `Path.of` method receives one or more strings, which it joins with the path separator of the default file system (`/` for a Unix-like file system, `\` for Windows). It then parses the result, throwing an `InvalidPathException` if the result is not a valid path in the given file system. The result is a `Path` object.

You can also provide a string with separators to the `Path.of` method:

```
Path homeDirectory = Path.of("/home/cay");
```



Note: A `Path` object does not have to correspond to a file that actually exists. It is merely an abstract sequence of names. To create a file, first make a path, then call a method to create the corresponding file—see [Section 9.2.2](#).

It is very common to combine or “resolve” paths. The call `p.resolve(q)` returns a path according to these rules:

- If `q` is absolute, then the result is `q`.
- if `q` does not have a root, then the result is obtained by joining `p` and `q`.
- Otherwise, the result depends on the rules of the file system.

For example, suppose your application needs to find its configuration file relative to the home directory. Here is how you can combine the paths:

```
Path workPath = homeDirectory.resolve("myapp/work");
// Same as homeDirectory.resolve(Path.of("myapp/work"));
```

There is a convenience method `resolveSibling` that resolves against a path’s parent, yielding a sibling path. For example, if `workPath` is `/home/cay/myapp/work`, the call

```
Path tempPath = workPath.resolveSibling("temp");  
yields /home/cay/myapp/temp.
```

The opposite of `resolve` is `relativize`. The call `p.relativize(r)` yields the path `q` which, when resolved with `p`, yields `r`. For example,

```
Path.of("/home/cay").relativize(Path.of("/home/fred/myapp"))
```

yields `..../fred/myapp`, assuming we have a file system that uses `..` to denote the parent directory.

The `normalize` method removes any redundant `.` and `..` components (or whatever the file system may deem redundant). For example, normalizing the path `/home/cay/..../fred/./myapp` yields `/home/fred/myapp`.

The `toAbsolutePath` method yields the absolute path of a given path. If the path is not already absolute, it is resolved against the *working directory*—that is, the directory of the process in which the JVM was invoked. For example, if you launched a Java program from `/home/cay/myapp`, then `Path.of("config").toAbsolutePath()` returns `/home/cay/myapp/config`.



Note: You can obtain the working directory by a call to `System.getProperty("user.dir")`.

The `Path` interface has methods for taking paths apart and combining them with other paths. This code sample shows some of the most useful ones:

```
Path p = Path.of("/home", "cay", "myapp.properties");  
Path parent = p.getParent(); // The path /home/cay  
Path file = p.getFileName(); // The last element, myapp.properties  
Path root = p.getRoot(); // The initial segment / (null for a relative path)  
Path first = p.getName(0); // The first element  
Path dir = p.subpath(1, p.getNameCount());  
// All but the first element, cay/myapp.properties
```

The `Path` interface extends the `Iterable<Path>` element, so you can iterate over the name components of a `Path` with an enhanced `for` loop:

```
for (Path component : path) {  
    ...  
}
```



Note: Occasionally, you may need to interoperate with legacy APIs that use the `File` class instead of the `Path` interface. The `Path` interface has a `toFile` method, and the `File` class has a `toPath` method.

9.2.2. Creating Files and Directories

To create a new directory, call

```
Files.createDirectory(path);
```

All but the last component in the path must already exist. To create intermediate directories as well, use

```
Files.createDirectories(path);
```

You can create an empty file with

```
Files.createFile(path);
```

The call throws an exception if the file already exists. The checks for existence and the creation are atomic. If the file doesn't exist, it is created before anyone else has a chance to do the same.

The call `Files.exists(path)` checks whether the given file or directory exists. To test whether it is a directory or a “regular” file (that is, with data in it, not something like a directory or symbolic link), call the static methods `isDirectory` and `isRegularFile` of the `Files` class.

There are convenience methods for creating a temporary file or directory in a given or system-specific location.

```
Path tempFile = Files.createTempFile(dir, prefix, suffix);
Path tempFile = Files.createTempFile(prefix, suffix);
Path tempDir = Files.createTempDirectory(dir, prefix);
Path tempDir = Files.createTempDirectory(prefix);
```

Here, `dir` is a `Path`, and `prefix/suffix` are strings which may be null. For example, the call `Files.createTempFile(null, ".txt")` might return a path such as `/tmp/1234405522364837194.txt`.

9.2.3. Copying, Moving, and Deleting Files

To copy a file from one location to another, simply call

```
Files.copy(fromPath, toPath);
```

To move the file instead, call

```
Files.move(fromPath, toPath);
```

You can also use this command to move an empty directory.

The copy or move will fail if the target exists. If you want to overwrite an existing target, use the `REPLACE_EXISTING` option. If you want to copy all file attributes, use the `COPY_ATTRIBUTES` option. You can supply both like this:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,
          StandardCopyOption.COPY_ATTRIBUTES);
```

You can specify that a move should be atomic. Then you are assured that either the move completed successfully, or the source continues to be present. Use the `ATOMIC_MOVE` option:

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

See [Table 9.3](#) for a summary of the options that are available for file operations.

Finally, to delete a file, simply call

```
Files.delete(path);
```

This method throws an exception if the file doesn't exist, so instead you may want to use

```
boolean deleted = Files.deleteIfExists(path);
```

The deletion methods can also be used to remove an empty directory.

Table 9.3: Standard Options for File Operations

Option	Description
<code>StandardOpenOption; use with newBufferedWriter, newInputStream, newOutputStream, write</code>	
READ	Open for reading.
WRITE	Open for writing.
APPEND	If opened for writing, append to the end of the file.
TRUNCATE_EXISTING	If opened for writing, remove existing contents.
CREATE_NEW	Create a new file and fail if it exists.
CREATE	Atomically create a new file if it doesn't exist.
DELETE_ON_CLOSE	Make a "best effort" to delete the file when it is closed.

Option	Description
SPARSE	A hint to the file system that this file will be sparse.
DSYNC SYNC	Requires that each update to the file data data and metadata be written synchronously to the storage device.
StandardCopyOption; use with copy, move	
ATOMIC_MOVE	Move the file atomically.
COPY_ATTRIBUTES	Copy the file attributes.
REPLACE_EXISTING	Replace the target if it exists.
LinkOption; use with all of the above methods and exists, isDirectory, isRegularFile	
NOFOLLOW_LINKS	Do not follow symbolic links.
FileVisitOption; use with find, walk, walkFileTree	
FOLLOW_LINKS	Follow symbolic links.

9.2.4. Visiting Directory Entries

The static `Files.list` method returns a `Stream<Path>` that reads the entries of a directory. The directory is read lazily, making it possible to efficiently process directories with huge numbers of entries.

Since reading a directory involves a system resource that needs to be closed, you should use a `try-with-resources` block:

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {
    ...
}
```

The `list` method does not enter subdirectories. To process all descendants of a directory, use the `Files.walk` method instead.

```
try (Stream<Path> entries = Files.walk(pathToRoot)) {
    // Contains all descendants, visited in depth-first order
}
```

Here is a sample traversal of the unzipped `src.zip` tree:

```
java
java.nio
java.nio/DirectCharBufferU.java
java.nio/ByteBufferAsShortBufferRL.java
java.nio/MappedByteBuffer.java
...
java.nio/ByteBufferAsDoubleBufferB.java
java.nio/charset
java.nio/charset/CoderMalfunctionError.java
java.nio/charset/CharsetDecoder.java
java.nio/charset/UnsupportedCharsetException.java
java.nio/charset/spi
java.nio/charset/spi/CharsetProvider.java
java.nio/charset/StandardCharsets.java
java.nio/charset/Charset.java
...
java.nio/charset/CoderResult.java
java.nio/HeapFloatBufferR.java
...
```

As you can see, whenever the traversal yields a directory, it is entered before continuing with its siblings.

You can limit the depth of the tree that you want to visit by calling `Files.walk(pathToRoot, depth)`. Both `walk` methods have a varargs parameter of type `FileVisitOption...`, but there is only one option you can supply: `FOLLOW_LINKS` to follow symbolic links.



Note: If you filter the paths returned by `walk` and your filter criterion involves the file attributes stored with a directory, such as size, creation time, or type (file, directory, symbolic link), then use the `find` method instead of `walk`. Call that method with a predicate function that accepts a path and a `BasicFileAttributes` object. The only advantage is efficiency. Since the directory is being read anyway, the attributes are readily available.

This code fragment uses the `Files.walk` method to copy one directory to another:

```
Files.walk(source).forEach(p -> {
    try {
        Path q = target.resolve(source.relativize(p));
        if (Files.isDirectory(p))
            Files.createDirectory(q);
        else
            Files.copy(p, q);
    } catch (IOException ex) {
```

```
        throw new UncheckedIOException(ex);
    }
});
```

Unfortunately, you cannot easily use the `Files.walk` method to delete a tree of directories since you need to first visit the children before deleting the parent. In that case, use the `walkFileTree` method. It requires an instance of the `FileVisitor` interface. Here is when the file visitor gets notified:

1. Before a directory is processed:

```
FileVisitResult preVisitDirectory(T dir, IOException ex)
```

2. When a file is encountered:

```
FileVisitResult visitFile(T path, BasicFileAttributes attrs)
```

3. When an exception occurs in the `visitFile` method:

```
FileVisitResult visitFileFailed(T path, IOException ex)
```

4. After a directory is processed:

```
FileVisitResult postVisitDirectory(T dir, IOException ex)
```

In each case, the notification method returns one of the following results:

- Continue visiting the next file: `FileVisitResult.CONTINUE`
- Continue the walk, but without visiting the entries in this directory: `FileVisitResult.SKIP_SUBTREE`
- Continue the walk, but without visiting the siblings of this file: `FileVisitResult.SKIP_SIBLINGS`
- Terminate the walk: `FileVisitResult.TERMINATE`

If any of the methods throws an exception, the walk is also terminated, and that exception is thrown from the `walkFileTree` method.

The `SimpleFileVisitor` class implements this interface, continuing the iteration at each point and rethrowing any exceptions.

Here is how you can delete a directory tree:

```
Files.walkFileTree(root, new SimpleFileVisitor<Path>() {
    public FileVisitResult visitFile(Path file,
        BasicFileAttributes attrs) throws IOException {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }
})
```

```

public FileVisitResult postVisitDirectory(Path dir,
    IOException ex) throws IOException {
    if (ex != null) throw ex;
    Files.delete(dir);
    return FileVisitResult.CONTINUE;
}
});

```



Caution: The `Files.walk` method throws an exception if any of the subdirectories are not readable. If you only want to visit readable directories, use the `walkFileTree` method.

9.2.5. ZIP File Systems

The `Paths` class looks up paths in the default file system—the files on the user's local disk. You can have other file systems. One of the more useful ones is a ZIP file system. If `zipname` is the name of a ZIP file, then the call

```
FileSystem zipfs = FileSystems.newFileSystem(Path.of(zipname));
```

establishes a file system that contains all files in the ZIP archive. It's an easy matter to copy a file out of that archive if you know its name:

```
Files.copy(zipfs.getPath(sourceName), targetPath);
```

Here, `zipfs.getPath` is the analog of `Path.of` for an arbitrary file system.

To list all files in a ZIP archive, walk the file tree:

```

Files.walk(zipfs.getPath("/")).forEach(p -> {
    Process p
});

```

You have to work a bit harder to create a new ZIP file. Here is the magic incantation:

```

Path zipPath = Path.of("myfile.zip");
var uri = URI.create("jar:" + zipPath.toUri());
// Constructs the URI jar:file://myfile.zip
try (FileSystem zipfs = FileSystems.newFileSystem(uri,
    Collections.singletonMap("create", "true"))) {
    // To add files, copy them into the ZIP file system
    Files.copy(sourcePath, zipfs.getPath("/").resolve(targetPath));
}

```



Note: There is an older API for working with ZIP archives, with classes `ZipInputStream` and `ZipOutputStream`, but it's not as easy to use as the one described in this section.

9.3. HTTP Connections

You can read from a URL by using the input stream returned from `URL.getInputStream` method. However, if you want additional information about a web resource, or if you want to write data, you need more control over the process than the `URL` class provides. The `URLConnection` class was designed before HTTP was the universal protocol of the Web. It provides support for a number of protocols, but its HTTP support is somewhat cumbersome. When the decision was made to support HTTP/2, it became clear that it would be best to provide a modern client interface instead of reworking the existing API. The `HttpClient` provides a more convenient API and HTTP/2 support.

In the following sections, I provide a cookbook for using the `HttpURLConnection` class, and then give an overview of the API.

9.3.1. The `URLConnection` and `HttpURLConnection` Classes

To use the `URLConnection` class, follow these steps:

1. Get an `URLConnection` object:

```
URLConnection connection = url.openConnection();
```

For an HTTP URL, the returned object is actually an instance of `HttpURLConnection`.

2. If desired, set request properties:

```
connection.setRequestProperty("Accept-Charset", "UTF-8, ISO-8859-1");
```

If a key has multiple values, separate them by commas.

3. To send data to the server, call

```
connection.setDoOutput(true);
try (OutputStream out = connection.getOutputStream()) {
    // Write to out
}
```

4. If you want to read the response headers and you haven't called `getOutputStream`, call

```
connection.connect();
```

Then query the header information:

```
Map<String, List<String>> headers = connection.getHeaderFields();
```

For each key, you get a list of values since there may be multiple header fields with the same key.

5. Read the response:

```
try (InputStream in = connection.getInputStream()) {
    // Read from in
}
```

A common use case is to post form data. The `URLConnection` class automatically sets the content type to `application/x-www-form-urlencoded` when writing data to a HTTP URL, but you need to encode the name/value pairs:

```
URL url = ...;
URLConnection connection = url.openConnection();
connection.setDoOutput(true);
try (var out = new OutputStreamWriter(
        connection.getOutputStream())) {
    Map<String, String> postData = ...;
    boolean first = true;
    for (Map.Entry<String, String> entry : postData.entrySet()) {
        if (first) first = false;
        else out.write("&");
        out.write(URLEncoder.encode(entry.getKey(), "UTF-8"));
        out.write("=");
        out.write(URLEncoder.encode(entry.getValue(), "UTF-8"));
    }
}
try (InputStream in = connection.getInputStream()) {
    ...
}
```

9.3.2. The HTTP Client API

The HTTP client API provides another mechanism for connecting to a web server which is simpler than the `URLConnection` class with its rather fussy set of stages. More importantly, the implementation supports HTTP/2.

An `HttpClient` can issue requests and receive responses. You get a client by calling

```
HttpClient client = HttpClient.newHttpClient();
```

Alternatively, if you need to configure the client, use a builder API like this:

```
HttpClient client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();
```

That is, you get a builder, call methods to customize the item that is going to be built, and then call the `build` method to finalize the building process. This is a common pattern for constructing immutable objects.

Follow the same pattern for formulating requests. Here is a GET request:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://horstmann.com"))
    .GET()
    .build();
```

The URI is the “uniform resource identifier” which is, when using HTTP, the same as a URL. However, in Java, the `URL` class has methods for actually opening a connection to a URL, whereas the `URI` class is only concerned with the syntax (scheme, host, port, path, query, fragment, and so on).

When sending the request, you have to tell the client how to handle the response. If you just want the body as a string, send the request with a `HttpResponse.BodyHandlers.ofString()`, like this:

```
HttpResponse<String> response
    = client.send(request, HttpResponse.BodyHandlers.ofString());
```

The `HttpResponse` class is a template whose type denotes the type of the body. You get the response body string simply as

```
String bodyString = response.body();
```

There are other response body handlers that get the response as a byte array or a file. One can hope that eventually the JDK will support JSON and provide a JSON handler.

With a POST request, you similarly need a “body publisher” that turns the request data into the data that is being posted. There are body publishers for strings, byte arrays, and files. Again, one can hope that the library designers will wake up to the reality that most POST requests involve form data, file uploads, or JSON objects, and provide appropriate publishers.

Nowadays, the most common POST request body contains JSON, which you need to convert to a string. Then you can form the following request:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create(urlString))
    .header("Content-Type", "application/json")
    .POST(HttpRequest.BodyPublishers.ofString(jsonString))
    .build();
```

The book’s companion code has examples for posting form data and file uploads.

The `HttpRequest.Builder` class also has build methods for the less common `PUT`, `DELETE`, and `HEAD` requests.

Java 16 adds a builder for filtering the headers of an existing `HttpRequest`. You provide the request and a function that receives the header names and values, returning true for those that should be retained. For example, here we modify the content type:

```
HttpRequest request2 = HttpRequest.newBuilder(request,
    (name, value) -> !name.equalsIgnoreCase("Content-Type")) // Remove old content type
    .header("Content-Type", "application/xml") // Add new content type
    .build();
```

The `HttpResponse` object also yields the status code and the response headers.

```
int status = response.statusCode();
HttpHeaders responseHeaders = response.headers();
```

You can turn the `HttpHeaders` object into a map:

```
Map<String, List<String>> headerMap = responseHeaders.map();
```

The map values are lists since in HTTP, each key can have multiple values.

If you just want the value of a particular key, and you know that there won't be multiple values, call the `firstValue` method:

```
Optional<String> lastModified = headerMap.firstValue("Last-Modified");
```

You get the response value or an empty optional if none was supplied.

The `HttpClient` is autocloseable, so you can declare it in a `try-with-resources` statement. Its `close` method waits for the completion of submitted requests and then closes its connection pool.



Tip: To enable logging for the `HttpClient`, add this line to `net.properties` in your JDK:

```
jdk.httpclient.HttpClient.log=all
```

Instead of `all`, you can specify a comma-separated list of headers, requests, content, errors, ssl, trace, and frames, optionally followed by `:control`, `:data`, `:window`, or `:all`. Don't use any spaces.

Then set the logging level for the logger named `jdk.httpclient.HttpClient` to `INFO`, for example by adding this line to the `logging.properties` file in your JDK:

```
jdk.httpclient.HttpClient.level=INFO
```

9.4. Regular Expressions

Regular expressions specify string patterns. Use them whenever you need to locate strings that match a particular pattern. For example, suppose you want to find hyperlinks in an HTML file. You need to look for strings of the pattern ``. But wait—there may be extra spaces, or the URL may be enclosed in single quotes. Regular expressions give you a precise syntax for specifying what sequences of characters are legal matches.

In the following sections, you will see the regular expression syntax used by the Java API, and how to put regular expressions to work.

9.4.1. The Regular Expression Syntax

In a regular expression, a character denotes itself unless it is one of the reserved characters

`. * + ? { | () [\ ^ $`

For example, the regular expression `Java` only matches the string `Java`.

The symbol `.` matches any single character. For example, `.a.a` matches `Java` and `data`.

The `*` symbol indicates that the preceding constructs may be repeated 0 or more times; for `a +`, it is 1 or more times. A suffix of `?` indicates that a construct is optional (0 or 1 times). For example, `be+s?` matches `be`, `bee`, and `bees`. You can specify other multiplicities with `{ }` (see [Table 9.4](#)).

A `|` denotes an alternative: `.(oo|ee)f` matches `beef` or `woof`. Note the parentheses—without them, `.oo|eef` would be the alternative between `.oo` and `eef`. Parentheses are also used for grouping—see [Section 9.4.4](#).

A *character class* is a set of character alternatives enclosed in brackets, such as `[Jj]`, `[0-9]`, `[A-Za-z]`, or `[^0-9]`. Inside a character class, the `-` denotes a range (all characters whose Unicode values fall between the two bounds). However, a `-` that is the first or last character in a character class denotes itself. A `^` as the first character in a character class denotes the complement (all characters except those specified).

[Table 9.4](#) contains a number of *predefined character classes* such as `\d` (digits). There are many more with the `\p` prefix, such as `\p{Sc}` (Unicode currency symbols)—see [Table 9.5](#).

The characters `^` and `$` match the beginning and end of input.

If you need to have a literal `. * + ? { | () [\ ^ $`, precede it by a backslash. Inside a character class, you only need to escape `[` and `\`, provided you are careful about the positions of `] - ^`. For example, `[]^-]` is a class containing all three of them.



Caution: If the regular expression is in a string literal, each backslash needs to be escaped with another backslash. If you forget that second backslash, you usually get an error because sequences such as `\$` or `\.` are not valid in string literals. But if you want to match a word boundary and accidentally use `\b` instead of `\\\b`, then you have a problem: `\b` is a valid escape sequence, indicating a backspace.

Instead of using backslashes, you can surround a string with `\Q` and `\E`. For example, `\($0\.99\)` and `\Q($0.99)\E` both match the string `($0.99)`.



Tip: If you have a string that may contain some of the many special characters in the regular expression syntax, you can escape them all by calling `Pattern.quote(str)`. This simply surrounds the string with `\Q` and `\E`, but it takes care of the special case where `str` may contain `\E`.

Table 9.4: Regular Expression Syntax

Expression	Description	Example
Characters		
<code>c, not one of . * + ? { () [\ ^ \$</code>	The character <code>c</code> .	<code>J</code>
<code>.</code>	Any character except line terminators, or any character if the DOTALL flag is set.	
<code>\X</code>	Any Unicode “extended grapheme cluster”, which is perceived as a character or symbol	
<code>\x{p}</code>	The Unicode code point with hex code <code>p</code> .	<code>\x{1D546}</code>
<code>\uhhhh, \xhh, \0o, \ooo, \0ooo</code>	The UTF-16 code unit with the given hex or octal value.	<code>\uFEFF</code>
<code>\a, \e, \f, \n, \r, \t</code>	Alert (<code>\x{7}</code>), escape (<code>\x{1B}</code>), form feed (<code>\x{B}</code>), newline (<code>\x{A}</code>), carriage return (<code>\x{D}</code>), tab (<code>\x{9}</code>).	<code>\n</code>

Expression	Description	Example
\cc, where c is in [A-Z] or one of @ [\] ^ _ ?	The control character corresponding to the character c.	\cH is a backspace (\x{8}).
\c, where c is not in [A-Za-z0-9]	The character c.	\c
\Q ... \E	Everything between the start and the end of the quotation.	\Q(...)\\E matches the string (...).

Character Classes

[C ₁ C ₂ ...], where C _i are characters, ranges c-d, or character classes	Any of the characters represented by C ₁ , C ₂ ...	[0-9+-]
[^...]	Complement of a character class.	[^\\d\\s]
[...&&...]	Intersection of character classes.	[\\p{L}&&[^A-Za-z]]
\p{...}, \P{...}	A predefined character class (see Table 9.5); its complement.	\p{L} matches a Unicode letter, and so does \p{L—you can omit braces around a single letter.}
\d, \D	Digits ([0-9], or \p{Digit} when the UNICODE_CHARACTER_CLASS flag is set); the complement.	\d+ is a sequence of digits.
\w, \W	Word characters ([a-zA-Z0-9_], or Unicode word characters when the UNICODE_CHARACTER_CLASS flag is set); the complement.	
\s, \S	Spaces ([\\n\\r\\t\\f\\x{B}], or \p{IsWhite_Space} when the UNICODE_CHARACTER_CLASS flag is set); the complement.	\s*, \s* is a comma surrounded by optional white space.

Expression	Description	Example
<code>\h, \v, \H, \V</code>	Horizontal whitespace, vertical whitespace, their complements.	
Sequences and Alternatives		
<code>XY</code>	Any string from <code>X</code> , followed by any string from <code>Y</code> .	<code>[1-9][0-9]*</code> is a positive number without leading zero.
<code>X Y</code>	Any string from <code>X</code> or <code>Y</code> .	<code>http ftp</code>
Grouping (see Section 9.4.4)		
<code>(X)</code>	Captures the match of <code>X</code> .	<code>'([^\']*)'</code> captures the quoted text.
<code>\n</code>	The <i>n</i> th group.	<code>([""]).*\1</code> matches 'Fred' or "Fred" but not 'Fred'.
<code>(?<name>X)</code>	Captures the match of <code>X</code> with the given name.	<code>'(?<id>[A-Za-z0-9]+)'</code> captures the match with name <code>id</code> .
<code>\k<name></code>	The group with the given name.	<code>\k<id></code> matches the group with name <code>id</code> .
<code>(?:X)</code>	Use parentheses without capturing <code>X</code> .	In <code>(?:http ftp)://(.*)</code> , the match after <code>://</code> is <code>\1</code> .
<code>(?f₁f₂...:X), (?f₁...-f_k...:X), with f_i in [dimsuUx]</code>	Matches, but does not capture, <code>X</code> with the given flags (see Section 9.4.7) on or off (after <code>-</code>).	<code>(?i:jpe?g)</code> is a case-insensitive match.
Other <code>(?....)</code>	See the Pattern API documentation.	
Quantifiers		
<code>X?</code>	Optional <code>X</code> .	<code>\+?</code> is an optional + sign.
<code>X*, X+</code>	0 or more <code>X</code> , 1 or more <code>X</code> .	<code>[1-9][0-9]+</code> is an integer ≥ 10 .

Expression	Description	Example
$X\{n\}$, $X\{n,\}$, $X\{m,n\}$	n times X , at least n times X , between m and n times X .	$[0-7]\{1,3\}$ are one to three octal digits.
$Q?$, where Q is a quantified expression	Reluctant quantifier, attempting the shortest match before trying longer matches.	$.^*(<.+?>).^*$ matches the shortest sequence enclosed in angle brackets.
Q^+ , where Q is a quantified expression	Possessive quantifier, taking the longest match without backtracking.	$'[^']^+'$ matches strings enclosed in single quotes and fails quickly on strings without a closing quote.
Boundary Matches		
$^$ $$$	Beginning, end of input (or beginning, end of line in multiline mode).	Java$ matches the input or line Java.
$\backslash A$ $\backslash Z$ $\backslash z$	Beginning of input, end of input, absolute end of input (unchanged in multiline mode).	
$\backslash b$ $\backslash B$	Word boundary, nonword boundary.	$\backslash bJava\backslash b$ matches the word Java.
$\backslash b\{g\}$	Grapheme cluster boundary	Useful with <code>split</code> to decompose a string into grapheme clusters
$\backslash R$	A Unicode line break.	
$\backslash G$	The end of the previous match.	

Table 9.5: Predefined Character Classes $\backslash p\{\dots\}$

Name	Description
<i>posixClass</i>	<i>posixClass</i> is one of Lower, Upper, Alpha, Digit, Alnum, Punct, Graph, Print, Cntrl, XDigit, Space, Blank, ASCII, interpreted as POSIX or Unicode class, depending on the <code>UNICODE_CHARACTER_CLASS</code> flag.
<i>IsScript</i> , <i>sc=Script</i> , <i>script=Script</i>	A script accepted by <code>Character.UnicodeScript.forName</code> .

Name	Description
InBlock, blk=Block, block=Block	A block accepted by Character.UnicodeBlock.forName.
Category, InCategory, gc=Category, general_category=Category	A one- or two-letter name for a Unicode general category.
IsProperty	Property is one of Alphabetic, Ideographic, Letter, Lowercase, Uppercase, Titlecase, Punctuation, Control, White_Space, Digit, Hex_Digit, Join_Control, Noncharacter_Code_Point, Assigned.
javaMethod	Invokes the method Character.isMethod (must not be deprecated).

9.4.2. Testing a Match

Generally, there are two ways to use a regular expression: Either you want to test whether a string matches the expression, or you want to find one or more matches of the expression in a string.

The static `matches` method tests whether an *entire string* matches a regular expression:

```
String regex = "[+-]?\d+";
CharSequence input = ...;
if (Pattern.matches(regex, input)) {
    // input matches the regular expression
    ...
}
```

If you need to use the same regular expression many times, it is more efficient to compile it. Then, create a `Matcher` for each input:

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) ...
```

If the match succeeds, you can retrieve the location of matched groups—see [Section 9.4.4](#).

To test whether a string *contains* a match, use the `find` method instead:

```
if (matcher.find()) {
    // A substring of input matches the regular expression
    ...
}
```

The `match` and `find` methods mutate the state of the `Matcher` object. If you just want to find out whether a given `Matcher` has found a match, call the `hasMatch` method instead.

You can turn the pattern into a predicate. This is particularly useful with the `filter` method of a stream:

```
Pattern digits = Pattern.compile("[0-9]+");
List<String> strings = List.of("December", "31st", "1999");
List<String> matchingStrings = strings.stream()
    .filter(digits.asMatchPredicate())
    .toList(); // ["1999"]
```

The result contains all strings that match the regular expression.

Use the `asPredicate` method to test whether a string *contains* a match:

```
List<String> stringsContainingMatch = strings.stream()
    .filter(digits.asPredicate())
    .toList(); // ["31st", "1999"]
```

9.4.3. Finding All Matches

In this section, we consider a common use case for regular expressions—finding all matches in an input. Use this loop:

```
String input = ...;
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
    String match = matcher.group();
    int matchStart = matcher.start();
    int matchEnd = matcher.end();
    ...
}
```

In this way, you can process each match in turn. As shown in the code fragment, you can get the matched string as well as its position in the input string.

More elegantly, you can call the `results` method to get a `Stream<MatchResult>`. The `MatchResult` interface has methods `group`, `start`, and `end`, just like `Matcher`. (In fact, the `Matcher` class implements this interface.) Here is how you get a list of all matches:

```
List<String> matches = pattern.matcher(input)
    .results()
    .map(MatchResult::group)
    .toList();
```

If you have the data in a file, then you can use the `Scanner.findAll` method to get a `Stream<MatchResult>`, without first having to read the contents into a string. You can pass a `Pattern` or a pattern string:

```
var in = new Scanner(path);
Stream<String> words = in.findAll("\\pL+")
    .map(MatchResult::group);
```

9.4.4. Groups

It is common to use groups for extracting components of a match. For example, suppose you have a line item in the invoice with item name, quantity, and unit price such as

Blackwell Toaster USD29.95

Here is a regular expression with groups for each component:

```
(\p{Alnum}+(\s+\p{Alnum}+*))\s+([A-Z]{3})([0-9.]*)
```

After matching, you can extract the `n`th group from the matcher as

```
String contents = matcher.group(n);
```

Groups are ordered by their opening parenthesis, starting at 1. (Group 0 is the entire input.) In this example, here is how to take the input apart:

```
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) {
    item = matcher.group(1);
    currency = matcher.group(3);
    price = matcher.group(4);
}
```

We aren't interested in group 2; it only arose from the parentheses that were required for the repetition. For greater clarity, you can denote that group as "non-capturing". Then it doesn't show up as a group in the matcher.

```
(\p{Alnum}+(:\s+\p{Alnum}+*))\s+([A-Z]{3})([0-9.]*)
```

Or, even better, use named groups:

```
(?<item>\p{Alnum}+(\s+\p{Alnum}+*))\s+(<currency>[A-Z]{3})(?<price>[0-9.]*)
```

Then, you can retrieve the groups by name:

```
item = matcher.group("item");
```

With the `start` and `end` methods of the `Matcher` and `MatchResult` classes, you can get the group positions in the input:

```
int itemStart = matcher.start("item");
int itemEnd = matcher.end("item");
```

The `namedGroups` method yields a `Map<String, Integer>` from group names to numbers.



Note: When you have a group inside a repetition, such as `(\s+\p{Alnum}+)*` in the example above, it is not possible to get all of its matches. The `group` method only yields the last match, which is rarely useful. You need to capture the entire expression with another group.

9.4.5. Splitting along Delimiters

Sometimes, you want to break an input along matched delimiters and keep everything else. The `Pattern.split` method automates this task. You obtain an array of strings, with the delimiters removed:

```
String input = ...;
Pattern commas = Pattern.compile("\s*,\s*");
String[] tokens = commas.split(input);
// "1, 2, 3" turns into ["1", "2", "3"]
```

If there are many tokens, you can fetch them lazily:

```
Stream<String> tokens = commas.splitAsStream(input);
```

To also collect the delimiters, use the `splitWithDelimiters` method:

```
tokens = commas.splitWithDelimiters(input, -1); // ["1", "", "", "2", "", "", "3", "", "", "4"]
```

If the second argument is a positive number n , the separator pattern is applied at most $n - 1$ times. and the last element is the remaining string. Otherwise, the pattern is applied as often as possible. With a limit of zero, trailing empty strings are discarded.

If you don't care about precompiling the pattern or lazy fetching, you can just use the `split` and `splitWithDelimiter` methods of the `String` class:

```
tokens = input.split("\s*,\s*");
```



Caution: It is easy to forget that the argument of `split` is a regular expression. For example,

```
"com.horstmann.corejava".split(".")
```

does not split along the dots. Instead, every character is a separator, and the result is an empty array!

You need to escape the dot with a backslash in the regular expression, and therefore with two backslashes in the string literal:

```
"com.horstmann.corejava".split("\\\\.")
```

Alternatively, use the `Pattern.quote` method:

```
"com.horstmann.corejava".split(Pattern.quote("."));
```

If the input is in a file, use a scanner:

```
var in = new Scanner(path);
in.useDelimiter("\\s*,\\s*");
Stream<String> tokens = in.tokens();
```

9.4.6. Replacing Matches

If you want to replace all matches of a regular expression with a string, call `replaceAll` on the matcher:

```
Matcher matcher = commas.matcher(input);
String result = matcher.replaceAll(",");
// Normalizes the commas
```

Or, if you don't care about precompiling, use the `replaceAll` method of the `String` class.

```
String result = input.replaceAll("\\s*,\\s*", ",");
```

The replacement string can contain group numbers `$n` or names `${name}`. They are replaced with the contents of the corresponding captured group.

```
String result = "3:45".replaceAll(
  "(\\d{1,2}):(?<minutes>\\d{2})",
  "$1 hours and ${minutes} minutes");
// Sets result to "3 hours and 45 minutes"
```

You can use `\` to escape `$` and `\` in the replacement string, or you can call the `Matcher.quoteReplacement` convenience method:

```
matcher.replaceAll(Matcher.quoteReplacement(str))
```

If you want to carry out a more complex operation than splicing in group matches, then you can provide a replacement function instead of a replacement string. The function accepts a `MatchResult` and yields a string. For example, here we replace all words with at least four letters with their uppercase version:

```
String result = Pattern.compile("\\pL{4,}")
    .matcher("Mary had a little lamb")
    .replaceAll(m -> m.group().toUpperCase());
// Yields "MARY had a LITTLE LAMB"
```

The `replaceFirst` method replaces only the first occurrence of the pattern.

9.4.7. Flags

Several *flags* change the behavior of regular expressions. You can specify them when you compile the pattern:

```
Pattern pattern = Pattern.compile(regex,
    Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CHARACTER_CLASS);
```

Or you can specify them inside the pattern:

```
String regex = "(?iU:expression)";
```

Here are the flags:

- `Pattern.CASE_INSENSITIVE` or `i`: Match characters independently of the letter case. By default, this flag takes only US ASCII characters into account.
- `Pattern.UNICODE_CASE` or `u`: When used in combination with `CASE_INSENSITIVE`, use Unicode letter case for matching.
- `Pattern.UNICODE_CHARACTER_CLASS` or `U`: Select Unicode character classes instead of POSIX. Implies `UNICODE_CASE`.
- `Pattern.MULTILINE` or `m`: Make `^` and `$` match the beginning and end of a line, not the entire input.
- `Pattern.UNIX_LINES` or `d`: Only `'\n'` is a line terminator when matching `^` and `$` in multiline mode.
- `Pattern.DOTALL` or `s`: Make the `.` symbol match all characters, including line terminators.
- `Pattern.COMMENTS` or `x`: Whitespace and comments (from `#` to the end of a line) are ignored.
- `Pattern.LITERAL`: The pattern is taking literally and must be matched exactly, except possibly for letter case.
- `Pattern.CANON_EQ`: Take canonical equivalence of Unicode characters into account. For example, `u` followed by `”` (diaeresis) matches `ü`.

The last two flags cannot be specified inside a regular expression.

9.5. Serialization

In the following sections, you will learn about object serialization—a mechanism for turning an object into a bunch of bytes that can be shipped somewhere else or stored on disk, and for reconstituting the object from those bytes.

Serialization is an essential tool for distributed processing, where objects are shipped from one virtual machine to another. It is also used for fail-over and load balancing, when serialized objects can be moved to another server. If you work with server-side software, you will often need to enable serialization for classes. The following sections tell you how to do that.

9.5.1. The Serializable Interface

In order for an object to be serialized—that is, turned into a bunch of bytes—it must be an instance of a class that implements the `Serializable` interface. This is a marker interface with no methods, similar to the `Cloneable` interface that you saw in [Chapter 4](#).

For example, to make `Employee` objects serializable, the class needs to be declared as

```
public class Employee implements Serializable {  
    private String name;  
    private double salary;  
    ...  
}
```

It is appropriate for a class to implement the `Serializable` interface if all instance variables have primitive or enum type, or contain references to serializable objects. Many classes in the standard library are serializable. Arrays and the collection classes that you saw in [Chapter 7](#) are serializable provided their elements are.

In the case of the `Employee` class, and indeed with most classes, there is no problem. In the following sections, you will see what to do when a little extra help is needed.

To serialize objects, you need an `ObjectOutputStream`, which is constructed with another `OutputStream` that receives the actual bytes.

```
var out = new ObjectOutputStream(Files.newOutputStream(path));
```

Now call the `writeObject` method:

```
var peter = new Employee("Peter", 90000);  
var paul = new Manager("Paul", 180000);  
out.writeObject(peter);  
out.writeObject(paul);
```

To read the objects back in, construct an `ObjectInputStream`:

```
var in = new ObjectInputStream(Files.newInputStream(path));
```

Retrieve the objects in the same order in which they were written, using the `readObject` method.

```
var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

When an object is written, the name of the class and the names and values of all instance variables are saved. If the value of an instance variable belongs to a primitive type, it is saved as binary data. If it is an object, it is again written with the `writeObject` method.

When an object is read in, the process is reversed. The class name and the names and values of the instance variables are read, and the object is reconstituted.

There is just one catch. Suppose there were two references to the same object. Let's say each employee has a reference to their boss:

```
var peter = new Employee("Peter", 90000);
var paul = new Manager("Barney", 105000);
var mary = new Manager("Mary", 180000);
peter.setBoss(mary);
paul.setBoss(mary);
out.writeObject(peter);
out.writeObject(paul);
```

When reading these two objects back in, both of them need to have the *same* boss, not two references to identical but distinct objects.

In order to achieve this, each object gets a *serial number* when it is saved. When you pass an object reference to `writeObject`, the `ObjectOutputStream` checks if the object reference was previously written. In that case, it just writes out the serial number and does not duplicate the contents of the object.

In the same way, an `ObjectInputStream` remembers all objects it has encountered. When reading in a reference to a repeated object, it simply yields a reference to the previously read object.



Note: If the superclass of a serializable class is not serializable, it must have an accessible no-argument constructor. Consider this example:

```
class Person // Not serializable
class Employee extends Person implements Serializable
```

When an `Employee` object is deserialized, its instance variables are read from the object input stream, but the `Person` instance variables are set by the `Person` constructor.

9.5.2. Transient Instance Variables

Certain instance variables should not be serialized—for example, database connections that are meaningless when an object is reconstituted. Also, when an object keeps a cache of values, it might be better to drop the cache and recompute it instead of storing it.

To prevent an instance variable from being serialized, simply tag it with the `transient` modifier. Always mark instance variables as transient if they hold instances of nonserializable classes. Transient instance variables are skipped when objects are serialized.

9.5.3. The `readObject` and `writeObject` Methods

In rare cases, you need to tweak the serialization mechanism. A serializable class can add any desired action to the default read and write behavior, by defining methods with the signature

```
@Serial private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
@Serial private void writeObject(ObjectOutputStream out)
    throws IOException
```

Then, the object headers continue to be written as usual, but the instance variables fields are no longer automatically serialized. Instead, these methods are called.

Note the `@Serial` annotation. The methods for tweaking serialization don't belong to interfaces. Therefore, you can't use the `@Override` annotation to have the compiler check the method declarations. The `@Serial` annotation is meant to enable the same checking for serialization methods. Up to Java 17, the `javac` compiler doesn't do that checking, but it might happen in the future. Some IDEs check the annotation.

A number of classes in the `java.awt.geom` package, such as `Point2D.Double`, are not serializable. Now, suppose you want to serialize a class `LabeledPoint` that stores a `String` and a `Point2D.Double`. First, you need to mark the `Point2D.Double` field as transient to avoid a `NotSerializableException`.

```
public class LabeledPoint implements Serializable {
    private String label;
    private transient Point2D.Double point;
    ...
}
```

In the `writeObject` method, first write the object descriptor and the `String` field, `label`, by calling the `defaultWriteObject` method. This is a special method of the `ObjectOutputStream` class that can only be called from within a `writeObject` method of a serializable class. Then we write the point coordinates, using the standard `DataOutput` calls.

```
@Serial before private void writeObject(ObjectOutputStream out) throws IOException {  
    out.defaultWriteObject();  
    out.writeDouble(point.getX());  
    out.writeDouble(point.getY());  
}
```

In the `readObject` method, we reverse the process:

```
@Serial before private void readObject(ObjectInputStream in)  
    throws IOException, ClassNotFoundException {  
    in.defaultReadObject();  
    double x = in.readDouble();  
    double y = in.readDouble();  
    point = new Point2D.Double(x, y);  
}
```

Another example is the `HashSet` class that supplies its own `readObject` and `writeObject` methods. Instead of saving the internal structure of the hash table, the `writeObject` method simply saves the capacity, load factor, size, and elements. The `readObject` method reads back the capacity and load factor, constructs a new table, and inserts the elements.

The `readObject` and `writeObject` methods only need to save and load their data. They do not concern themselves with superclass data or any other class information.

The `Date` class uses this approach. Its `writeObject` method saves the milliseconds since the “epoch” (January 1, 1970). The data structure that caches calendar data is not saved.



Caution: Just like a constructor, the `readObject` method operates on partially initialized objects. If you call a non-final method inside `readObject` that is overridden in a subclass, it may access uninitialized data.



Note: If a serializable class defines a field

```
@Serial private static final ObjectStreamField[] serialPersistentFields
```

then serialization uses those field descriptors instead of the non-transient non-static fields. There is also an API for setting the field values before serialization or reading them after deserialization. This is useful for preserving a legacy layout after a class

has evolved. For example, the `BigDecimal` class uses this mechanism to serialize its instances in a format that no longer reflects the instance fields.

9.5.4. The `readExternal` and `writeExternal` Methods

Instead of letting the serialization mechanism save and restore object data, a class can define its own mechanism. For example, you can encrypt the data or use a format that is more efficient than the serialization format.

To do this, a class implements the `Externalizable` interface instead of the `Serializable` interface. This, in turn, requires two methods:

```
public void readExternal(ObjectInputStream in)
    throws IOException
public void writeExternal(ObjectOutputStream out)
    throws IOException
```

Unlike the `readObject` and `writeObject` methods, these methods are fully responsible for saving and restoring the entire object, *including the superclass data*. When writing an object, the serialization mechanism merely records the class of the object in the output stream. When reading an externalizable object, the object input stream creates an object with the no-argument constructor and then calls the `readExternal` method.

In this example, the `LabeledPixel` class extends the serializable `Point` class, but it takes over the serialization of the class and superclass. The fields of the object are not stored in the standard serialization format. Instead, the data are placed in an opaque block.

```
public class LabeledPixel extends Point implements Externalizable {
    private String label;

    public LabeledPixel() {} // required for externalizable class

    @Override public void writeExternal(ObjectOutput out)
        throws IOException {
        out.writeInt((int) getX());
        out.writeInt((int) getY());
        out.writeUTF(label);
    }

    @Override public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        int x = in.readInt();
        int y = in.readInt();
        setLocation(x, y);
        label = in.readUTF();
```

```
    }  
    ...  
}
```



Note: The `readExternal` and `writeExternal` methods should not be annotated with `@Serial`. Since they are defined in the `Externalizable` interface, you can simply annotate them with `@Override`.



Caution: Unlike the `readObject` and `writeObject` methods, which are private and can only be called by the serialization mechanism, the `readExternal` and `writeExternal` methods are public. In particular, `readExternal` potentially permits modification of the state of an existing object.



Note: You cannot customize the serialization of enumerations and records. If you define `readObject/writeObject` or `readExternal/writeExternal` methods, they are not used for serialization.

9.5.5. The `readResolve` and `writeReplace` Methods

We take it for granted that objects can only be constructed with the constructor. However, a deserialized object is *not constructed*. Its instance variables are simply restored from an object stream.

This is a problem if the constructor enforces some condition. For example, a singleton object may be implemented so that the constructor can only be called once. As another example, database entities can be constructed so that they always come from a pool of managed instances.

You shouldn't implement your own mechanism for singletons. If you need a singleton, make an enumerated type with one instance that is, by convention, called `INSTANCE`.

```
public enum PersonDatabase {  
    INSTANCE;  
  
    public Person findById(int id) { ... }  
    ...  
}
```

This works because `enum` are guaranteed to be deserialized properly.

Now let's suppose that you are in the rare situation where you want to control the identity of each deserialized instance. As an example, suppose a `Person` class wants to restore its instances from a database when deserializing. Then you should not serialize the object

itself. Instead request that a proxy instance is saved. When restored, that proxy locates and constructs the desired object. Your class needs to provide a `writeReplace` method that returns the proxy object:

```
public class Person implements Serializable {  
    private int id;  
    // Other instance variables  
    ...  
    @Serial private Object writeReplace() {  
        return new PersonProxy(id);  
    }  
}
```

When a `Person` object is serialized, none of its instance variables are saved. Instead, the `writeReplace` method is called and *its return value* is serialized and written to the stream.

The proxy class needs to implement a `readResolve` method that yields a `Person` instance:

```
class PersonProxy implements Serializable {  
    private int id;  
  
    public PersonProxy(int id) {  
        this.id = id;  
    }  
  
    @Serial private Object readResolve() {  
        return PersonDatabase.INSTANCE.findById(id);  
    }  
}
```

When the `readObject` method finds a `PersonProxy` in an `ObjectInputStream`, it deserializes the proxy, calls its `readResolve` method, and returns the result.



Note: Unlike the `readObject` and `writeObject` methods, the `readResolve` and `writeReplace` methods need not be private.



Note: With enumerations and records, `readObject/writeObject` or `readExternal/writeExternal` methods are not used for serialization. With records, but not with enumerations, the `writeReplace` method will be used.

9.5.6. Versioning

Serialization was intended for sending objects from one virtual machine to another, or for short-term persistence of state. If you use serialization for long-term persistence, or in any

situation where classes can change between serialization and deserialization, you will need to consider what happens when your classes evolve. Can version 2 read the old data? Can the users who still use version 1 read the files produced by the new version?

The serialization mechanism supports a simple versioning scheme. When an object is serialized, both the name of the class and its `serialVersionUID` are written to the object stream. That unique identifier is assigned by the implementor, by defining an instance variable

```
@Serial private static final long serialVersionUID = 1L; // Version 1
```

When the class evolves in an incompatible way, the implementor should change the UID. Whenever a deserialized object has a nonmatching UID, the `readObject` method throws an `InvalidClassException`.

If the `serialVersionUID` matches, deserialization proceeds even if the implementation has changed. Each non-transient instance variable of the object to be read is set to the value in the serialized state, provided that the name and type match. All other instance variables are set to the default: `null` for object references, zero for numbers, and `false` for boolean values. Anything in the serialized state that doesn't exist in the object to be read is ignored.

Is that process safe? Only the implementor of the class can tell. If it is, then the implementor should give the new version of the class the same `serialVersionUID` as the old version.

If you don't assign a `serialVersionUID`, one is automatically generated by hashing a canonical description of the instance variables, methods, and supertypes. You can see the hash code with the `serialver` utility. The command

```
serialver ch09.sec05.Employee
```

displays

```
private static final long serialVersionUID = -4932578720821218323L;
```

When the class implementation changes, there is a very high probability that the hash code changes as well.

If you need to be able to read old version instances, and you are certain that is safe to do so, run `serialver` on the old version of your class and add the result to the new version.



Note: If you want to implement a more sophisticated versioning scheme, override the `readObject` method and call the `readFields` method instead of the `defaultReadObject` method. You get a description of all fields found in the stream, and you can do with them what you want.



Note: Enumerations and records ignore the serialVersionUID field. An enumeration always has a serialVersionUID of 0L. You can declare the serialVersionUID of a record, but the IDs don't have to match for deserialization.



Note: In this section, you saw what happens when the reader's version of a class has instance variables that aren't present in the object stream. It is also possible during class evolution for a superclass to be added. Then a reader using the new version may read an object stream in which the instance variables of the superclass are not set. By default, those instance fields are set to their 0/false/null default. That may leave the superclass in an unsafe state. The superclass can defend against that problem by defining an initialization method

```
@Serial private void readObjectNoData() throws ObjectStreamException
```

The method should either set the same state as the no-argument constructor or throw an InvalidObjectException. It is only called in the unusual circumstance where an object stream is read that contains an instance of a subclass with missing superclass data.

9.5.7. Deserialization and Security

During deserialization of a serializable class, objects are created without invoking any constructor of the class. Even if the class has a no-argument constructor, it is not used. The field values are set directly from the values of the object input stream.



Note: For serializable *records*, deserialization calls the canonical constructor, passing it the values of the components from the object input stream. (As a consequence, cyclic references in records are not restored.)

Bypassing construction is a security risk. An attacker can craft bytes describing an invalid object that could have never been constructed. Suppose, for example, that the Employee constructor throws an exception when called with a negative salary. We would like to think that no Employee object can have a negative salary as a result. But it is not difficult to inspect the bytes for a serialized object and modify some of them. This way, one can craft bytes for an employee with a negative salary and then deserialize them.

A serializable class can optionally implement the ObjectInputValidation interface and define a validateObject method to check whether its objects are properly deserialized. For example, the Employee class can check that salaries are not negative:

```
public void validateObject() throws InvalidObjectException {  
    System.out.println("validateObject");  
    if (salary < 0)  
        throw new InvalidObjectException("salary < 0");  
}
```

Unfortunately, the method is not invoked automatically. To invoke it, you also must provide the following method:

```
@Serial private void readObject(ObjectInputStream in)  
    throws IOException, ClassNotFoundException {  
    in.registerValidation(this, 0);  
    in.defaultReadObject();  
}
```

The object is then scheduled for validation, and the `validateObject` method is called when this object and all dependent objects have been loaded. The second parameter lets you specify a priority. Validation requests with higher priorities are done first.

There are other security risks. Adversaries can create data structures that consume enough resources to crash a virtual machine. More insidiously, any class on the class path can be deserialized. Hackers have been devious about piecing together “gadget chains”—sequences of operations in various utility classes that use reflection and culminate in calling methods such as `Runtime.exec` with a string of their choice.

Any application that receives serialized data from untrusted sources over a network connection is vulnerable to such attacks. For example, some servers serialize session data and deserialize whatever data are returned in the HTTP session cookie.

You should avoid situations in which arbitrary data from untrusted sources are deserialized. In the example of session data, the server should sign the data, and only deserialize data with a valid signature.

A *serialization filter* mechanism can harden applications from such attacks. The filters see the names of deserialized classes and several metrics (stream size, array sizes, total number of references, longest chain of references). Based on those data, the deserialization can be aborted.

In its simplest form, you provide a pattern describing the valid and invalid classes. For example, if you start our sample serialization demo as

```
java -Djdk.serialFilter='serial.*;java.**;!*' serial.ObjectStreamTest
```

then the objects will be loaded. The filter allows all classes in the `serial` package and all classes whose package name starts with `java`, but no others. If you don’t allow `java.**`, or at least `java.util.Date`, deserialization fails.

You can place the filter pattern into a configuration file and specify multiple filters for different purposes. You can also implement your own filters. See <https://docs.oracle.com/en/java/javase/21/core/serialization-filtering1.html> for details.

9.6. Exercises

1. Write a utility method for copying all of an `InputStream` to an `OutputStream`, without using any temporary files. Provide another solution, without a loop, using operations from the `Files` class, using a temporary file.
2. Write a program that reads a text file and produces a file with the same name but extension `.toc`, containing an alphabetized list of all words in the input file together with a list of line numbers in which each word occurs. Assume that the file's encoding is UTF-8.
3. Write a program that reads a file containing text and, assuming that most words are English, guesses whether the encoding is ASCII, ISO 8859-1, UTF-8, or UTF-16, and if the latter, which byte ordering is used.
4. Using a `Scanner` is convenient, but it is a bit slower than using a `BufferedReader`. Read in a long file a line at a time, counting the number of input lines, with (a) a `Scanner` and `hasNextLine/nextLine`, (b) a `BufferedReader` and `readLine`, (c) a `BufferedReader` and `lines`. Which is the fastest? The most convenient?
5. When an encoder of a `Charset` with partial Unicode coverage can't encode a character, it replaces it with a default—usually, but not always, the encoding of "?". Find all replacements of all available character sets that support encoding. Use the `newEncoder` method to get an encoder, and call its `replacement` method to get the replacement. For each unique result, report the canonical names of the `Charsets` that use it.
6. The BMP file format for uncompressed image files is well documented and simple. Using random access, write a program that reflects each row of pixels in place, without writing a new file.
7. Look up the API documentation for the `MessageDigest` class and write a program that computes the SHA-512 digest of a file. Feed blocks of bytes to the `MessageDigest` object with the `update` method, then display the result of calling `digest`. Verify that your program produces the same result as the `sha512sum` utility.
8. Write a utility method for producing a ZIP file containing all files from a directory and its descendants.
9. Using the `URLConnection` class, read data from a password-protected web page with "basic" authentication. Concatenate the user name, a colon, and the password, and compute the Base64 encoding:

```
String input = username + ":" + password;
String encoding = Base64.getEncoder().encodeToString(input.getBytes());
```

Set the HTTP header `Authorization` to the value `"Basic " + encoding`. Then read and print the page contents.

10. Using a regular expression, extract all decimal integers (including negative ones) from a string into an `ArrayList<Integer>` (a) using `find`, and (b) using `split`. Note that a `+` or `-` that is not followed by a digit is a delimiter.

11. Using regular expressions, extract the directory path names (as an array of strings), the file name, and the file extension from an absolute or relative path such as /home/cay/myfile.txt.
12. Come up with a realistic use case for using group references in `Matcher.replaceAll` and implement it.
13. Implement a method that can produce a clone of any serializable object by serializing it into a byte array and deserializing it.
14. Implement a serializable class `Point` with instance variables for `x` and `y`. Write a program that serializes an array of `Point` objects to a file, and another that reads the file.
15. Continue the preceding exercise, but change the data representation of `Point` so that it stores the coordinates in an array. What happens when the new version tries to read a file generated by the old version? What happens when you fix up the `serialVersionUID`? Suppose your life depended upon making the new version compatible with the old. What could you do?
16. Which classes in the standard Java library implement `Externalizable`? Which of them use `writeReplace/readResolve`?
17. Unzip the API source and investigate how the `LocalDate` class is serialized. Why does the class define `writeExternal` and `readExternal` methods even though it doesn't implement `Externalizable`? (Hint: Look at the `Ser` class.) Why does the class define a `readObject` method? How could it be invoked?

This page intentionally left blank

Index

Symbols

- ! operator
 - in property files [261](#)
 - operator [18, 23](#)
- != operator [18, 23](#)
 - for wrapper classes [51](#)
- "" [27, 28, 156](#)
 - "" [33](#)
- '..." (single quotes, for strings)
 - for strings [7](#)
 - in javadoc hyperlinks [101](#)
 - in text blocks [34](#)
- # (number sign)
 - flag (for output) [39](#)
 - in javadoc hyperlinks [101](#)
 - in option files [440](#)
 - in property files [261](#)
- \$ (dollar sign)
 - flag (for output) [39](#)
 - in regular expressions [325, 326, 329, 334, 335](#)
 - in variable names [16](#)
- % (percent sign)
 - conversion character [37, 38](#)
 - operator [18, 20](#)
- %% [215](#)
- %= operator [18](#)
- %g [215](#)
- %h [215](#)
- %t [215](#)
- %u [215](#)
- & (ampersand) [18, 24](#)
- && operator
 - in regular expressions [327](#)
 - operator [18, 23](#)
- &= operator [18](#)
- > (right angle bracket)
 - in shell syntax [36](#)
 - operator [23](#)
- >>=, >>> operators [18](#)
- >=, >>, >>> operators [18, 23, 24](#)
- < (left angle bracket)
 - flag (for output) [39](#)
 - in shell syntax [36](#)
 - operator [23](#)
- <>
 - for constructors of generic classes [220](#)
- << operator [18, 24](#)
- <<= operator [18](#)
- <...> (angle brackets)
 - for element types, in array lists [50](#)
 - for type parameters [117, 220](#)
 - in javadoc hyperlinks [101](#)
- in regular expressions [328](#)
- <= operator [18, 23](#)
- '... (for character literals)
 - for character literals [14](#)
- ((left parenthesis) [39](#)
- (...) (parentheses)
 - empty, for anonymous classes [138](#)
 - for casts [22, 109](#)
 - in regular expressions [325, 328, 332](#)
 - operator [18](#)
- * (asterisk)
 - for annotation processors [416](#)
 - in documentation comments [99](#)
 - in regular expressions [325, 328, 333](#)
 - operator [18, 19](#)
- wildcard
 - in class path [89](#)
 - in imported classes [91](#)
- *= operator [18](#)
- + (plus sign)
 - flag (for output) [39](#)
 - in regular expressions [325, 328](#)
 - operator [18, 19](#)
 - for strings [25, 28, 156](#)
- ++ operator [18, 20](#)
- += operator [18](#)
- , (comma)
 - flag (for output) [39](#)
 - normalizing [334](#)
 - trailing, in arrays [49](#)
- (minus sign)
 - flag (for output) [39](#)
 - in regular expressions [325](#)
 - operator [18, 19](#)
- > operator [121, 124](#)
- operator
 - in command-line options [88](#)
 - operator [18, 20](#)
- = operator [18, 19](#)
- . (period) [314](#)
 - in method calls [6](#)
 - in package names [3, 86](#)
 - in regular expressions [325, 326, 335](#)
 - operator [18](#)
- ... (ellipsis) [59](#)
- / (slash)
 - file separator (Unix) [262, 313](#)
 - in javac path segments [3](#)
 - operator [18, 19](#)
 - root component [313](#)
- /*...*/ [98](#)
- //, /*...*/ comments [2](#)
- /= operator [18](#)
- 0

as default value [75, 77](#)
 flag (for output) [39](#)
 prefix (for octal literals) [13](#)
0b [13](#)
0x [13, 39](#)
0xFF [305](#)
: (colon)
 in assertions [206](#)
 in switch statement [42](#)
 path separator (Unix) [89, 262](#)
 switch [41](#)
 switch statement [42, 171](#)
:: (C++ operator) [124, 147](#)
; (semicolon)
 path separator (Windows) [89, 262](#)
= operator [18, 19](#)
== operator [18, 23, 158](#)
 for class objects [175](#)
 for enumerations [164](#)
 for strings [27](#)
 for wrapper classes [51](#)
? (question mark)
 in regular expressions [325, 326, 328](#)
 replacement character [309](#)
 wildcard, for types [223, 228, 239](#)
?: [18, 23](#)
@ (at sign)
 in java command [440](#)
 in javadoc comments [98](#)
[...] (brackets)
 for arrays [48, 55](#)
 in regular expressions [325, 327](#)
 operator [18](#)
[I [156, 175](#)
[L [175](#)
**** (backslash)
 character literal [15](#)
 file separator (Windows) [262, 313](#)
 in option files [440](#)
 in regular expressions [325, 326, 334](#)
 in text blocks [34](#)
\0 [326](#)
\a, \A, in regular expressions [326, 329](#)
\b (backslash character literal) [15](#)
\b, \B, in regular expressions [329](#)
\c [327](#)
\d, \D, in regular expressions [327](#)
\e, \E, in regular expressions [326](#)
\f (form feed character literal) [326](#)
\G [329](#)
\h, \H, in regular expressions [328](#)
\k [328](#)
\n (newline character literal)
 for character literals [15](#)
 in property files [261, 262](#)
 in regular expressions [326, 328, 335](#)
\p, \P, in regular expressions [327](#)
\Q [326, 327](#)
\r (carriage return character literal)
 for character literals [15](#)
 in property files [262](#)
\r, \R, in regular expressions [326, 329](#)
\s, \S, in regular expressions [327](#)
\t (tab character literal)
 in regular expressions [326](#)
 tab, for character literals [15](#)
\u (Unicode character literal)
 for character literals [14](#)
 in regular expressions [326](#)
\v, \V, in regular expressions [328](#)
\w, \W, in regular expressions [327](#)
\x [326](#)
\z, \Z, in regular expressions [329](#)
^ (caret)
 for function parameters [121](#)
 in regular expressions [325, 329, 335](#)
 operator [18, 24](#)
^= operator [18](#)
_ (underscore)
 in number literals [13](#)
 in variable names [16, 71](#)
{...} (braces)
 in annotation elements [401](#)
 in lambda expressions [122](#)
 in regular expressions [325, 329, 334](#)
 with arrays [49](#)
{...} [146](#)
| operator
 in regular expressions [325, 328](#)
 operator [18, 24](#)
|= operator [18](#)
|| operator [18, 23](#)
~ operator [18, 24](#)

A

a, A conversion characters [38](#)
 abstract keyword [109, 148](#)
 abstract classes [148](#)
 abstract methods [123](#)
 AbstractCollection class [114](#)
 AbstractMethodError class [114](#)
 AbstractProcessor class [416](#)
 accept [128, 129](#)
 acceptEither method [374, 375](#)
 AccessibleObject class
 setAccessible method [184, 185, 186](#)
 trySetAccessible method [184](#)
 accessors [66](#)
 accumulate [378](#)
 accumulateAndGet method [377](#)
 accumulator functions [293](#)
 ActionListener interface [120](#)
 add method
 of ArrayDeque [266](#)
 of ArrayList [50, 66](#)
 of BlockingQueue [365](#)
 of Collection [245](#)
 of List [247](#)

of `ListIterator` 252
of `LongAdder` 378
`addAll` method
 of `Collection` 226, 245
 of `Collections` 249
 of `List` 247
`addExact` method 21
`addition` 19
 identity for 293
`addSuppressed` method 200
`aggregators` 441
`allMatch` method 281
`allOf` method
 of `CompletableFuture` 374, 375
 of `EnumSet` 265
`allProcesses` method 392
`and, andNot` methods (`BitSet`) 263
`and, andThen` methods (functional interfaces) 128
`Android` 120, 376
`AnnotatedConstruct` interface 417
`AnnotatedElement` interface 413, 415
`Annotation` interface
 extending 408
`annotation interfaces` 405
`annotation processors` 416
`annotations`
 accessing 407, 442
 applicability of 409
 container 412, 414
 declaration 402
 documented 409, 411
 generating source code with 417
 inherited 409, 412, 414
 key/value pairs in 400, 407
 meta 406, 412
 modifiers and 404
 multiple 401
 processing
 at runtime 412
 source-level 415
 repeatable 401, 409, 412, 414
 standard 408
 type use 403
`anonymous classes` 137
`anyMatch` method 281
`anyOf` method 374, 375
`Apache Commons CSV` 438
`API documentation` 30, 32
 generating 98
`Applet` 178
`apply, applyAsXxx` methods (functional interfaces) 128, 129
`applyToEither` method 374, 375
`arithmetic operations` 18
`Array` class 188
`array lists` 50
 anonymous 146
 checking for nulls 227
 constructing 51
 converting between 223
 copying 53
 elements of 50, 52
 filling 53
 instantiating with type variables 234
 size of 51
 sorting 54
 variables of 50
`array variables`
 assigning values to 49
 copying 52
 declaring 48
 initializing 48
`ArrayBlockingQueue` class 366
`ArrayDeque` class 265
`ArrayIndexOutOfBoundsException` class 48
`ArrayList` class 50, 248
 `add` method 50, 66
 `clone` method 163
 `forEach` method 124
 `get, remove` methods 51
 `removeIf` method 123
 `set, size` methods 51
`arrays` 48, 50
 accessing nonexisting elements in 48
 allocating 234
 annotating 403
 `asList` method 265
 casting 188
 checking 188
 comparing 158
 computing values of 369
 constructing 48, 49
 constructor references with 126
 converting
 to a reference of type `Object` 155
 to/from streams 286, 296, 369
 copying 52
 `copyOf` method 52, 189
 covariant 223
 `deepToString` method 156
 `equals` method 158
 `fill` method 53
 filling 49, 53
 generating `Class` objects for 175
 growing 188, 189
 hash codes of 160
 `hashCode` method 160
 length of 48, 50, 134
 multidimensional 55, 156
 of bytes 302
 of generic types 126, 235
 of objects 49, 369
 of primitive types 369
 of strings 333
 `parallelXxx` methods 54, 369
 passing into methods 58
 printing 54, 57, 156
 `Serializable` 336
 `setAll` method 127
 `sort` method 54, 118, 119, 123, 124

sorting 54, 117, 369
 stream method 273, 294
 superclass assignment in 145
`toString` method 54, 156
 using class literals with 175
`ArrayStoreException` class 145, 223, 235
 ASCII 30, 304
 ASM tool 420
`asMatchPredicate`, `asPredicate` methods (Pattern) 331
`assert` keyword 206
`AssertionError` class 206
 assertions 205
 checking 403
 enabling/disabling 206
 assignment operators 19
 associative operations 293
`asSubclass` method 239
 asynchronous computations 369
`AsyncTask` 376
 atomic operations 361, 364, 376, 382
 performance and 377
`AtomicXxx` classes 377
`@author` annotation 99, 102
 autoboxing 51, 130
`AutoCloseable` interface 199, 221
 `close` method 199
 automatic modules 437, 439
`availableCharsets` method 305
`availableProcessors` method 351
 average method (`XxxStream`) 295

B

b, B conversion characters 38
 Base classes. See Superclasses
`BasicFileAttributes` interface 318
`BeanInfo` interface 188
`BiConsumer` interface 128
`BiFunction` interface 128, 130
 big-endian format 305, 310, 312
`BigDecimal` class 14, 24, 339
`BigInteger` class 12, 24
 binary data, reading/writing 310
 binary numbers 13, 14
 binary trees 252
`BinaryOperator` interface 129
`binarySearch` method 250
`BiPredicate` interface 129
`BitSet` class 262
 collecting streams into 294
 methods of 263
 bitwise operators 24
 block statement, labeled 46
 blocking queues 365
`BlockingQueue` interface 365, 366
 boolean type 15, 51
 default value of 75, 77
 formatting for output 38
 reading/writing 310

streams of 294
`BooleanSupplier` interface 129
 bootstrap class loader 178
 boxed 295
 branches 39
`break` keyword 42, 43, 44
 bridge methods 230
 clashes of 237
`BufferedReader` class 307
 build 322
 bulk operations 365
 byte type 12, 51, 302
 MAX VALUE, MIN VALUE constants 12
 streams of 294
 `toUnsignedInt` method 13
 type conversions of 22
 byte codes 3
 byte order mark 305
`ByteArrayXxxStream` classes 302
`ByteBuffer` class 312
 bytes
 arrays of 302
 converting to strings 306
 reading 302
 skipping 303
 writing 303

C

C# programming language, type parameters in 227
 c, C conversion characters 38
 C/C++
 `#include` directive in 92
 allocating memory in 361
 integer types in 12
 pointers in 66
 C:\ 313
`CachedRowSetImpl` 440
 calculators 166
 calendars 64
 call by reference 73
`Callable` interface
 `call` method 353
 callbacks 120, 371
 registering 369
 camel case 16
 cancel method
 of `CompletableFuture` 371
 of `Future` 353
 cancellation requests 356
`CancellationException` class 371
 cardinality 263
 carriage return 15
`case` keyword 40
`cast` 239
`cast` operator 22
 casts 22, 109, 146
 annotating 404
 generic types and 232

inserting 229
 catch keyword 197
 annotating parameters of 402
 in try-with-resources 200
 no type variables in 237
 ceiling 254
 Channel interface 112
 channels 311
 char type 14
 streams of 294
 type conversions of 22
 Character class 51
 character classes 325
 character encodings 304
 native 305
 partial 305, 309
 character literals 14
 characters 301
 formatting for output 38
 reading/writing 310
 charAt method 33
 CharSequence interface 30, 274
 chars, codePoints methods 294
 Charset class
 availableCharsets method 305
 defaultCharset method 305
 forName method 306
 checked exceptions 194, 197
 combining in a superclass 196
 declaring 196
 documenting 197
 generic types and 238
 in lambda expressions 197
 no-argument constructors and 186
 not allowed in a method 203
 rethrowing 202
 checked views 232, 268
 checkedXxx methods (Collections) 251, 269
 Checker Framework 403
 checkIndex method 205
 Child classes. See Subclasses
 Church, Alonzo 121
 Class class 174, 175, 240
 asSubclass, cast methods 239
 comparing objects of 175
 forName method 174, 175, 176, 179, 180, 195, 204
 generic 238
 getCanonicalName method 175, 176
 getClassLoader method 177
 getComponentType method 176, 188
 getConstructor(s) methods 177, 183, 186, 238, 239
 getDeclaredConstructor(s) methods 177, 183, 239
 getDeclaredField(s) methods 177
 getDeclaredMethod(s) methods 177, 185
 getDeclaringClass method 176
 getEnclosingXxx methods 176
 getEnumConstants method 239
 getField(s) methods 177, 183
 getInterfaces method 176
 getMethod(s) methods 177, 183, 185
 getModifiers method 176
 getName method 174, 176
 getPackage method 176
 getPackageName method 176
 getPermittedSubclasses method 176
 getRecordComponents method 177
 getResource method 178
 getResourceAsStream method 177, 178
 getSimpleName method 176
 getSuperclass method 176, 239
 get TypeName method 176
 getTypeParameters method 240
 isXxx methods 176, 188
 newInstance method 186, 238
 toGenericString, toString methods 176
 class declarations
 annotations in 402, 412
 initialization blocks in 76
 class files 3, 178
 paths of 87
 processing annotations in 420
 class literals 175
 no annotations for 404
 no type variables in 233
 class loaders 178, 180
 class objects 175
 class path 88, 425
 ClassCastException class 110, 232
 classes 2, 64
 abstract 109, 116, 148
 accessing from a different module 442
 adding to packages 91
 anonymous 137
 companion 114
 constructing objects of 15
 deprecated 100, 408, 409
 deserialization of 344
 documentation comments for 98, 99
 encapsulation of 423, 424
 evolving 342
 extending 142
 fields of 141
 final 148
 generic 50
 immutable 30, 362
 implementing 68, 161
 importing 91
 inner 94
 instances of 6, 68, 85
 loading 184
 local 136
 members of 141
 enumerating 167, 183
 naming 16, 86, 174
 nested 92, 403
 not known at compile time 175, 190
 protected 149
 public 90, 429
 sealed 152
 serializable 338, 339

static initialization of 179
 static methods of 85
 system 207
 testing 90
 utility 90, 180
 wrapper 51
 classes win rule 160
 classifier functions 289
ClassLoader class
 defineClass method 440
 findClass, loadClass methods 179
 setXxxAssertionStatus methods 207
 classloader inversion 180
ClassNotFoundException class 195
 CLASSPATH 89
 clear method
 of BitSet 263
 of Collection 246
 of Map 257
 clone method
 of ArrayList 163
 of Enum 165
 of Message 162, 163
 of Object 150, 155, 161, 185
 protected 161
 Cloneable interface 162
 CloneNotSupportedException class 162, 165
 cloning 161
 close method
 of AutoCloseable, Closeable 199
 of PrintWriter 198, 199
 throwing exceptions 200
 Closeable interface 112
 close method 199
 closures 132
 code 99
 code generator tools 410
 code points 30, 277, 304
 code units 14, 32, 294
 in regular expressions 326
 codePoints method
 of CharSequence 294
 Collator class 28
 collect 286, 294
 collectingAndThen method 291
Collection interface 114, 245
 add method 245
 addAll method 226, 245
 clear, contains, containsAll methods 246
 isEmpty method 246
 iterator method 247
 parallelStream method 247, 272, 295, 368
 remove, removeXxx, retainAll methods 246
 size method 246
 spliterator method 247
 stream method 247, 272
 toArray method 247
Collections class 114, 245, 249, 269
 addAll method 249
 binarySearch method 250
 branching 292
 copy method 249
 disjoint method 249
 fill method 53, 250
 frequency method 250
 generic 268
 given elements of 264
 indexOfSubList, lastIndexOfSubList methods 250
 iterating over elements of 271
 mutable 265
 nCopies method 248, 250
 processing 249
 replaceAll method 249
 reverse method 54, 250
 rotate method 250
 serializable 336
 shuffle method 54, 250
 sort method 54, 226, 227, 240, 250
 swap method 250
 synchronizedXxx methods 250
 threadsafe 367
 unmodifiable views of 265, 268
 unmodifiableXxx methods 250
 vs. streams 272
Collector interface 286
Collectors class 92
 counting method 290
 filtering method 292
 flatMapting method 291
 groupingBy method 289, 292
 groupingByConcurrent method 290, 297
 joining method 287
 mapping method 291
 maxBy, minBy methods 291
 partitioningBy method 289, 292
 reducing method 292
 summarizingXxx methods 287, 291
 summingXxx methods 290
 teeing method 292
 toCollection method 287
 toConcurrentMap method 289
 toMap method 287
 toSet method 287, 290
Collectors class
 collectingAndThen method 291
 mapping method 291
 command-line arguments 54
 comments 2
 documentation 98
 commonPool method 298, 370
 companion classes 114
 Comparable interface 117, 165, 226, 253
 compareTo method 117
 priority queues with 266
 streams of 279
 Comparator interface 92, 118, 134, 253
 comparing, comparingXxx methods 135
 naturalOrder method 136
 nullsFirst, nullsLast methods 136
 priority queues with 266

reversed method 135
reverseOrder method 136
streams of 279
thenComparing method 135
compare 118
compareTo method
 of Enum 165
 of String 28, 117
compareToIgnoreCase method 124
compareUnsigned method 21
compatibility, drawbacks of 228
compilation 3
compile 330, 335
compile-time errors 16, 111
compiler
 instruction reordering in 358
completable futures 369, 375
 combining 375
 composing 371
 interrupting 371
CompletableFuture class 369, 375
 acceptEither method 374, 375
 allOf, anyOf methods 374, 375
 applyToEither method 374, 375
 cancel method 371
 complete, completeExceptionally methods 370
 completeOnTimeout method 374
 exceptionally method 373, 374
 exceptionallyCompose method 374
 handle method 374
 isDone method 371
 orTimeout method 374
 runAfterXxx methods 374, 375
 supplyAsync method 370, 371
 thenAccept method 369, 373
 thenAcceptBoth method 374
 thenApply, thenApplyAsync methods 372, 373
 thenCombine method 374
 thenCompose method 373
 thenRun method 374
 whenComplete method 370, 373, 374
CompletionStage interface 375
compose 128
computations
 asynchronous 369
 mutator 66
 precision of 14
compute method
 of ConcurrentHashMap 364
 of Map 257
computeIfXxx methods
 of ConcurrentHashMap 364
 of Map 257
concat 279
concatenation 25
 objects with strings 156
concurrent programming 349, 391
 access errors in 134
 strategies for 361
ConcurrentHashMap class 363, 384
compute method 364
computeIfXxx methods 364
forEachXxx methods 365
keySet method 367
merge method 364
newKeySet method 367
no null values in 256
putIfAbsent method 364
reduceXxx, searchXxx methods 365
 threadsafe 380
ConcurrentModificationException class 252, 367
ConcurrentSkipListXxx classes 367
conditional operator 23
configuration files
 editing 211
 locating 178
 resolving paths for 313
confinement 361
connect 321
Console class 36
ConsoleHandler class 213, 215
constants 17, 112
 naming 17
 static 83
 using in another class 17
Constructor class 183
 getModifiers, getName methods 183
 newInstance method 186, 187
constructor references 125
 annotating 404
constructors 73
 abstract classes and 149
 annotating 236, 402, 403
 canonical, compact, custom 81
 documentation comments for 98
 executing 74
 for subclasses 144
 implementing 73
 invoking another constructor from 75
 no-argument 77, 144, 186
 overloading 74
 public 73, 183
 references in 363
Consumer interface 128
contains method (String) 29
contains, containsAll methods (Collection) 246
containsXxx methods (Map) 257
context class loaders 180
continue keyword 45, 46
control flow 39
conversion characters 37
cooperative cancellation 355
Cooperative scheduling 351
copy method
 of Collections 249
 of Files 304, 315, 320
copyOf method 52, 189
CopyOnWriteArrayList classes 367
CORBA 424
count 272, 280

counters
 atomic [376](#)
 de/incrementing [201](#)

counting [290](#)

country codes [289](#)

covariance [222](#)

createDirectory, createDirectories, createFile methods
 (Files) [315](#)

createInstance method [180](#)

createTempXxx methods (Files) [315](#)

critical sections [361, 379, 385](#)

current method
 of ProcessHandlex [392](#)
 of ThreadLocalRandom [387](#)

D

D [14](#)
 conversion character [37](#)

daemon threads [388](#)

Databases [399](#)
 persisting objects in [433](#)

DataInput/Output interfaces [310](#)
 read/writeXxx methods [310, 311](#)

DataXxxStream classes [311](#)

date classes
 immutability of [362](#)

DayOfWeek enumeration [65](#)

deadlocks [361, 380, 384, 385](#)

debugging
 messages for [194](#)
 overriding methods for [148](#)
 primary arrays for [54](#)
 streams [279](#)
 threads [387](#)
 using anonymous subclasses for [146](#)
 with assertions [206](#)

DecimalFormat class [86](#)

declaration-site variance [227](#)

decomposition
 of classes [58](#)

decrement operator [20](#)

decrementExact method [21](#)

deep copies [161](#)

deepToString method [156](#)

default keyword [114, 407](#)

default label (in switch) [40](#)

default methods [114, 116](#)
 conflicts of [115, 154](#)
 in interfaces [160](#)

defaultCharset method [305](#)

defaultReadObject method [339, 343](#)

defaultWriteObject method [339](#)

defensive programming [205](#)

deferred execution [126](#)

defineClass method [440](#)

delete, deleteIfExists methods (Files) [316](#)

delimiters, for scanners [307](#)

@deprecated annotation [100, 408, 409](#)

Deque interface [249, 265](#)

Derived classes. See Subclasses

destroy, destroyForcibly methods
 of Process [391](#)
 of ProcessHandle [393](#)

diamond syntax
 for array lists [50](#)
 for constructors of generic classes [220](#)

directories [313](#)
 checking for existence [315, 317](#)
 creating [315, 317](#)
 deleting [316, 319](#)
 moving [315](#)
 temporary [315](#)
 user [314](#)
 visiting [317](#)
 working [388](#)

directory [388](#)

disjoint [249](#)

distinct [279, 296](#)

divideUnsigned method [21](#)

division [19](#)

do [43](#)

doc-files [99](#)

documentation comments [98](#)

@Documented annotation [409, 411](#)

domain names
 for modules [425](#)
 for packages [86](#)

dot notation [6, 17](#)

double type [13, 51](#)
 atomic operations on [379](#)
 compare method [118](#)
 equals method [158](#)
 functional interfaces for [129](#)
 isFinite, isInfinite methods [14](#)
 NaN, NEGATIVE_INFINITY, POSITIVE_INFINITY values [14](#)
 parseDouble method [29](#)
 streams of [294](#)
 toString method [29](#)
 type conversions of [21](#)

double brace initialization [146](#)

DoubleAccumulator, DoubleAdder classes [379](#)

DoubleConsumer, DoubleXxxOperator, DoublePredicate,
 DoubleSupplier, DoubleToXxxFunction interfaces [129](#)

DoubleFunction interface [129, 231](#)

doubles [295](#)

DoubleStream interface [294](#)

DoubleSummaryStatistics class [287, 295](#)

downstream collectors [290, 297](#)

Driver.parentLogger method [441](#)

dropWhile method [279](#)

dynamic method lookup [145, 230, 231](#)

E

E [21](#)

e, E
 conversion characters [38](#)

Eclipse 5
 effectively final variables 133
 efficiency, and final modifier 148
 element 366, 416, 417
 Elements 400, 407
 else keyword 40
 em 99
 empty method
 of Optional 284
 of Stream 273
 empty string 27, 156
 concatenating 28
 encapsulation 64, 423, 424, 425, 433
 Encodings. See Character encodings
 end 331, 333
 End-of-line character. See Line feed
 endsWith method 29
 enhanced for loop 52, 57, 133
 for collections 251
 for enumerations 164
 for iterators 182
 for paths 314
 enhanced form 42, 171
 Entry 228
 entrySet method 258
 enum keyword 18, 164, 165
 enum instances
 adding methods to 166
 construction 166
 referred by name 168
 enumeration sets 265
 enumerations 164
 annotating 402
 comparing 164, 165
 constructing 165
 customizing serialization of 341
 defining 18
 nested inside classes 167
 serialization of 341
 static members of 167
 traversing instances of 164
 using in switch 167
 EnumMap, EnumSet classes 265
 environment variables 390
 equality, testing for 23
 equals method
 final 159
 null-safe 158
 of Arrays 158
 of Double 158
 of Object 155, 159
 of Objects 158
 of records 79
 of String 27
 of subclasses vs. superclass 158
 of wrapper classes 51
 overriding 157, 159
 symmetric 159
 values from different classes and 159
 equalsIgnoreCase method 28
 Error class 194
 error messages, for generic methods 221
 errorReader method 389
 errors
 AbstractMethodError 114
 AssertionError 206
 even numbers 20
 Exception class 194
 exceptionally 373, 374
 exceptionallyCompose method 374
 exceptions 193
 annotating 404
 ArrayIndexOutOfBoundsException 48
 ArrayStoreException 145, 223, 235
 CancellationException 371
 catching 197, 202
 chaining 202
 checked 186, 194, 197
 ClassCastException 110, 232
 ClassNotFoundException 195
 CloneNotSupportedException 162, 165
 combining in a superclass 196
 ConcurrentModificationException 252, 367
 creating 196
 documenting 197
 ExecutionException 353
 FileNotFoundException 195
 generic types and 237
 hierarchy of 194
 IllegalArgumentException 206
 IllegalStateException 287, 365
 InaccessibleObjectException 184, 434
 IndexOutOfBoundsException 205
 InterruptedException 355
 InvalidClassException 343
 InvalidPathException 313
 IOException 195, 200, 307
 NoSuchElementException 283, 366
 NullPointerException 27, 50, 68, 76, 195, 205, 255, 280
 NumberFormatException 195
 ReflectiveOperationException 175
 rethrowing 200, 203
 RuntimeException 194
 SecurityException 184
 ServletException 203
 suppressed 200
 throwing 194
 TimeoutException 353
 uncaught 204
 unchecked 194
 UncheckedIOException 307
 exec 388
 Executable class
 getModifiers method 187
 getName method 187
 getParameters method 183, 187
 ExecutableElement interface 416
 ExecutionException class 353
 Executor interface 372

executor services [351, 370](#)
 ExecutorCompletionService class [354](#)
 Executors class
 newFixedThreadPool method [351](#)
 newVirtualThreadPerTaskExecutor method [351](#)
 ExecutorService interface
 execute method [351](#)
 invokeAll, invokeAny methods [354](#)
 exhaustiveness [41](#)
 exists [315, 317](#)
 exitValue method [391](#)
 exports keyword [426, 429, 432](#)
 qualified [442](#)
 extends keyword [112, 142, 221, 226](#)
 Externalizable interface, read/writeExternal
 methods [340](#)

F

F [14, 38](#)
 factory methods [73, 85](#)
 failures, logging [202](#)
 false literal [15](#)
 as default value [75, 77](#)
 Field class [183](#)
 get method [184, 186](#)
 getBoolean, getByte, getChar, getDouble, getFloat, getInt,
 getLong methods [184, 186](#)
 getModifiers, getName methods [183, 186](#)
 getShort method [184, 186](#)
 getType method [183](#)
 set, setXxx methods [186](#)
 fields [141](#)
 enumerating [183](#)
 final [359](#)
 provided [150](#)
 public [183](#)
 retrieving values of [184](#)
 setting [185](#)
 transient [338](#)
 File class [315](#)
 file attributes
 copying [316](#)
 filtering paths by [318](#)
 file handlers [214](#)
 file pointers [311](#)
 file.separator package [261](#)
 FileChannel class
 get, getXxx methods [312](#)
 lock method [312](#)
 open method [311](#)
 put, putXxx methods [312](#)
 tryLock method [312](#)
 FileFilter interface [129](#)
 FileHandler class [213, 215](#)
 FileNotFoundException class [195](#)
 files
 archiving [320](#)
 channels to [311](#)

checking for existence [195, 315, 317](#)
 closing [198](#)
 copy method [304, 315, 320](#)
 copying [315](#)
 createTempXxx methods [315](#)
 createXxx methods [315](#)
 creating [313, 317](#)
 delete, deleteIfExists methods [316](#)
 deleting [316](#)
 empty [315](#)
 encoding of [304, 305](#)
 exists method [315, 317](#)
 find method [317, 318](#)
 isDirectory, isRegularFile methods [315, 317](#)
 lines method [274, 297, 306](#)
 list method [317](#)
 locking [312](#)
 memory-mapped [297, 311](#)
 move method [315](#)
 moving [315](#)
 newBufferedReader method [307](#)
 newBufferedWriter method [307, 316](#)
 newXxxStream methods [302, 316, 336](#)
 random-access [311](#)
 read method [303](#)
 readAllBytes method [303](#)
 readAllLines method [306](#)
 reading from/writing to [36, 195, 303](#)
 readNBytes method [303](#)
 skipNBytes method [303](#)
 temporary [315](#)
 walk method [317, 320](#)
 walkFileTree method [317, 319](#)
 write method [308, 316](#)
 FileSystem, FileSystems classes [320](#)
 FileVisitor interface [319](#)
 fill method
 of Arrays [53](#)
 of Collections [53, 250](#)
 Filter class [215](#)
 of Optional [282](#)
 of Stream [272, 276, 280](#)
 filtering [292](#)
 final keyword [17, 77, 148](#)
 final fields [359](#)
 final methods [363](#)
 final variables [359, 362](#)
 finalize [155](#)
 finally keyword [200](#)
 for locks [380](#)
 return statements in [201](#)
 financial calculations [14](#)
 find [317, 318](#)
 findAll method [332](#)
 findAny method [280](#)
 findClass method [179](#)
 findFirst method [182, 280](#)
 first [254](#)
 flag bits, sequences of [262](#)
 flatMap method

of `Optional` 284, 285
 of `Stream` 277
`flatMap` method 291
`flip` 263
 float type 13, 51
 streams of 294
 type conversions of 21
 floating-point types 13
 binary number system and 14
 comparing 118
 division of 20
 formatting for output 38
 in hexadecimal notation 14
 type conversions of 21
`floor` 254
`floorMod` method 20
`for` keyword 43, 44
 declaring variables for 47
 enhanced 52, 57, 133, 164, 251, 314
 multiple variables in 44
`forEach` method
 of `ArrayList` 124
 of `Map` 257
 of `Stream` 286
`forEachOrdered` method 286
`forEachXxx` methods (`ConcurrentHashMap`) 365
`ForkJoinPool` class 372
 common pool method 298, 370
 format specifiers 37
 formatted 39
 formatted output 37
`Formatter` class 215
 forms, posting data from 322, 324
`forName` method
 of `Charset` 306
 of `Class` 174, 175, 176, 179, 180, 195, 204
 frequency 250
`Function` interface 128, 287
 function types 121, 127
 functional interfaces 123, 409, 410
 as method parameters 224
 common 128
 contravariant in parameter types 225
 for primitive types 129
 implementing 130
`@FunctionalInterface` annotation 130, 409, 410, 411
 functions 63
 higher-order 134
 Functions. See Methods
`Future` interface 354
 cancel, `isCancelled`, `isDone` methods 353
 get method 353, 369
 futures 353
 completable 369, 375

G

`g`, `G`
 conversion characters 38

gadget chains 345
 garbage collector 267
 generate 273, 294
`@Generated` annotation 409, 410
 generators, converting to streams 296
 generic classes 50, 220
 constructing objects of 220
 information available at runtime 239
 instantiating 220
 generic collections 268
 generic constructors 240
 generic methods 220
 calling 221
 declaring 221
 information available at runtime 239
 generic type declarations 240, 241
 generic types 117
 annotating 403
 arrays of 126
 casting 232
 exceptions and 237
 in JVM 228
 invariant 223, 225
 lambda expressions and 225
 reflection and 238
 restrictions on 231
`GenericArrayType` interface 240
`get` method
 of `Array` 189
 of `ArrayList` 51
 of `BitSet` 263
 of `Field` 184, 186, 312
 of `Future` 353, 369
 of `List` 248
 of `LongAccumulator` 378
 of `Map` 255, 256
 of `Optional` 283, 285
 of `Path` 314
 of `ServiceLoader.Provider` 182
 of `Supplier` 128
 GET requests 323
`getAndXxx` methods (`AtomicXxx`) 377
`getAnnotation`, `getAnnotationsByType` methods
 of `AnnotatedConstruct` 417
 of `AnnotatedElement` 413, 415
`getAsXxx` methods
 of `OptionalXxx` 295
 of `XxxSupplier` 129
`getAudioClip` method 178
`getAverage` method 287
`getBoolean` method
 of `Array` 189
 of `Field` 184, 186
 of `FileChannel` 312
`getByte` method
 of `Array` 189
 of `Field` 184, 186
 of `FileChannel` 312
`getCanonicalName` method 175, 176
`getChar` method

of Array [189](#)
of Field [184](#), [186](#)
of FileChannel [312](#)
getClass method [148](#), [155](#), [158](#), [174](#), [233](#), [238](#)
getClassLoader method [177](#)
getComponentType method [176](#), [188](#)
getConstructor(s) methods (Class) [177](#), [183](#), [186](#), [238](#), [239](#)
getContextClassLoader method [180](#)
getCountry method [289](#)
getCurrencyInstance method [85](#)
getDayOfMonth methods
 of LocalDate [65](#)
getDeclaredAnnotationXxx methods
 (AnnotatedElement) [413](#), [415](#)
getDeclaredConstructor(s) methods (Class) [177](#), [183](#), [239](#)
getDeclaredField(s) methods (Class) [177](#)
getDeclaredMethod(s) methods (Class) [177](#), [185](#)
getDeclaringClass method
 of Class [176](#)
 of Enum [165](#)
getDefault method
 of RandomGenerator [7](#)
getDouble method
 of Array [189](#)
 of Field [184](#), [186](#)
 of FileChannel [312](#)
getElementsAnnotatedWith method [417](#)
getEnclosedElements method [417](#)
getEnclosingXxx methods (Class) [176](#)
getEnumConstants method [239](#)
getErrorStream method [389](#), [390](#)
getField(s) methods (Class) [177](#), [183](#)
getFileName method [314](#)
getFilePointer method [311](#)
getFloat method
 of Array [189](#)
 of Field [184](#), [186](#)
 of FileChannel [312](#)
getHead method [215](#)
getHeaderFields method [321](#)
getInputStream method
 of Process [389](#)
 of URL [321](#)
 of URLConnection [322](#)
getInstant method [216](#)
getInt method
 of Array [189](#)
 of Field [184](#), [186](#)
 of FileChannel [312](#)
getInterfaces method [176](#)
getLength method [189](#)
getLevel method [216](#)
getLogger method [208](#), [210](#)
getLoggerName method [216](#)
getLong method
 of Array [189](#)
 of Field [184](#), [186](#)
 of FileChannel [312](#)
getLongThreadID method [216](#)
getMax method [287](#)
getMessage method [216](#)
getMethod(s) methods (Class) [177](#), [183](#), [185](#)
getModifiers method
 of Class [176](#)
 of Constructor [183](#)
 of Executable [187](#)
 of Field [183](#), [186](#)
 of Method [183](#)
getMonthValue method
 of LocalDate [65](#)
getName method
 of Class [174](#), [176](#)
 of Constructor [183](#)
 of Executable [187](#)
 of Field [183](#), [186](#)
 of Method [183](#)
 of Parameter [187](#)
 of Path [314](#)
 ofPropertyDescriptor [188](#)
 of System.Logger [210](#)
getOrDefault method [255](#), [256](#)
getOutputStream method
 of Process [389](#)
 of URLConnection [321](#)
getPackage method [176](#)
getPackageName method [176](#)
getParameters method
 of Executable [183](#), [187](#)
 of LogRecord [216](#)
getParent method [314](#)
getPath method [320](#)
getPercentInstance method [85](#)
getPermittedSubclasses method [176](#)
getProperties method [261](#)
getProperty method [179](#), [205](#), [261](#)
getPropertyDescriptors method [188](#)
getPropertyType, getReadMethod methods
 (PropertyDescriptor) [188](#)
getQualifiedName method [417](#)
getRecordComponents method [177](#)
getResource method [178](#)
getResourceAsStream method
 of Class [177](#), [178](#)
 of Module [435](#)
getResourceBundle, getResourceBundleName methods
 (LogRecord) [216](#)
getRoot method [314](#)
getSequenceNumber method [216](#)
getShort method
 of Array [189](#)
 of Field [184](#), [186](#)
 of FileChannel [312](#)
getSimpleName method
 of Class [176](#)
 of Element [417](#)
getSourceXxxName methods (LogRecord) [216](#)
getSuperclass method [176](#), [239](#)
getSuppressed method [200](#)

getTail method 215
 Getter/setter pairs. See Properties
 getThrown method 216
 getType method
 of Field 183
 of Parameter 187
 getTypeName method 176
 getTypeParameters method 240
 getURLs method 179
 getValue method 65
 getWriteMethod method 188
 Goetz, Brian 349
 graphemeClusters method
 of String 279
 group 331, 333
 grouping 289
 classifier functions of 289
 reducing to numbers 290
 groupingBy method 289, 292
 groupingByConcurrent method 290, 297
 GUI
 callbacks in 120
 long-running tasks in 375

H

h, H conversion characters 38
 handle 374
 Hansen, Per Brinch 382
 hash 160
 hash codes 159
 computing in String class 160
 formatting for output 38
 hash functions 159, 253
 hash maps
 concurrent 363
 weak 267
 hash tables 252
 hashCode method
 of Arrays 160
 of Enum 165
 of Object 155, 157, 159
 of records 79
 HashMap class 255
 null values in 256
 HashSet class 252
 readObject, writeObject methods 339
 Hashtable class 382
 hasNext method
 declaring 107
 of Iterator 251
 of Scanner 36, 307
 hasNextXxx methods (Scanner) 36, 307
 headMap method 268
 headSet method
 of NavigableSet 254
 of SortedSet 254, 268
 heap pollution 232, 268
 Hello, World! program 1

modular 426
 helper methods 228
 hexadecimal numbers 13, 14
 formatting for output 38
 higher 254
 higher-order functions 134
 hn, hr elements (HTML) 99
 Hoare, Tony 382
 HTML
 generating documentation in 419
 including code in 34
 HTTP connections 321
 HTTP/2 support 321
 HttpClient class 321, 324
 enabling logging for 324
 newBuilder, newHttpClient methods 322, 370
 HttpHeaders class 324
 HttpResponse interface 323, 324
 HttpURLConnection class 321
 hyperlinks
 in documentation comments 101
 regular expressions for 325

I

IDE 3, 5
 identity method
 of Function 128, 287
 of UnaryOperator 128
 identity values 293
 if keyword 39, 40
 ifPresent, ifPresentOrElse methods (Optional) 281
 IllegalArgumentException class 206
 IllegalStateException class 287, 365
 ImageIcon class 178
 images, locating 178
 img 99
 immutability 362
 immutable classes 362
 implements keyword 107, 108
 import keyword 8, 91
 no annotations for 404
 static 92
 import static 168
 InaccessibleObjectException class 184, 434
 increment 378
 increment operator 20
 incrementAndGet method 377
 incrementExact method 21
 indexOf method
 of List 248
 of String 29
 indexOfSubList method 250
 IndexOutOfBoundsException class 205
 info 392
 Information hiding. See Encapsulation
 inheritance 142, 163
 classes win rule 154, 160
 default methods and 154

@Inherited annotation [409, 412](#)
 initCause method [203](#)
 initialization blocks [76](#)
 static [84](#)
 inlining [148](#)
 inner classes [94](#)
 anonymous [137](#)
 invoking methods of outer classes [96](#)
 local [133, 136](#)
 syntax for [97](#)
 input
 reading [35, 306](#)
 splitting along delimiters [333](#)
 input prompts [37](#)
 input streams [301](#)
 copying [304](#)
 obtaining [302](#)
 reading from [302](#)
 inputReader method [389](#)
 InputStream class [302](#)
 transferTo method [304](#)
 InputStreamReader class [306](#)
 INSTANCE [341](#)
 instance methods [6, 70](#)
 instance variables [68, 71](#)
 abstract classes and [149](#)
 annotating [402](#)
 comparing [158](#)
 default values of [75](#)
 final [77](#)
 in records [79, 81](#)
 initializing [76, 144](#)
 not accessible from static methods [85](#)
 of serialized objects [341, 343](#)
 protected [149](#)
 setting [74](#)
 transient [338](#)
 vs. local [75](#)
 instanceof keyword [110, 146, 158, 159](#)
 annotating [404](#)
 with pattern matching [110](#)
 instances [2, 6](#)
 instruction reordering [358](#)
 int type [12](#)
 functional interfaces for [129](#)
 processing values of [127](#)
 random number generator for [7](#)
 streams of [294](#)
 type conversions of [21](#)
 using class literals with [175](#)
 IntBinaryOperator interface [129](#)
 IntConsumer interface [127, 129](#)
 Integer class [51](#)
 compare method [118](#)
 MAX_VALUE, MIN_VALUE constants [12](#)
 parseInt method [29, 195](#)
 toString method [28](#)
 unsigned division in [13](#)
 xxxUnsigned methods [21](#)
 integer types [12](#)
 comparing [118](#)
 computing [20, 21](#)
 formatting for output [37](#)
 in hexadecimal notation [13](#)
 reading/writing [310, 311](#)
 type conversions of [21](#)
 values of
 even/odd [20](#)
 signed [13](#)
 interface keyword [107, 405, 406, 407](#)
 sealed [152](#)
 interface methods [114, 116](#)
 interfaces [106](#)
 annotating [402, 403](#)
 compatibility of [114](#)
 declarations of [106](#)
 defining variables in [112](#)
 documentation comments for [98](#)
 evolution of [114](#)
 extending [112](#)
 functional [123, 409, 410](#)
 implementing [107](#)
 multiple [112](#)
 methods of [107, 108](#)
 nested, enumerating [183](#)
 no instance variables in [113](#)
 no redefining methods of the Object class in [160](#)
 views of [267](#)
 interrupted [356](#)
 interrupted status [356](#)
 InterruptedException class [355](#)
 intersects [264](#)
 IntFunction interface [129, 231](#)
 IntPredicate interface [129](#)
 intrinsic locks [380](#)
 ints [295](#)
 IntSequence [108, 137](#)
 IntStream interface [294](#)
 parallel method [295](#)
 IntSummaryStatistics class [287, 295](#)
 IntSupplier, IntToXxxFunction,
 IntUnaryOperator interfaces [129](#)
 InvalidClassException class [343](#)
 InvalidPathException class [313](#)
 InvocationHandler interface [190](#)
 invoke [185, 187](#)
 invokeAll, invokeAny methods (ExecutorService) [354](#)
 IOException class [195, 307](#)
 addSuppressed, getSuppressed methods [200](#)
 isAbstract method [177, 183](#)
 isAlive method
 of Process [391](#)
 of ProcessHandle [393](#)
 of Thread [385](#)
 isAnnotation method [176](#)
 isAnonymousClass method [176](#)
 isArray method [176, 188](#)
 isAssignableFrom method [176](#)
 isCancelled method [353](#)
 isDirectory method [315, 317](#)

isDone method
 of CompletableFuture 371
 of Future 353
 isEmpty method
 of BitSet 264
 of Collection 246
 of Map 257
 isEnum method 176
 isEqual method 129
 isFinite, isInfinite methods (Double) 14
 isInstance method 176
 isInterface method 177, 183
 isInterrupted method 356
 isLocalClass method 176
 isLoggable method
 of Filter 215
 of System.Logger 210
 isMemberClass method 176
 isNamePresent method 187
 isNative method 177, 183
 isNull method 124
 ISO 8601 format 410
 ISO 8859-1 encoding 305, 309
 isPresent method 283, 285
 isPrimitive method 176
 isPrivate, isProtected, isPublic methods (Modifier) 177, 183
 isRecord method 176
 isRegularFile method 315, 317
 isSealed method 176
 isStatic, isStrict, isSynchronized methods (Modifier) 177, 183
 isSynthetic method 176
 isVolatile method 177, 183
 Iterable interface 251, 314
 iterator method 251
 iterate 274, 279, 294, 368
 iterator method
 next, hasNext methods 251
 of Collection 247
 of ServiceLoader 182
 of Stream 286
 remove, removeIf methods 251
 iterators 251, 286
 converting to streams 275, 296
 invalid 252
 traversing 182
 weakly consistent 367

J

jar 87
 --module-version option 436
 -C option 436
 -d option 436
 JAR files 87
 dependencies in 444
 for split packages 436
 manifest for 438

modular 436
 processing order of 89
 resources in 178
 scanning for deprecated elements 409
 Java 3
 --add-exports, --add-opens options 440
 --add-module option 437
 --illegal-access option 439
 -cp (--class-path, -classpath) option 89
 -d (disableassertions) option 207
 -ea (enableassertions) option 206
 -esa (enablesystemassertions) option 207
 -m, -p (--module, --module-path) options 426, 436
 compatibility with older versions of 153, 154, 228
 online API documentation on 30, 32
 option files for 440
 option names in 88
 strongly typed 15
 Unicode support in 30
 uniformity of 2, 116
 Java Persistence Architecture 399
 Java Platform Module System 423
 layers in 437
 migration to 437, 439
 no support for versioning in 425, 427, 436
 service loading in 443
 java.awt package 90, 424
 java.awt.geom package 338
 java.base package 428
 java.class.path package 261
 java.desktop package 428
 java.home package 261
 java.io.tmpdir package 261
 java.lang, java.lang.annotation packages 408
 java.lang.reflect package 183
 java.logging package 441
 java.time package
 immutability of classes 362
 java.util package 8, 367
 java.util.concurrent package 363, 366
 java.util.concurrent.atomic package 376
 java.util.logging package 207, 211
 java.util.random package 106
 java.version package 261
 JavaBeans 187
 javac 3
 -author option 102
 -cp (--class-path, -classpath) option 89
 -d option 87, 102
 -link, -linksources options 102
 -parameters option 183
 -processor option 416
 -version option 102
 -XprintRounds option 419
 javadoc 98
 including annotations in 411
 JavaFX 120, 376
 javan.log files 213
 JavaServer Faces framework 259
 javax.annotation package 408

javax.swing package 428
 JAXB 433
 JCommander 399
 jconsole 213
 jdeprscan 409
 jdeps 444
 JDK 3
 obsolete features in 424
 JEP 246 (platform logging API) 207
 jlink 445
 jmod 446
 job scheduling 266
 join method
 of String 26
 of Thread 385
 joining 287
 JPA 433
 JShell 8
 imported packages in 11
 loading modules into 437
 JSON 150
 JUnit 399, 400

K

key/value pairs
 in annotations 400, 407
 in maps 254
 removed by garbage collector 267
 Key/value pairs. See Properties
 keySet method
 of ConcurrentHashMap 367
 of Map 258, 267
 keywords 16
 contextual 153

L

L 13
 L64X128MixRandom 106
 labeled statements 45, 46
 lambda expressions 121
 annotating targets for 410
 capturing variables in 132
 executing 127
 for loggers 209
 generic types and 225
 parameters of 122
 processing 126
 return type of 122
 scope of 131
 this reference in 131
 throwing exceptions in 197
 using with streams 276, 368
 language codes 289
 language model API 416
 last 254
 lastIndexOf method
 of List 248
 of String 29
 lastIndexOfSubList method 250
 lazy operations 272, 276, 279, 333
 length method
 of arrays 48
 of RandomAccessFile 311
 of String 7, 33
 .level 212
 lib/modules 446
 limit 278, 297
 line feed 34
 character literal for 15
 formatting for output 38
 in regular expressions 329
 line.separator package 262
 lines method
 of BufferedReader 307
 of Files 274, 297, 306
 @link annotation 101
 linked lists 248, 252
 LinkedBlockingQueue class 366, 384
 LinkedHashMap class 259
 LinkedList class 248
 List class 226, 247, 248, 317
 add, addAll, get, indexOf, lastIndexOf, listIterator methods 247
 of method 51, 53, 248, 264
 remove, replaceAll, set, sort methods 248
 subList method 248, 267
 ListIterator interface 252
 lists
 converting to streams 296
 mutable 265
 printing elements of 124
 removing null values from 124
 sublists of 267
 unmodifiable views of 268
 literals
 character 14
 floating-point 14
 integer 13
 string 27, 33
 little-endian format 305
 load 182, 444
 load balancing 336
 loadClass method 179
 local classes 136
 local variables 46
 annotating 402, 403
 vs. instance 75
 LocalDate class 64
 getXxx methods 65
 now method 73, 85
 of method 64, 73
 plus, plusXxx methods 65, 66, 68
 Locale class 288
 getCountry method 289
 locales 288, 291
 LocalTime class
 final 148

lock method
 of `FileChannel` 312
 of `ReentrantLock` 380
 locks 361
 error-prone 362
 intrinsic 380
 reentrant 379
 releasing 201, 359
 log handlers 213
 filtering/formatting 215
 Log4j 207
 Logback 207
 Logger class (`java.util.logging`) 441
 Logger interface (`System`) 208, 211
 `getName` method 210
 `isLoggable` method 210
 `log` method 208, 210
 loggers
 filtering/formatting 215
 hierarchy of 212
 naming 208
 logging 207
 configuring 211, 213
 failures 202
 levels of 209, 213
 overriding methods for 148
 LogRecord class, methods of 216
 long type 12, 51
 atomic operations on 377, 379
 functional interfaces for 129
 MAX_VALUE, MIN_VALUE constants 12
 streams of 294
 type conversions of 21
 unsigned division in 13
 xxxUnsigned methods 21
 long-term persistence 342
 LongAccumulator class 377
 `accumulate`, `get` methods 378
 LongAdder class 377, 379
 `add`, `increment`, `sum` methods 378
 threadsafe 380
 LongConsumer, LongXxxOperator, LongPredicate, LongSupplier, LongToXxxFunction interfaces 129
 LongFunction interface 129, 231
 longs 295
 LongStream interface 294
 LongSummaryStatistics class 287, 295
 Lookup 435
 lookup method (`MethodHandles`) 435
 loops 43
 exiting 44
 infinite 44
 lower 254

M

main 2, 6
 decomposing 58
 string array parameter of 54

Map interface 249
 `clear` method 257
 `compute` method 257
 `computeIfXxx` methods 257
 `containsXxx` methods 257
 `entrySet` method 258
 `forEach` method 257
 `get`, `getOrDefault` methods 255, 256
 `isEmpty` method 257
 `keySet` method 258, 267
 `merge` method 255, 256
 `of` method 258, 264
 `of Optional` 282
 `of Stream` 276
 `ofEntries` method 264
 `put` method 254, 256
 `putAll` method 257
 `putIfAbsent` method 256
 `remove` method 257
 `replace`, `replaceAll` methods 257
 `size` method 257
 `values` method 258, 267
 mapping method 291
 `of Collectors` 291
 maps 254
 concurrent 257, 289
 empty 257
 iterating over 258
 of stream elements 287, 297
 order of elements in 259
 views of 258
 unmodifiable 268
 mapToInt method 293
 mapToXxx methods (`XxxStream`) 295
 marker interfaces 162
 Matcher class 330, 333
 methods of 334
 matcher, matches methods (`Pattern`) 330
 MatchResult interface 331, 335
 Math class
 E constant 21
 `floorMod` method 20
 `max`, `min` methods 21
 PI constant 21, 83, 92
 `pow` method 21, 84, 92
 `round` method 22
 `sqrt` method 21
 TAU constant 21
 xxxExact methods 21, 23
 max method
 `of Stream` 280
 `of XxxStream` 295
 MAX_VALUE 12
 maxBy method
 `of BinaryOperator` 129
 `of Collectors` 291
 memory
 allocating 361
 caching 358
 concurrent access to 359

memory-mapped files [311](#)
 merge method
 of ConcurrentHashMap [364](#)
 of Map [255, 256](#)
 Message [162, 163](#)
 meta-annotations [406, 412](#)
 META-INF/MANIFEST.MF [438](#)
 META-INF/services [443](#)
 Method class [183](#)
 getModifiers, getName methods [183](#)
 invoke method [185, 187](#)
 method calls [6](#)
 receiver of [70](#)
 method expressions [124, 147](#)
 method references [124, 233](#)
 annotating [404](#)
 MethodHandles.lookup method [435](#)
 methods [2](#)
 abstract [123, 148](#)
 accessor [66, 79](#)
 annotating [236, 402](#)
 atomic [364](#)
 body of [69](#)
 chaining calls of [65](#)
 clashes of [236](#)
 compatible [160](#)
 declarations of [69](#)
 default [114, 116](#)
 deprecated [100, 408, 409](#)
 documentation comments for [98, 99](#)
 enumerating [183](#)
 factory [73, 85](#)
 final [148, 363](#)
 for throwing exceptions [204](#)
 header of [69](#)
 inlining [148](#)
 instance [70](#)
 invoking [185](#)
 modifying functions [135](#)
 mutator [66, 268, 363](#)
 naming [16, 79](#)
 native [84](#)
 overloading [74, 125](#)
 overriding [114, 143, 148, 197, 408, 409](#)
 parameters of [183](#)
 null checks for [204](#)
 passing arrays into [58](#)
 private [116](#)
 proxied [191](#)
 public [107, 108, 183](#)
 restricted to subclasses [149](#)
 return value of [2, 69](#)
 returning functions [134](#)
 static [58, 84, 85, 92, 113](#)
 storing in variables [7](#)
 symmetric [159](#)
 synchronized [381, 384](#)
 used for serialization [408, 410](#)
 utility [90](#)
 variable number of arguments of [59](#)

Microsoft Notepad [305](#)
 Microsoft Windows
 line ending in [34](#)
 path separator in [89, 262](#)
 min method
 of Math [21](#)
 of Stream [280](#)
 of XxxStream [295](#)
 MIN_VALUE [12](#)
 minBy method
 of BinaryOperator [129](#)
 of Collectors [291](#)
 Modifier class
 isXxx methods [177, 183](#)
 toString method [177](#)
 modifiers, checking [183](#)
 module keyword [426](#)
 module path [426, 436, 438](#)
 module-info.class [426, 436](#)
 module-info.java [426](#)
 Module.getResourceAsStream method [435](#)
 modules [423](#)
 aggregator [441](#)
 annotating [427](#)
 automatic [437, 439](#)
 bundling up the minimal set of [445](#)
 declaration of [425, 426](#)
 documentation comments for [98, 102](#)
 explicit [439](#)
 illegal access to [439](#)
 inspecting files in [446](#)
 loading into JShell [437](#)
 naming [425, 438](#)
 open [434](#)
 reflective access for [184, 185](#)
 required [427, 440](#)
 tools for [444](#)
 transitive [440](#)
 unnamed [438](#)
 versioning and [425, 427, 436](#)
 monitors (classes) [382](#)
 move [315](#)
 multiplication [19](#)
 mutators [66](#)
 unmodifiable views and [268](#)

N

n
 conversion character [38](#)
 name [165](#)
 NaN [14](#)
 native encoding [305](#)
 native methods [84](#)
 naturalOrder method [136](#)
 navigable maps/sets [268](#)
 NavigableMap interface [367](#)
 NavigableSet interface [249, 253, 268](#)
 methods of [254](#)

nCopies method 248, 250
 negate 129
 negateExact method 21
 negative values 12
 NEGATIVE_INFINITY 14
 nested classes 92
 annotating 403
 enumerating 183
 inner 94
 public 93
 static 93
 new keyword 7, 15, 18, 74
 as constructor reference 125
 for anonymous classes 138
 for arrays 48, 49, 56
 newBufferedReader method 307
 newBufferedWriter method 307, 316
 newBuilder method 322, 370
 newFileSystem method 320
 newFixedThreadPool method
 of Executors 351
 newHttpClient method 322, 370
 newInputStream method 302, 316, 336
 newInstance method
 of Array 189
 of Class 186, 238
 of Constructor 186, 187
 newKeySet method 367
 newOutputStream method 302, 316, 336
 newProxyInstance method 190
 newVirtualThreadPerTaskExecutor method
 of Executors 351
 next method
 declaring 107
 of Iterator 251
 of Scanner 35
 nextClearBit method 263
 nextDouble method
 common for all generators 106
 of Scanner 35, 307
 nextInt method
 common for all generators 106
 of Random 7
 of Scanner 35
 nextLine method 35
 nextSetBit method 263
 nominal typing 127
 non-sealed keyword 152
 noneMatch method 281
 noneOf method 265
 noninterference, of stream operations 276
 @NonNull annotation 403
 normalize 314
 NoSuchElementException class 283, 366
 notify, notifyAll methods (Object) 384
 now method
 of LocalDate 73, 85
 null literal 27, 67
 as default value 75, 77
 checking parameters for 204
 comparing against 158
 converting to strings 156
 NullPointerException class 27, 50, 68, 76, 195, 205, 255
 vs. Optional 280
 nullsFirst, nullsLast methods (Comparator) 136
 NumberFormat class
 getXXXInstance methods 85
 NumberFormatException class 195
 numbers
 average of 107, 108
 big 24
 comparing 118
 converting to strings 28
 default value of 75, 77
 even or odd 20
 formatting 37
 from grouped elements 290
 in regular expressions 327
 non-negative 206, 262
 printing 37
 random 7, 106, 273, 278, 295
 reading/writing 307, 310, 311
 rounding 14, 22
 type conversions of 21
 unsigned 13, 21
 with fractional parts 13

O

o 38
 Object class 154
 clone method 150, 155, 161, 185
 equals method 155, 159
 finalize method 155
 getClass method 148, 155, 158, 174, 233, 238
 hashCode method 155, 157, 159
 notify, notifyAll methods 384
 toString method 155, 157
 wait method 383, 384
 object references 66
 attempting to change 72
 comparing 157
 default value of 75, 77
 null 67
 passed by value 73
 serialization and 337
 object-oriented programming 63
 encapsulation in 423, 424
 object-relational mappers 433
 ObjectInputStream class 336, 337
 defaultReadObject method 339, 343
 readDouble method 339
 readFields method 343
 readObject method 337, 345
 ObjectInputValidation interface 344, 345
 ObjectOutputStream class 336
 defaultWriteObject method 339
 writeDouble method 339

writeObject method [336](#), [339](#)
 objects [2](#), [63](#)
 calling methods on [7](#)
 casting [109](#)
 checkIndex method [205](#)
 cloning [161](#)
 comparing [51](#), [157](#)
 constructing [7](#), [73](#), [186](#)
 converting
 to JSON [433](#)
 to strings [155](#)
 converting to streams [274](#)
 deep/shallow copies of [161](#), [163](#)
 deserialized [341](#), [343](#)
 equals method [158](#)
 hash method [160](#)
 immutable [66](#)
 initializing variables with [15](#)
 inspecting [184](#)
 invoking static methods on [85](#)
 isNull method [124](#)
 mutable [77](#)
 requireNonNull, requireNonNullXxx methods [204](#)
 serializable [336](#)
 sorting [117](#)
 state of [63](#)
 ObjXxxConsumer interfaces [129](#)
 octal numbers [13](#)
 formatting for output [38](#)
 octonions [30](#)
 odd numbers [20](#)
 of method
 of EnumSet [265](#)
 of IntStream [294](#)
 of List [51](#), [53](#), [248](#), [264](#)
 of LocalDate [64](#), [73](#)
 of Map [258](#), [264](#)
 of Optional [284](#)
 of Path [313](#), [314](#), [320](#)
 of ProcessHandle [392](#)
 of Set [264](#)
 of Stream [273](#)
 ofEntries method [264](#)
 offer [366](#)
 ofNullable method
 of Optional [284](#)
 of Stream [274](#), [286](#)
 ofString method [323](#)
 onExit method
 of Process [391](#)
 of ProcessHandle [393](#)
 open keyword [311](#)
 open keyword [435](#)
 openConnection method [321](#)
 opens keyword [434](#)
 qualified [442](#)
 openStream method [302](#)
 Operation [166](#)
 operations
 associative [293](#)
 atomic [361](#), [364](#), [376](#), [382](#)
 bulk [365](#)
 lazy [272](#), [276](#), [279](#), [333](#)
 parallel [368](#)
 performed optimistically [377](#)
 stateless [296](#)
 threadsafe [363](#)
 operators [18](#)
 cast [22](#)
 precedence of [19](#)
 option files [440](#)
 Optional class [280](#), [285](#)
 creating values of [284](#)
 empty method [284](#)
 filter method [282](#)
 flatMap method [284](#), [285](#)
 for empty streams [292](#), [293](#)
 for processes [392](#)
 get method [283](#), [285](#)
 ifPresent method [281](#)
 ifPresentOrElse method [282](#)
 isPresent method [283](#), [285](#)
 map method [282](#)
 of, ofNullable methods [284](#)
 or method [282](#)
 orElse method [280](#)
 orElseThrow method [281](#), [283](#)
 proper usage of [283](#)
 stream method [285](#)
 OptionalXxx classes [295](#)
 or method
 of BitSet [263](#)
 of Predicate, BiPredicate [129](#)
 order [312](#)
 ordinal [165](#)
 orElseThrow method [281](#), [283](#)
 org.omg.corba package [424](#)
 orTimeout method [374](#)
 os.arch, os.name, os.version system properties [261](#)
 OSGi [424](#)
 output
 formatted [37](#)
 writing [307](#)
 output streams [301](#)
 closing [303](#)
 obtaining [302](#)
 writing to [303](#)
 OutputStream class [336](#)
 write method [303](#)
 OutputStreamWriter class [307](#)
 outputWriter method [389](#)
 @Override annotation [144](#), [338](#), [341](#), [408](#), [409](#)
 overriding [143](#)
 for logging/debugging [148](#)
 overview.html package [102](#)

P

package keyword [87](#)

package declarations [86](#)
 package-info.java [102, 402](#)
 packages [2, 86](#)

- accessing [90, 150, 424, 430, 431, 434, 437](#)
- adding classes to [91](#)
- annotating [402, 403](#)
- default [87](#)
- documentation comments for [98, 102](#)
- exporting [429, 435](#)
- naming [86](#)
- not nesting [86](#)
- split [436](#)

 parallel [295](#)
 parallel streams [368](#)
 parallelStream method [247, 272, 295, 368](#)
 parallelXxx methods (Arrays) [54, 369](#)
 @param annotation [99](#)
 Parameter class [187](#)
 parameter variables [72](#)

- annotating [402](#)
- scope of [47](#)

 Parameterized types. See Type parameters
 ParameterizedType interface [240](#)
 Parent classes. See Superclasses
 parentLogger method (Driver) [441](#)
 parseDouble method [29](#)
 parseInt method [29, 195](#)
 partitioning [362](#)
 partitioningBy method [289, 292](#)
 Pascal triangle [56](#)
 passwords [36](#)
 Path interface [114, 313](#)

- get method [314](#)
- getXxx methods [314](#)
- normalize method [314](#)
- of method [313, 314, 320](#)
- relativize method [314](#)
- resolve, resolveSibling methods [313](#)
- subpath method [314](#)
- toAbsolutePath, toFile methods [314](#)

 path separators [313](#)
 path.separator package [262](#)
 Paths class [114, 313](#)

- absolute vs. relative [313, 314](#)
- combining [314](#)
- filtering [318](#)
- resolving [313](#)
- taking apart [314](#)

 Pattern class

- asMatchPredicate, asPredicate methods [331](#)
- compile method [330, 335](#)
- flags [335](#)
- matcher, matches methods [330](#)
- split method [333](#)
- splitAsStream method [274, 333](#)
- splitWithDelimiters method [333](#)

 pattern variables [215](#)
 Pattern.quote method [326](#)
 PECS (producer extends, consumer super) [225](#)
 peek method

- of BlockingQueue [366](#)
- of Stream [279](#)

 performance

- atomic operations and [377](#)
- big numbers and [24](#)
- combined operators and [20](#)
- memory caching and [358](#)

 permits keyword [152](#)
 @Persistent annotation [412](#)
 PI [21, 83, 92](#)
 Picocli [399](#)
 platform class loader [178](#)
 platform logging API [207, 212](#)
 Platform threads [350](#)
 plugins, loading [179](#)
 plus, plusXxx methods

- of LocalDate [65, 66, 68](#)

 Point class [155](#)
 poll [366](#)
 pollXxx methods (NavigableSet) [254](#)
 pools, for parallel streams [298](#)
 pop [266](#)
 POSITIVE_INFINITY [14](#)
 POST requests [323](#)
 postVisitDirectory method [319](#)
 pow [21, 84, 92](#)
 predefined character classes [325, 327, 329](#)
 Predicate interface [123, 129](#)

- and method [129](#)
- isEqual method [129](#)
- or, negate methods [129](#)
- test method [129, 224](#)

 predicate functions [289](#)
 previous method

- of ListIterator [252](#)

 previousClearBit method [263](#)
 previousSetBit method [263](#)
 preVisitDirectory method [319](#)
 primitive types [12](#)

- comparing [158](#)
- converting to strings [156](#)
- functions interfaces for [129](#)
- passed by value [73](#)
- streams of [293, 295](#)
- type parameters and [231](#)
- variables of, no updating for [72](#)
- wrapper classes for [51](#)

 printStackTrace method [204](#)
 PrintStream class [6, 156, 308](#)

- print method [6, 37, 207, 308](#)
- printf method [37, 38, 59, 308](#)
- println method [6, 7, 35, 37, 54, 124, 308](#)

 PrintWriter class [308](#)

- close method [198, 199](#)
- print method [308](#)
- printf method [308](#)
- println method [308](#)

 priority queues [266](#)
 private keyword [2, 90](#)

- for enum constructors [166](#)

Process class [388](#)
 destroy, destroyForcibly methods [391](#)
 errorReader method [389](#)
 exitValue method [391](#)
 getErrorStream method [389, 390](#)
 getInputStream, getOutputStream methods [389](#)
 inputReader method [389](#)
 isAlive method [391](#)
 onExit method [391](#)
 outputWriter method [389](#)
 supportsNormalTermination method [391](#)
 toHandle method [392](#)
 waitFor method [391](#)

ProcessBuilder class [388](#)
 directory method [388](#)
 redirectXxx methods [389, 390](#)
 start, startPipeline methods [390](#)

processes [388](#)
 building [388](#)
 getting info about [392](#)
 killing [391](#)
 running [390](#)

ProcessHandle interface [392](#)
 allProcesses method [392](#)
 current method [392](#)
 destroy, destroyForcibly methods [393](#)
 info method [392](#)
 isAlive method [393](#)
 of method [392](#)
 onExit method [393](#)
 supportsNormalTermination method [393](#)

processing pipeline [371, 390](#)

Processor interface [416](#)

programming languages
 functional [105](#)
 object-oriented [2](#)

programs
 compiling [3](#)
 configuration options for [260](#)
 packaging [446](#)
 responsive [375](#)
 running [3](#)
 testing [206](#)

promises (in concurrent libraries) [370](#)

properties [187, 260](#)
 loading from file [261](#)
 naming [188](#)
 read-only/write-only [187](#)
 testing for [224](#)

property files
 encoding [261](#)
 generating [419](#)

protected keyword [149](#)

Provider.get, Provider.type methods [182](#)

provides keyword [444](#)

Proxy class [190](#)
 newProxyInstance method [190](#)

public keyword [2, 90](#)
 for interface methods [107, 108](#)
 method overriding and [144](#)

push [266](#)

put method
 of BlockingQueue [365](#)
 of FileChannel [312](#)
 of Map [254, 256](#)

putAll method [257](#)

putBoolean method
 of FileChannel [312](#)

putByte method [312](#)

putChar method [312](#)

putDouble, putFloat methods
 of FileChannel [312](#)

putIfAbsent method
 of ConcurrentHashMap [364](#)
 of Map [256](#)

putInt, putLong methods
 of FileChannel [312](#)

putShort method [312](#)

Q

qualified exports [442](#)

Queue class [249, 265](#)
 synchronizing methods in [382](#)
 using ArrayDeque with [266](#)

quote method (Pattern) [326](#)

quoteReplacement method [334](#)

R

race conditions [296, 359](#)

Random class [7, 106](#)
 nextInt method [7](#)
 random numbers [7, 106](#)
 streams of [273, 278, 295](#)

RandomAccess interface [248](#)

RandomAccessFile class [311](#)
 getFilePointer method [311](#)
 length method [311](#)
 seek method [311](#)

RandomGenerator interface [106](#)
 getDefault method [7](#)
 methods of [295](#)

RandomNumbers [85](#)

range [265](#)

range, rangeClosed methods (XxxStream) [294](#)

ranges [267](#)
 converting to streams [296](#)

raw types [228, 232](#)

read method
 of Files [303](#)
 of InputStream [302](#)
 of InputStreamReader [306](#)

readAllXxx methods (Files) [303, 306](#)

readByte, readChar methods (DataInput) [310](#)

readDouble method
 of DataInput [310](#)
 of ObjectInputStream [339](#)

Reader class [306](#)

readers 301
 readExternal method 340
 readFields method 343
 readFloat, readFully methods (DataInput) 310
 readInt method 310, 311
 readLine method
 of BufferedReader 307
 of Console 36
 readLong method 310
 readNBytes method 303
 readObject method
 of HashSet 339
 of ObjectInputStream 337, 345
 readPassword method 36
 readResolve method 341
 readShort method 310
 readUnsignedXxx, readUTF methods (DataOutput) 310
 receiver parameters 70, 405
 records 78
 customizing serialization of 341
 serializable 344
 redirection syntax 36
 redirectXxx methods (ProcessBuilder) 389, 390
 reduce 292
 reduceXxx methods (ConcurrentHashMap) 365
 reducing 292
 reductions 280, 292
 ReentrantLock class 379
 lock, unlock methods 380
 reflection 183
 generic types and 233, 238
 module system and 184, 185, 433, 440
 processing annotations with 413
 security and 345
 ReflectiveOperationException class 175
 regular expressions 325
 flags for 335
 groups in 332
 replacing matches with 334
 splitting input with 333
 testing matches of 330, 332
 turning into predicates 331
 relational operators 23
 relativize 314
 remainderUnsigned method 21
 remove method
 of ArrayDeque 266
 of ArrayList 51
 of BlockingQueue 365
 of Collection 246
 of Iterator 251
 of List 248
 of Map 257
 removeAll method 246
 removeIf method
 of ArrayList 123
 of Collection 246
 of Iterator 251
 @Repeatable annotation 409, 412
 @RepeatedTest annotation 401

replace method
 of Map 257
 of String 29
 replaceAll method
 of Collections 249
 of List 248
 of Map 257
 of Matcher 334, 335
 of String 334
 replaceFirst method 335
 requireNonNull, requireNonNullXxx methods (Objects) 204
 requires keyword 426, 429, 432, 437, 440
 Reserved words. See Keywords
 resolve, resolveSibling methods (Path) 313
 resources 174
 loading 178, 435
 managing 198
 resume 385
 retainAll method 246
 @Retention annotation 406, 409
 return keyword 58, 69, 99
 in finally blocks 201
 in lambda expressions 122
 return types, covariant 144, 231
 return values
 as arrays 59
 missing 280
 providing type of 58
 reverse 54, 250
 reverse domain name convention 86, 425
 reversed 135
 reverseOrder method 136
 rotate 250
 round 22
 RoundEnvironment interface 417
 roundoff errors 14
 RowSetProvider class 440
 runAfterXxx methods (CompletableFuture) 374, 375
 Runnable interface 119, 128, 352, 353
 executing on the UI thread 376
 run method 128, 350, 355, 385
 using class literals with 175
 runtime
 availableProcessors method 351
 exec method 388
 raw types at 232
 safety checks at 229
 runtime image file 446
 RuntimeException class 194

S

s, S conversion characters 38
 safety checks, as runtime 229
 @SafeVarargs annotation 235, 409, 410
 sample code 6
 Scala 227
 Scanner class 35
 findAll method 332

hasNext, hasNextXxx methods 35, 307
next, nextXxx methods 35, 307
tokens method 274, 307
sealed keyword 152
sealed types 150
searchXxx methods (ConcurrentHashMap) 365
security 91, 344
SecurityException class 184
@see annotation 101
seek 311
sequences, producing 274
@Serial annotation 338, 341, 408, 410
serial numbers 337
Serializable interface 336
 readResolve, writeReplace methods 341
serialization 336
 filters for 345
serialVersionUID method 343
server-side software 336
ServiceLoader class 181, 443
 iterator method 182
 load method 182, 444
ServiceLoader.Provider interface 182
services
 configurable 181
 loading 181, 443
ServletException 203
set method 249, 367
 of Array 189
 of ArrayList 51
 of BitSet 263
 of Field 186
 of List 248
 of ListIterator 252
 of method 264
 working with EnumSet 265
setAccessible method 184, 185, 186
setAll method 127
setBoolean, setByte, setChar methods
 of Array 189
 of Field 186
setClassAssertionStatus method 207
setContextClassLoader method 180
setDaemon method 388
setDefaultAssertionStatus method 207
setDefaultUncaughtExceptionHandler method 204
setDoOutput method 321
setDouble, setFloat, setInt, setLong methods
 of Array 189
 of Field 186
setOut method 84
setPackageAssertionStatus method 207
setProperty method 211
setRequestProperty method 321
sets 252
 immutable 362
 threadsafe 367
 unmodifiable views of 268
setShort method
 of Array 189
of Field 186
setUncaughtExceptionHandler method 385
shallow copies 161, 163
shared variables 359, 362
 atomic mutations of 376
 locking 379
shell
 redirection syntax of 36
 scripts for, generating 419
shift operators 24
Shift_JIS encoding 305
short type 12, 51
 MAX_VALUE, MIN_VALUE constants 12
streams of 294
 type conversions of 22
short circuit evaluation 23
short-term persistence 342
shuffle 54, 250
SimpleFileVisitor class 319
@since annotation 100
singletons 341
size method
 of ArrayList 51
 of Collection 246
 of Map 257
skip 278
skipNBytes method 303
sleep 355
SLF4J 207, 425
SOAP protocol 424
SocketHandler class 213
sort method
 of Arrays 54, 118, 119, 123, 124
 of Collections 54, 226, 227, 240, 250
 of List 248
sorted 279
sorted maps 267, 268
sorted sets 249, 267
 traversing 253
 unmodifiable views of 268
sorted streams 296
SortedMap interface 268
SortedSet interface 249, 253
 first method 254
 headSet method 254, 268
 last method 254
 subSet, tailSet methods 254, 268
sorting
 array lists 54
 arrays 54, 117
 chaining comparators for 135
 changing order of 134
 streams 279
 strings 28, 124
source code, generating 409, 410, 417
source files
 documentation comments for 102
 placing, in a file system 87
space flag (for output) 39
spaces

in regular expressions 327
removing 29
split method
 of Pattern 333
 of String 26, 333
splitAsStream method 274, 333
spliterator 247
Spliterators class
 spliteratorUnknownSize method 275
splitWithDelimiters method 333
SQL 34
sqrt 21
square root, computing 284
Stack class 265
stack trace 203, 205
StackWalker class 204
standard output 2
StandardCharsets class 305
start method
 of Matcher, MatchResult 331, 333
 of ProcessBuilder 390
 of Thread 385
startPipeline method 390
startsWith method 29
stateless operations 296
statements, combining 48
static keyword 2, 17, 58, 82, 167
 for modules 442
static constants 83
static imports 92
static initialization 179
static methods 58, 84, 85
 accessing static variables from 85
 importing 92
 in interfaces 113, 114
static nested classes 93
static variables 82
 accessing from static methods 85
 importing 92
 visibility of 359
stop 385
Stream interface
 anyMatch method 281
 collect method 286, 294
 concat method 279
 count method 272, 280
 distinct method 279, 296
 dropWhile method 278
 empty method 273
 filter method 272, 276, 280
 findAny method 280
 findFirst method 182, 280
 flatMap method 277
 forEach, forEachOrdered methods 286
 generate method 273, 294
 iterate method 274, 279, 294, 368
 iterator method 286
 limit method 278, 297
 map method 276
 mapToInt method 293
 max, min methods 280
 noneMatch method 281
 of Arrays 273, 294
 of BitSet 263
 of Collection 247, 272
 of method 273
 of Optional 285
 of StreamSupport 275
 ofNullable method 274, 286
 peek method 279
 reduce method 292
 skip method 278
 sorted method 279
 takeWhile method 278
 toArray method 126, 286
 toList method 275
 unordered method 296
streams 271, 276
 collecting elements of 286, 289
 combining 278
 computing values from 292
 converting to/from arrays 273, 286, 296, 369
 creating 273
 debugging 279
 empty 273, 280, 292, 293
 filtering 285
 finite 274
 flattening 277, 285
 infinite 272, 273, 278, 279
 intermediate operations for 273
 locating services with 182
 noninterference of 276
 of primitive type values 293, 295
 of random numbers 295
 ordered 296
 parallel 272, 280, 286, 289, 290, 293, 295, 368
 processed lazily 272, 276, 279
 reductions of 280
 removing duplicates from 279
 returned by Files.lines 297
 sorting 279
 splitting 278
 summarizing 287, 295
 terminal operation for 273, 280
 transformations of 276, 295
 vs. collections 272
StreamSupport class
 stream method 275
String class 7, 29
 charAt method 33
 compareTo method 28, 117
 compareToIgnoreCase method 124
 contains method 29
 endsWith method 29
 equals method 27
 equalsIgnoreCase method 28
 final 148
 formatted method 39
 graphemeClusters method 279
 hash codes 160

immutability of 30, 362
 indexOf, lastIndexOf methods 29
 join method 26
 length method 7, 33
 replace method 29
 replaceAll method 334
 split method 26, 333
 startsWith method 29
 substring method 26
 toLowerCase method 29, 276
 toUpperCase method 29
StringBuilder class 26
 strings 7, 25
 comparing 27
 concatenating 25, 156
 converting
 from byte arrays 306
 from objects 155
 to numbers 28
 converting to code points 277
 empty 27, 28, 156
 formatting for output 38
 internal representation of 33
 sorting 28, 124
 splitting 26, 274
 transforming to lower/uppercase 276
StringWriter class 309
strip 29
strong 99
subclasses 142
 anonymous 146, 166
 calling `toString` method in 156
 constructors for 144
 inheriting annotations 409
 initializing instance variables in 144
 methods in 142
 preventing 148
 public 144
 superclass assignments in 145
sublist method 248, 267
subMap method 268
subpath 314
subSet method
 of `NavigableSet` 254
 of `SortedSet` 254, 268
substring 26
subtractExact method 21
subtraction 19
 accurate 25
 not associative 293
subtypes 109
 wildcards for 223
sum method
 of `LongAdder` 378
 of `Xxstream` 295
summarizingXxx methods (Collectors) 287, 291
summaryStatistics method 295
summingXxx methods (Collectors) 290
super keyword 115, 143, 144, 147, 224, 227
superclasses 142
 annotating 403
 calling `equals` method on 158
 default methods of 154
 methods of 143
 public 144
 serializability of 337
supertypes 109, 110, 112
 wildcards for 224
Supplier interface 128, 370
supplyAsync method 370, 371
supportsNormalTermination method
 of `Process` 391
 of `ProcessHandle` 393
@SuppressWarnings annotation 232, 409, 410, 411, 427
suspend 385
swap 250
Swing GUI toolkit 120, 376
SwingConstants interface 112
SwingWorker class 376
switch keyword 40
 enhanced 42, 171
 exhaustive 41
 fall-through variant of 42
 using enumerations in 167
 with pattern matching 151
symbolic links 317, 318
synchronized keyword 379, 384
synchronized views 269
synchronizedXxx methods (Collections) 250
System class
 `getLogger` method 208, 210
 `getProperties` method 261
 `getProperty` method 179, 205, 261
 `setOut` method 84
 `setProperty` method 211
 system class loader 178, 180
 system classes, enabling/disabling assertions for 207
 system properties 261, 262
 `System.err` 203, 213, 388
 `System.in` 35
 `System.Logger` interface 208, 211
 `getName` method 210
 `isLoggable` method 210
 `log` method 208, 210
 `System.Logger.Level` enumeration 209
 `System.out` 6, 7, 17, 35, 38, 54, 59, 83, 124, 207, 308

T

t, T conversion characters 38
 tab completion 10
 tagging interfaces 162
 tailMap method 268
 tailSet method
 of `NavigableSet` 254
 of `SortedSet` 254, 268
 take 365
 takeWhile method 278
 tar 87, 88

@Target annotation 406, 409
 tasks 350
 canceling 354
 combining results from 353
 computationally intensive 351
 coordinating work between 365
 defining 119
 executing 351
 executing in a thread 120
 groups of 387
 long-running 375
 running 350
 submitting 353
 vs. threads 351
 working simultaneously 370
 TAU 21
 teeing 292
 terminal window 3, 4
 test method 400, 405, 406
 of BiPredicate 129
 of Predicate 129, 224
 of XxxPredicate 129
 text blocks 33
 thenAccept method 369, 373
 thenAcceptBoth method 374
 thenApply, thenApplyAsync methods
 (CompletableFuture) 372, 373
 thenCombine method 374
 thenComparing method 135
 thenCompose method 373
 thenRun method 374
 third-party libraries 437, 438
 this keyword 71
 annotating 405
 in constructors 75, 363
 in lambda expressions 131
 in method references 125
 Thread class
 getContextClassLoader method 180
 interrupted, isInterrupted methods 356
 isAlive method 385
 join method 385
 properties 387
 resume method (deprecated) 385
 setContextClassLoader method 180
 setDaemon method 388
 setDefaultUncaughtExceptionHandler method 204
 setUncaughtExceptionHandler method 385
 sleep method 355
 start method 351, 385
 stop, suspend methods (deprecated) 385
 ThreadLocal class 386
 ThreadLocalRandom.current method 387
 threads 350, 384
 atomic mutations in 376
 daemon 388
 groups of 387
 interrupting 354, 355
 locking 379
 names of 387
 platform 350
 priorities of 387
 race conditions in 296, 359
 running tasks in 119
 starting 385
 states of 387
 terminating 353
 uncaught exception handlers of 388
 virtual 350
 visibility and 357, 381
 vs. tasks 351
 waiting on conditions 382
 worker 375
 throw keyword 194
 Throwable class 194
 in assertions 206
 initCause method 203
 no generic subtypes for 237
 printStackTrace method 204
 @throws annotation 99, 196, 197
 type variables in 237
 TimeoutException class 353
 Timestamp class 159
 toAbsolutePath method 314
 toArray method
 of Collection 247
 of Stream 126, 286
 of XxxStream 295
 toByteArray method
 of BitSet 263
 of ByteArrayOutputStream 302
 toCollection method 287
 toConcurrentMap method 289
 ToDoubleFunction interface 129, 231
 toFile method 315
 toGenericString method 176
 toHandle method 392
 toIntExact method 23
 ToIntFunction interface 129, 231
 tokens 274, 307
 toList method 275
 toLongArray method 263
 ToLongFunction interface 129, 231
 toLowerCase method 29, 276
 toMap method 287
 toPath method 315
 toSet method 287, 290
 toString method
 calling from subclasses 156
 of Arrays 54, 156
 of BitSet 263
 of Class 176
 of Double 28
 of Enum 165
 of Integer 28
 of Modifier 177
 of Object 155, 157
 of Point 155
 of records 79
 toUnsignedInt method 13

toUpperCase method 29
 transferTo method 304
 transient keyword 338
 transitive keyword 441
 TreeMap class 255, 288
 TreeSet class 252
 Troubleshooting. See Debugging
 true literal 15
 try keyword 197, 202
 for visiting directories 317
 try-with-resources 198
 closing output streams with 303
 for file locking 312
 tryLock method 312
 trySetAccessible method 184
 Type interface 240
 type bounds 221, 240
 annotating 404
 type erasure 228, 236
 clashes after 236
 type method (ServiceLoader.Provider) 182
 type parameters 117, 220
 annotating 402
 primitive types and 220, 231
 type variables
 exceptions and 237
 in static context 236
 no instantiating of 233
 wildcards with 226
 TypeElement interface 417
 TypeVariable interface 240

U

U+ 30
 UnaryOperator interface 128
 uncaught exception handlers 385, 388
 unchecked exceptions 194
 documenting 197
 generic types and 238
 uncheckedIOException class 307
 Unicode 30, 294, 304
 replacement character in 309
 Unit tests 399
 Unix
 executable files in 4
 path separator in 89, 262
 wildcard in classpath in 89
 unlock 380
 unmodifiableXxx methods (Collections) 250
 unordered 296
 updateAndGet method 377
 URI class 323
 URL class 323
 final 148
 getInputStream method 321
 openConnection method 321
 openStream method 302
 URLClassLoader class 179

URLConnection class 321
 connect method 321
 getHeaderFields method 321
 getInputStream method 322
 getOutputStream method 321
 setDoOutput method 321
 setRequestProperty method 321
 URLs, reading from 302, 321
 user.dir, user.home, user.name system properties 261
 uses keyword 444
 UTF-16 14, 30, 294, 305
 in regular expressions 326
 UTF-8 304
 modified 310
 Util.createInstance 180
 utility classes 90, 180

V

validateObject method 344, 345
 valueOf method
 of BitSet 264
 of Enum 164, 165
 values method
 of Enum 164
 of Map 258, 267
 var keyword 15, 16
 varargs parameters
 declaring 59
 safety of 409, 410
 VarHandle class 435
 variable handles 435
 VariableElement interface 416
 variables 7, 15
 atomic mutations of 376
 capturing, in lambda expressions 132
 declaring 15, 17
 defined in interfaces 112
 deprecated 100, 408, 409
 documentation comments for 98, 100
 effectively final 133
 final 359, 362
 holding object references 66
 initializing 15, 17
 instance 68, 71, 74, 77, 79, 81, 85, 144, 149, 158, 338, 341, 343
 local 46
 naming 16
 parameter 72
 private 68, 90
 public static final 112
 redefining 47
 scope of 46, 90
 shared 359, 362, 379
 static 82, 85, 92, 359
 thread-local 386
 using an abstract class as type of 149
 visibility of 357, 381
 volatile 359

@version annotation [99, 102](#)
versioning [342](#)
views [267](#)
virtual machine [3](#)
 instruction reordering in [358](#)
Virtual threads [350](#)
visibility [357](#)
 guaranteed with locks [381](#)
visitFile, visitFileFailed methods (FileVisitor) [319](#)
void keyword [2, 58](#)
 using class literals with [175](#)
volatile keyword [359](#)

W

wait [383, 384](#)
waitFor method [391](#)
waiting on a condition [383](#)
walk [317, 320](#)
walkFileTree method [317, 319](#)
warning [409](#)
warnings, suppressing [232, 235, 410](#)
weak references [267](#)
weaker access privilege [144](#)
WeakHashMap class [267](#)
weakly consistent iterators [367](#)
WeakReference class [267](#)
web pages
 extracting links from [371](#)
 reading [373, 375](#)
whenComplete method [370, 373, 374](#)
while keyword [43](#)
 breaking/continuing [45, 46](#)
 declaring variables for [47](#)
white space
 in regular expressions [327](#)
 in text blocks [34](#)
 removing [29](#)
wildcards
 annotating [404](#)
 capturing [227](#)
 for annotation processors [416](#)
 for types [223, 225](#)
 in class path [89](#)
 unbounded [227](#)
 with imported classes [91](#)
 with type variables [226](#)
WildcardType interface [240](#)
Window class [90](#)

WindowAdapter class [114](#)
WindowListener interface [114](#)
words
 in regular expressions [327](#)
 reading from a file [307](#)
working directory [314, 388](#)
wrapper classes [51](#)
write method
 of Files [308, 316](#)
 of OutputStream [303](#)
 of Writer [307](#)
writeByte, writeChar methods (DataOutput) [310](#)
writeDouble method
 of DataOutput [310](#)
 of ObjectOutputStream [339](#)
writeExternal method [340](#)
writeFloat, writeFully methods (DataOutput) [310](#)
writeInt method [310, 311](#)
writeLong method [310](#)
writeObject method
 of HashSet [339](#)
 of ObjectOutputStream [336, 339](#)
Writer class [307, 309](#)
 write method [307](#)
writeReplace method [341](#)
writers [301](#)
writeShort, writeUnsignedXxx, writeUTF methods
 (DataOutput) [310](#)

X

x, X
 conversion characters [38](#)
XML descriptors, generating [419](#)
xor [263](#)
Xoroshiro128PlusPlus [106](#)

Y

yield keyword [42, 43](#)
yield method (Thread) [351](#)

Z

ZIP file systems [320](#)
ZipInputStream, ZipOutputStream classes [321](#)