This page intentionally left blank
Preface .................................................................................................................. xv
Acknowledgments ................................................................................................. xxi

Chapter 1: Streams .................................................................................................. 1
  1.1 From Iterating to Stream Operations .......................................................... 2
  1.2 Stream Creation .......................................................................................... 5
  1.3 The filter, map, and flatMap Methods ....................................................... 11
  1.4 Extracting Substreams and Combining Streams ........................................ 12
  1.5 Other Stream Transformations .................................................................... 14
  1.6 Simple Reductions ...................................................................................... 15
  1.7 The Optional Type ...................................................................................... 16
    1.7.1 Getting an Optional Value ................................................................. 17
    1.7.2 Consuming an Optional Value .......................................................... 17
    1.7.3 Pipelining Optional Values ............................................................... 18
    1.7.4 How Not to Work with Optional Values .......................................... 19
    1.7.5 Creating Optional Values .................................................................. 20
    1.7.6 Composing Optional Value Functions with flatMap .................. 21
    1.7.7 Turning an Optional into a Stream .................................................... 22
  1.8 Collecting Results ....................................................................................... 25
  1.9 Collecting into Maps ................................................................................... 30
  1.10 Grouping and Partitioning ......................................................................... 34
  1.11 Downstream Collectors ............................................................................. 36
  1.12 Reduction Operations .............................................................................. 41
  1.13 Primitive Type Streams ........................................................................... 43
  1.14 Parallel Streams ....................................................................................... 48

Chapter 2: Input and Output .................................................................................. 55
  2.1 Input/Output Streams ................................................................................ 56
    2.1.1 Reading and Writing Bytes .............................................................. 56
    2.1.2 The Complete Stream Zoo .............................................................. 59
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.3</td>
<td>Combining Input/Output Stream Filters</td>
<td>63</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Text Input and Output</td>
<td>68</td>
</tr>
<tr>
<td>2.1.5</td>
<td>How to Write Text Output</td>
<td>68</td>
</tr>
<tr>
<td>2.1.6</td>
<td>How to Read Text Input</td>
<td>70</td>
</tr>
<tr>
<td>2.1.7</td>
<td>Saving Objects in Text Format</td>
<td>72</td>
</tr>
<tr>
<td>2.1.8</td>
<td>Character Encodings</td>
<td>75</td>
</tr>
<tr>
<td>2.2</td>
<td>Reading and Writing Binary Data</td>
<td>78</td>
</tr>
<tr>
<td>2.2.1</td>
<td>The DataInput and DataOutput interfaces</td>
<td>78</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Random-Access Files</td>
<td>80</td>
</tr>
<tr>
<td>2.2.3</td>
<td>ZIP Archives</td>
<td>85</td>
</tr>
<tr>
<td>2.3</td>
<td>Object Input/Output Streams and Serialization</td>
<td>88</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Saving and Loading Serializable Objects</td>
<td>88</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Understanding the Object Serialization File Format</td>
<td>93</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Modifying the Default Serialization Mechanism</td>
<td>100</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Serializing Singletons and Typesafe Enumerations</td>
<td>102</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Versioning</td>
<td>103</td>
</tr>
<tr>
<td>2.3.6</td>
<td>Using Serialization for Cloning</td>
<td>106</td>
</tr>
<tr>
<td>2.4</td>
<td>Working with Files</td>
<td>109</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Paths</td>
<td>109</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Reading and Writing Files</td>
<td>112</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Creating Files and Directories</td>
<td>113</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Copying, Moving, and Deleting Files</td>
<td>114</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Getting File Information</td>
<td>116</td>
</tr>
<tr>
<td>2.4.6</td>
<td>Visiting Directory Entries</td>
<td>118</td>
</tr>
<tr>
<td>2.4.7</td>
<td>Using Directory Streams</td>
<td>120</td>
</tr>
<tr>
<td>2.4.8</td>
<td>ZIP File Systems</td>
<td>123</td>
</tr>
<tr>
<td>2.5</td>
<td>Memory-Mapped Files</td>
<td>124</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Memory-Mapped File Performance</td>
<td>125</td>
</tr>
<tr>
<td>2.5.2</td>
<td>The Buffer Data Structure</td>
<td>132</td>
</tr>
<tr>
<td>2.6</td>
<td>File Locking</td>
<td>134</td>
</tr>
<tr>
<td>2.7</td>
<td>Regular Expressions</td>
<td>137</td>
</tr>
<tr>
<td>2.7.1</td>
<td>The Regular Expression Syntax</td>
<td>137</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Matching a String</td>
<td>142</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Finding Multiple Matches</td>
<td>145</td>
</tr>
</tbody>
</table>
Chapter 3: XML ................................................................. 153

3.1 Introducing XML ............................................................ 154
3.2 The Structure of an XML Document .............................. 156
3.3 Parsing an XML Document .......................................... 159
3.4 Validating XML Documents ....................................... 169
  3.4.1 Document Type Definitions ................................ 171
  3.4.2 XML Schema ...................................................... 179
  3.4.3 A Practical Example .......................................... 182
3.5 Locating Information with XPath ............................... 188
3.6 Using Namespaces ...................................................... 193
3.7 Streaming Parsers ....................................................... 196
  3.7.1 Using the SAX Parser ...................................... 197
  3.7.2 Using the StAX Parser ...................................... 202
3.8 Generating XML Documents ...................................... 206
  3.8.1 Documents without Namespaces ...................... 206
  3.8.2 Documents with Namespaces .............................. 207
  3.8.3 Writing Documents .......................................... 208
  3.8.4 Writing an XML Document with StAX ............ 210
  3.8.5 An Example: Generating an SVG File ............ 215
3.9 XSL Transformations ............................................... 216

Chapter 4: Networking ..................................................... 227

4.1 Connecting to a Server ............................................... 227
  4.1.1 Using Telnet ...................................................... 227
  4.1.2 Connecting to a Server with Java ...................... 230
  4.1.3 Socket Timeouts ................................................. 232
  4.1.4 Internet Addresses ............................................ 234
4.2 Implementing Servers ............................................... 236
  4.2.1 Server Sockets .................................................. 236
  4.2.2 Serving Multiple Clients .................................... 239
  4.2.3 Half-Close ......................................................... 243
  4.2.4 Interruptible Sockets ........................................ 244
4.3 Getting Web Data ....................................................... 251
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1</td>
<td>URLs and URIs</td>
<td>251</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Using a URLConnection to Retrieve Information</td>
<td>254</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Posting Form Data</td>
<td>261</td>
</tr>
<tr>
<td>4.4</td>
<td>The HTTP Client</td>
<td>271</td>
</tr>
<tr>
<td>4.5</td>
<td>Sending E-Mail</td>
<td>278</td>
</tr>
<tr>
<td>5.1</td>
<td>The Design of JDBC</td>
<td>284</td>
</tr>
<tr>
<td>5.1.1</td>
<td>JDBC Driver Types</td>
<td>285</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Typical Uses of JDBC</td>
<td>286</td>
</tr>
<tr>
<td>5.2</td>
<td>The Structured Query Language</td>
<td>287</td>
</tr>
<tr>
<td>5.3</td>
<td>JDBC Configuration</td>
<td>293</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Database URLs</td>
<td>294</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Driver JAR Files</td>
<td>294</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Starting the Database</td>
<td>294</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Registering the Driver Class</td>
<td>295</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Connecting to the Database</td>
<td>296</td>
</tr>
<tr>
<td>5.4</td>
<td>Working with JDBC Statements</td>
<td>299</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Executing SQL Statements</td>
<td>299</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Managing Connections, Statements, and Result Sets</td>
<td>303</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Analyzing SQL Exceptions</td>
<td>304</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Populating a Database</td>
<td>306</td>
</tr>
<tr>
<td>5.5</td>
<td>Query Execution</td>
<td>310</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Prepared Statements</td>
<td>311</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Reading and Writing LOBs</td>
<td>317</td>
</tr>
<tr>
<td>5.5.3</td>
<td>SQL Escapes</td>
<td>319</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Multiple Results</td>
<td>321</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Retrieving Autogenerated Keys</td>
<td>322</td>
</tr>
<tr>
<td>5.6</td>
<td>Scrollable and Updatable Result Sets</td>
<td>322</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Scrollable Result Sets</td>
<td>323</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Updatable Result Sets</td>
<td>325</td>
</tr>
<tr>
<td>5.7</td>
<td>Row Sets</td>
<td>329</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Constructing Row Sets</td>
<td>330</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Cached Row Sets</td>
<td>330</td>
</tr>
<tr>
<td>5.8</td>
<td>Metadata</td>
<td>334</td>
</tr>
<tr>
<td>5.9</td>
<td>Transactions</td>
<td>344</td>
</tr>
</tbody>
</table>
5.9.1 Programming Transactions with JDBC ............................. 344
5.9.2 Save Points ................................................................. 345
5.9.3 Batch Updates ............................................................. 345
5.9.4 Advanced SQL Types ................................................... 348
5.10 Connection Management in Web and Enterprise Applications .. 349

Chapter 6: The Date and Time API ................................................. 353
6.1 The Time Line ....................................................................... 354
6.2 Local Dates .......................................................................... 358
6.3 Date Adjusters ........................................................................ 364
6.4 Local Time ............................................................................. 365
6.5 Zoned Time ........................................................................... 367
6.6 Formatting and Parsing ....................................................... 371
6.7 Interoperating with Legacy Code ........................................ 376

Chapter 7: Internationalization .................................................... 379
7.1 Locales ................................................................................. 380
7.1.1 Why Locales? ............................................................... 380
7.1.2 Specifying Locales ....................................................... 381
7.1.3 The Default Locale ....................................................... 384
7.1.4 Display Names ............................................................. 384
7.2 Number Formats ................................................................. 387
7.2.1 Formatting Numeric Values ........................................... 387
7.2.2 Currencies ...................................................................... 393
7.3 Date and Time ....................................................................... 394
7.4 Collation and Normalization ............................................... 402
7.5 Message Formatting ........................................................... 409
7.5.1 Formatting Numbers and Dates ..................................... 409
7.5.2 Choice Formats ............................................................. 411
7.6 Text Input and Output .......................................................... 413
7.6.1 Text Files ...................................................................... 414
7.6.2 Line Endings .................................................................... 414
7.6.3 The Console ................................................................. 414
7.6.4 Log Files ........................................................................ 415
7.6.5 The UTF-8 Byte Order Mark ........................................ 415
7.6.6 Character Encoding of Source Files ............................... 416
7.7 Resource Bundles ............................................................ 417
Chapter 8: Scripting, Compiling, and Annotation Processing .......... 439

8.1 Scripting for the Java Platform ............................................. 440
  8.1.1 Getting a Scripting Engine ............................................. 440
  8.1.2 Script Evaluation and Bindings ....................................... 441
  8.1.3 Redirecting Input and Output ......................................... 444
  8.1.4 Calling Scripting Functions and Methods ........................ 444
  8.1.5 Compiling a Script ...................................................... 446
  8.1.6 An Example: Scripting GUI Events ............................... 447

8.2 The Compiler API ........................................................................ 452
  8.2.1 Invoking the Compiler ................................................... 453
  8.2.2 Launching a Compilation Task ....................................... 453
  8.2.3 Capturing Diagnostics ................................................... 454
  8.2.4 Reading Source Files from Memory ............................... 454
  8.2.5 Writing Byte Codes to Memory ..................................... 455
  8.2.6 An Example: Dynamic Java Code Generation ............... 457

8.3 Using Annotations ..................................................................... 463
  8.3.1 An Introduction into Annotations ................................... 464
  8.3.2 An Example: Annotating Event Handlers ..................... 465

8.4 Annotation Syntax ...................................................................... 471
  8.4.1 Annotation Interfaces .................................................... 471
  8.4.2 Annotations .................................................................... 473
  8.4.3 Annotating Declarations .............................................. 475
  8.4.4 Annotating Type Uses .................................................... 476
  8.4.5 Annotating this ............................................................ 477

8.5 Standard Annotations ................................................................. 478
  8.5.1 Annotations for Compilation ......................................... 480
  8.5.2 Annotations for Managing Resources ........................... 480
  8.5.3 Meta-Annotations .......................................................... 481

8.6 Source-Level Annotation Processing ........................................... 484
  8.6.1 Annotation Processors .................................................... 484
  8.6.2 The Language Model API .............................................. 485
8.6.3 Using Annotations to Generate Source Code .......... 486
8.7 Bytecode Engineering .................................................. 489
  8.7.1 Modifying Class Files ........................................... 490
  8.7.2 Modifying Bytecodes at Load Time ....................... 495

Chapter 9: The Java Platform Module System ....................... 499
  9.1 The Module Concept .................................................. 500
  9.2 Naming Modules ........................................................ 501
  9.3 The Modular “Hello, World!” Program ....................... 502
  9.4 Requiring Modules .................................................... 504
  9.5 Exporting Packages .................................................. 506
  9.6 Modular JARs ........................................................... 510
  9.7 Modules and Reflective Access ................................... 511
  9.8 Automatic Modules ................................................... 515
  9.9 TheUnnamed Module ................................................ 517
  9.10 Command-Line Flags for Migration ......................... 518
  9.11 Transitive and Static Requirements ......................... 519
  9.12 Qualified Exporting and Opening ............................ 521
  9.13 Service Loading ...................................................... 522
  9.14 Tools for Working with Modules ......................... 524

Chapter 10: Security ......................................................... 529
  10.1 Class Loaders .......................................................... 530
    10.1.1 The Class-Loading Process ................................. 530
    10.1.2 The Class Loader Hierarchy .............................. 532
    10.1.3 Using Class Loaders as Namespaces .................... 534
    10.1.4 Writing Your Own Class Loader ....................... 534
    10.1.5 Bytecode Verification ....................................... 541
  10.2 Security Managers and Permissions ....................... 546
    10.2.1 Permission Checking ....................................... 546
    10.2.2 Java Platform Security ..................................... 547
    10.2.3 Security Policy Files ....................................... 551
    10.2.4 Custom Permissions ........................................ 559
    10.2.5 Implementation of a Permission Class ............... 560
  10.3 User Authentication ................................................ 566
    10.3.1 The JAAS Framework ....................................... 566
    10.3.2 JAAS Login Modules ........................................ 573
10.4 Digital Signatures ................................................................. 582
  10.4.1 Message Digests ............................................................. 583
  10.4.2 Message Signing .......................................................... 587
  10.4.3 Verifying a Signature .................................................... 589
  10.4.4 The Authentication Problem ...................................... 592
  10.4.5 Certificate Signing ..................................................... 594
  10.4.6 Certificate Requests ................................................... 596
  10.4.7 Code Signing ............................................................. 597

10.5 Encryption .......................................................................... 599
  10.5.1 Symmetric Ciphers ..................................................... 600
  10.5.2 Key Generation .......................................................... 602
  10.5.3 Cipher Streams .......................................................... 607
  10.5.4 Public Key Ciphers ..................................................... 608

Chapter 11: Advanced Swing and Graphics ........................................

11.1 Tables .................................................................................. 613
  11.1.1 A Simple Table ............................................................ 614
  11.1.2 Table Models ............................................................. 618
  11.1.3 Working with Rows and Columns .............................. 622
    11.1.3.1 Column Classes .................................................. 622
    11.1.3.2 Accessing Table Columns .................................. 623
    11.1.3.3 Resizing Columns .............................................. 624
    11.1.3.4 Resizing Rows .................................................. 625
    11.1.3.5 Selecting Rows, Columns, and Cells .................. 626
    11.1.3.6 Sorting Rows .................................................. 627
    11.1.3.7 Filtering Rows ................................................ 628
    11.1.3.8 Hiding and Displaying Columns ......................... 630
  11.1.4 Cell Rendering and Editing .......................................... 639
    11.1.4.1 Rendering Cells .............................................. 639
    11.1.4.2 Rendering the Header ..................................... 641
    11.1.4.3 Editing Cells ................................................ 641
    11.1.4.4 Custom Editors ............................................. 642

11.2 Trees ................................................................................... 652
  11.2.1 Simple Trees ............................................................. 654
    11.2.1.1 Editing Trees and Tree Paths ............................ 663
  11.2.2 Node Enumeration ..................................................... 672
11.2.3 Rendering Nodes ............................................................... 674
11.2.4 Listening to Tree Events .................................................. 677
11.2.5 Custom Tree Models .......................................................... 684
11.3 Advanced AWT ........................................................................ 693
  11.3.1 The Rendering Pipeline .................................................... 694
  11.3.2 Shapes ............................................................................. 696
    11.3.2.1 The Shape Class Hierarchy ....................................... 697
    11.3.2.2 Using the Shape Classes ......................................... 698
  11.3.3 Areas ............................................................................. 714
  11.3.4 Strokes .......................................................................... 715
  11.3.5 Paint ............................................................................... 724
  11.3.6 Coordinate Transformations ......................................... 727
  11.3.7 Clipping .......................................................................... 733
  11.3.8 Transparency and Composition .................................... 735
11.4 Raster Images ........................................................................... 744
  11.4.1 Readers and Writers for Images ....................................... 745
    11.4.1.1 Obtaining Readers and Writers for Image File Types ........................................ 745
    11.4.1.2 Reading and Writing Files with Multiple Images ........................................... 747
  11.4.2 Image Manipulation ....................................................... 756
    11.4.2.1 Constructing Raster Images .................................... 756
    11.4.2.2 Filtering Images ...................................................... 763
11.5 Printing .................................................................................. 772
  11.5.1 Graphics Printing ............................................................ 772
  11.5.2 Multiple-Page Printing .................................................... 782
  11.5.3 Print Services ................................................................. 792
  11.5.4 Stream Print Services ..................................................... 796
  11.5.5 Printing Attributes ......................................................... 799

Chapter 12: Native Methods .............................................................. 809
  12.1 Calling a C Function from a Java Program .......................... 810
  12.2 Numeric Parameters and Return Values .............................. 817
  12.3 String Parameters ............................................................... 819
  12.4 Accessing Fields ................................................................. 825
    12.4.1 Accessing Instance Fields .......................................... 825
12.4.2 Accessing Static Fields ........................................................ 829
12.5 Encoding Signatures ........................................................................ 831
12.6 Calling Java Methods ........................................................................ 832
  12.6.1 Instance Methods .................................................................... 833
  12.6.2 Static Methods ....................................................................... 834
  12.6.3 Constructors .......................................................................... 835
  12.6.4 Alternative Method Invocations ............................................. 835
12.7 Accessing Array Elements ............................................................... 840
12.8 Handling Errors ................................................................................ 844
12.9 Using the Invocation API .................................................................. 849
12.10 A Complete Example: Accessing the Windows Registry ............. 855
  12.10.1 Overview of the Windows Registry ........................................ 855
  12.10.2 A Java Platform Interface for Accessing the Registry .... 856
  12.10.3 Implementation of Registry Access Functions as
          Native Methods ....................................................................... 857

Index ............................................................................................................. 873
Preface

To the Reader

The book you have in your hands is the second volume of the eleventh edition of Core Java, fully updated for Java SE 11. The first volume covers the essential features of the language; this volume deals with the advanced topics that a programmer needs to know for professional software development. Thus, as with the first volume and the previous editions of this book, we are still targeting programmers who want to put Java technology to work in real projects.

As is the case with any book, errors and inaccuracies are inevitable. Should you find any in this book, we would very much like to hear about them. Of course, we would prefer to hear about them only once. For this reason, we have put up a web site at http://horstmann.com/corejava with a FAQ, bug fixes, and workarounds. Strategically placed at the end of the bug report web page (to encourage you to read the previous reports) is a form that you can use to report bugs or problems and to send suggestions for improvements for future editions.

About This Book

The chapters in this book are, for the most part, independent of each other. You should be able to delve into whatever topic interests you the most and read the chapters in any order.

In Chapter 1, you will learn all about the Java stream library that brings a modern flavor to processing data, by specifying what you want without describing in detail how the result should be obtained. This allows the stream library to focus on an optimal evaluation strategy, which is particularly advantageous for optimizing concurrent computations.

The topic of Chapter 2 is input and output handling (I/O). In Java, all input and output is handled through input/output streams. These streams (not to be confused with those in Chapter 1) let you deal, in a uniform manner, with communications among various sources of data, such as files, network connections, or memory blocks. We include detailed coverage of the reader and
writer classes that make it easy to deal with Unicode. We show you what goes on under the hood when you use the object serialization mechanism, which makes saving and loading objects easy and convenient. We then move on to regular expressions and working with files and paths. Throughout this chapter, you will find welcome enhancements in recent Java versions.

**Chapter 3** covers XML. We show you how to parse XML files, how to generate XML, and how to use XSL transformations. As a useful example, we show you how to specify the layout of a Swing form in XML. We also discuss the XPath API, which makes finding needles in XML haystacks much easier.

**Chapter 4** covers the networking API. Java makes it phenomenally easy to do complex network programming. We show you how to make network connections to servers, how to implement your own servers, and how to make HTTP connections. This chapter includes coverage of the new HTTP client.

**Chapter 5** covers database programming. The main focus is on JDBC, the Java database connectivity API that lets Java programs connect to relational databases. We show you how to write useful programs to handle realistic database chores, using a core subset of the JDBC API. (A complete treatment of the JDBC API would require a book almost as big as this one.) We finish the chapter with a brief introduction into hierarchical databases and discuss JNDI (the Java Naming and Directory Interface) and LDAP (the Lightweight Directory Access Protocol).

Java had two prior attempts at libraries for handling date and time. The third one was the charm in Java 8. In **Chapter 6**, you will learn how to deal with the complexities of calendars and time zones, using the new date and time library.

**Chapter 7** discusses a feature that we believe can only grow in importance: internationalization. The Java programming language is one of the few languages designed from the start to handle Unicode, but the internationalization support on the Java platform goes much further. As a result, you can internationalize Java applications so that they cross not only platforms but country boundaries as well. For example, we show you how to write a retirement calculator that uses either English, German, or Chinese languages.

**Chapter 8** discusses three techniques for processing code. The scripting and compiler APIs allow your program to call code in scripting languages such as JavaScript or Groovy, and to compile Java code. Annotations allow you to add arbitrary information (sometimes called metadata) to a Java program. We
show you how annotation processors can harvest these annotations at the source or class file level, and how annotations can be used to influence the behavior of classes at runtime. Annotations are only useful with tools, and we hope that our discussion will help you select useful annotation processing tools for your needs.

In Chapter 9, you will learn about the Java Platform Module System that was introduced in Java 9 to facilitate an orderly evolution of the Java platform and core libraries. This module system provides encapsulation for packages and a mechanism for describing module requirements. You will learn the properties of modules so that you can decide whether to use them in your own applications. Even if you decide not to, you need to know the new rules so that you can interact with the Java platform and other modularized libraries.

Chapter 10 takes up the Java security model. The Java platform was designed from the ground up to be secure, and this chapter takes you under the hood to see how this design is implemented. We show you how to write your own class loaders and security managers for special-purpose applications. Then, we take up the security API that allows for such important features as message and code signing, authorization and authentication, and encryption. We conclude with examples that use the AES and RSA encryption algorithms.

Chapter 11 contains all the Swing material that didn’t make it into Volume I, especially the important but complex tree and table components. We also cover the Java 2D API, which you can use to create realistic drawings and special effects. Of course, not many programmers need to program Swing user interfaces these days, so we pay particular attention to features that are useful for images that can be generated on a server.

Chapter 12 takes up native methods, which let you call methods written for a specific machine such as the Microsoft Windows API. Obviously, this feature is controversial: Use native methods, and the cross-platform nature of Java vanishes. Nonetheless, every serious programmer writing Java applications for specific platforms needs to know these techniques. At times, you need to turn to the operating system’s API for your target platform when you interact with a device or service that is not supported by Java. We illustrate this by showing you how to access the registry API in Windows from a Java program.

As always, all chapters have been completely revised for the latest version of Java. Outdated material has been removed, and the new APIs of Java 9, 10, and 11 are covered in detail.
Conventions

As is common in many computer books, we use monospace type to represent computer code.

NOTE: Notes are tagged with “note” icons that look like this.

TIP: Tips are tagged with “tip” icons that look like this.

CAUTION: When there is danger ahead, we warn you with a “caution” icon.

C++ NOTE: There are a number of C++ notes that explain the difference between the Java programming language and C++. You can skip them if you aren’t interested in C++.

Java comes with a large programming library, or Application Programming Interface (API). When using an API call for the first time, we add a short summary description at the end of the section. These descriptions are a bit more informal but, we hope, also a little more informative than those in the official online API documentation. The names of interfaces are in italics, just like in the official documentation. The number after a class, interface, or method name is the JDK version in which the feature was introduced.

Programs whose source code is included in the companion code for this book are listed as examples, for instance

Listing 1.1 ScriptTest.java

You can download the companion code from http://horstmann.com/corejava.
This page intentionally left blank
Acknowledgments

Writing a book is always a monumental effort, and rewriting doesn’t seem to be much easier, especially with such a rapid rate of change in Java technology. Making a book a reality takes many dedicated people, and it is my great pleasure to acknowledge the contributions of the entire Core Java team.

A large number of individuals at Pearson provided valuable assistance, but they managed to stay behind the scenes. I’d like them all to know how much I appreciate their efforts. As always, my warm thanks go to my editor, Greg Doench, for steering the book through the writing and production process, and for allowing me to be blissfully unaware of the existence of all those folks behind the scenes. I am very grateful to Julie Nahil for production support, and to Dmitry Kirsanov and Alina Kirsanova for copyediting and typesetting the manuscript.

Thanks to the many readers of earlier editions who reported embarrassing errors and made lots of thoughtful suggestions for improvement. I am particularly grateful to the excellent reviewing team that went over the manuscript with an amazing eye for detail and saved me from many more embarrassing errors.

Reviewers of this and earlier editions include Chuck Allison (Contributing Editor, C/C++ Users Journal), Lance Anderson (Oracle), Alec Beaton (PointBase, Inc.), Cliff Berg (iSavvix Corporation), Joshua Bloch, David Brown, Corky Cartwright, Frank Cohen (PushToTest), Chris Crane (devXsolution), Dr. Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), Robert Evans (Senior Staff, The Johns Hopkins University Applied Physics Lab), David Geary (Sabreware), Jim Gish (Oracle), Brian Goetz (Oracle), Angela Gordon, Dan Gordon, Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Steve Haines, Marty Hall (The Johns Hopkins University Applied Physics Lab), Vincent Hardy, Dan Harkey (San Jose State University), William Higgins (IBM), Vladimir Ivanovic (PointBase), Jerry Jackson (ChannelPoint Software), Tim Kimmet (Preview Systems), Chris Laffra, Charlie Lai, Angelika Langer, Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University), Hao Pham, Paul Philion, Blake Ragsdell, Ylber Ramadani (Ryerson University),
Stuart Reges (University of Arizona), Simon Ritter, Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nice, France), Dr. Paul Sanghera (San Jose State University and Brooks College), Paul Sevinc (Teamup AG), Yoshiki Shabata, Devang Shah, Richard Slywczak (NASA/Glenn Research Center), Bradley A. Smith, Steven Stelting, Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (author of Core JFC, Second Edition), Janet Traub, Paul Tyma (consultant), Christian Ullenboom, Peter van der Linden, Burt Walsh, Joe Wang (Oracle), and Dan Xu (Oracle).

Cay Horstmann
San Francisco, California
December 2018
In this chapter, we cover the Java Application Programming Interfaces (APIs) for input and output. You will learn how to access files and directories and how to read and write data in binary and text format. This chapter also shows you the object serialization mechanism that lets you store objects as easily as you can store text or numeric data. Next, we will turn to working with files and directories. We finish the chapter with a discussion of regular expressions, even though they are not actually related to input and output. We couldn’t find a better place to handle that topic, and apparently neither could the Java team—the regular expression API specification was attached to a specification request for “new I/O” features.
2.1 Input/Output Streams

In the Java API, an object from which we can read a sequence of bytes is called an input stream. An object to which we can write a sequence of bytes is called an output stream. These sources and destinations of byte sequences can be—and often are—files, but they can also be network connections and even blocks of memory. The abstract classes InputStream and OutputStream are the basis for a hierarchy of input/output (I/O) classes.

NOTE: These input/output streams are unrelated to the streams that you saw in the preceding chapter. For clarity, we will use the terms input stream, output stream, or input/output stream whenever we discuss streams that are used for input and output.

Byte-oriented input/output streams are inconvenient for processing information stored in Unicode (recall that Unicode uses multiple bytes per character). Therefore, a separate hierarchy provides classes, inheriting from the abstract Reader and Writer classes, for processing Unicode characters. These classes have read and write operations that are based on two-byte char values (that is, UTF-16 code units) rather than byte values.

2.1.1 Reading and Writing Bytes

The InputStream class has an abstract method:

    abstract int read()

This method reads one byte and returns the byte that was read, or -1 if it encounters the end of the input source. The designer of a concrete input stream class overrides this method to provide useful functionality. For example, in the FileInputStream class, this method reads one byte from a file. System.in is a predefined object of a subclass of InputStream that allows you to read information from “standard input,” that is, the console or a redirected file.

The InputStream class also has nonabstract methods to read an array of bytes or to skip a number of bytes. Since Java 9, there is a very useful method to read all bytes of a stream:

    byte[] bytes = in.readAllBytes();

There are also methods to read a given number of bytes—see the API notes. These methods call the abstract read method, so subclasses need to override only one method.
Similarly, the OutputStream class defines the abstract method

```java
abstract void write(int b)
```

which writes one byte to an output location.

If you have an array of bytes, you can write them all at once:

```java
byte[] values = ...;
out.write(values);
```

The `transferTo` method transfers all bytes from an input stream to an output stream:

```java
in.transferTo(out);
```

Both the `read` and `write` methods *block* until the byte is actually read or written. This means that if the input stream cannot immediately be accessed (usually because of a busy network connection), the current thread blocks. This gives other threads the chance to do useful work while the method is waiting for the input stream to become available again.

The `available` method lets you check the number of bytes that are currently available for reading. This means a fragment like the following is unlikely to block:

```java
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
    var data = new byte[bytesAvailable];
    in.read(data);
}
```

When you have finished reading or writing to an input/output stream, close it by calling the `close` method. This call frees up the operating system resources that are in limited supply. If an application opens too many input/output streams without closing them, system resources can become depleted. Closing an output stream also *flushes* the buffer used for the output stream: Any bytes that were temporarily placed in a buffer so that they could be delivered as a larger packet are sent off. In particular, if you do not close a file, the last packet of bytes might never be delivered. You can also manually flush the output with the `flush` method.

Even if an input/output stream class provides concrete methods to work with the raw `read` and `write` functions, application programmers rarely use them. The data that you are interested in probably contain numbers, strings, and objects, not raw bytes.
Instead of working with bytes, you can use one of many input/output classes that build upon the basic InputStream and OutputStream classes.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract int read()</td>
<td>reads a byte of data and returns the byte read; returns -1 at the end of the input stream.</td>
</tr>
<tr>
<td>int read(byte[] b)</td>
<td>reads into an array of bytes and returns the actual number of bytes read, or -1 at the end of the input stream; this method reads at most $b.length$ bytes.</td>
</tr>
<tr>
<td>int read(byte[] b, int off, int len)</td>
<td>reads up to $len$ bytes, if available without blocking (read), or blocking until all values have been read (readNBytes). Values are placed into $b$, starting at $off$. Returns the actual number of bytes read, or -1 at the end of the input stream.</td>
</tr>
<tr>
<td>byte[] readAllBytes()</td>
<td>yields an array of all bytes that can be read from this stream.</td>
</tr>
<tr>
<td>long transferTo(OutputStream out)</td>
<td>transfers all bytes from this input stream to the given output stream, returning the number of bytes transferred. Neither stream is closed.</td>
</tr>
<tr>
<td>long skip(long n)</td>
<td>skips $n$ bytes in the input stream, returns the actual number of bytes skipped (which may be less than $n$ if the end of the input stream was encountered).</td>
</tr>
<tr>
<td>int available()</td>
<td>returns the number of bytes available, without blocking (recall that blocking means that the current thread loses its turn).</td>
</tr>
<tr>
<td>void close()</td>
<td>closes the input stream.</td>
</tr>
<tr>
<td>void mark(int readlimit)</td>
<td>puts a marker at the current position in the input stream (not all streams support this feature). If more than $readlimit$ bytes have been read from the input stream, the stream is allowed to forget the marker.</td>
</tr>
<tr>
<td>void reset()</td>
<td>returns to the last marker. Subsequent calls to read reread the bytes. If there is no current marker, the input stream is not reset.</td>
</tr>
<tr>
<td>boolean markSupported()</td>
<td>returns true if the input stream supports marking.</td>
</tr>
</tbody>
</table>
2.1.2 The Complete Stream Zoo

Unlike C, which gets by just fine with a single type FILE*, Java has a whole zoo of more than 60 (!) different input/output stream types (see Figures 2.1 and 2.2).

Let’s divide the animals in the input/output stream zoo by how they are used. There are separate hierarchies for classes that process bytes and characters. As you saw, the InputStream and OutputStream classes let you read and write individual bytes and arrays of bytes. These classes form the basis of the hierarchy shown in Figure 2.1. To read and write strings and numbers, you need more capable subclasses. For example, DataInputStream and DataOutputStream let you read and write all the primitive Java types in binary format. Finally, there are input/output streams that do useful stuff; for example, the ZipInputStream and ZipOutputStream let you read and write files in the familiar ZIP compression format.

For Unicode text, on the other hand, you can use subclasses of the abstract classes Reader and Writer (see Figure 2.2). The basic methods of the Reader and Writer classes are similar to those of InputStream and OutputStream.

```java
abstract int read()
abstract void write(int c)
```

The read method returns either a UTF-16 code unit (as an integer between 0 and 65535) or -1 when you have reached the end of the file. The write method is called with a Unicode code unit. (See Volume I, Chapter 3 for a discussion of Unicode code units.)

There are four additional interfaces: Closeable, Flushable, Readable, and Appendable (see Figure 2.3). The first two interfaces are very simple, with methods

```java
void close() throws IOException
```
Figure 2.1 Input and output stream hierarchy

and

```java
void flush()
```

respectively. The classes `InputStream`, `OutputStream`, `Reader`, and `Writer` all implement the `Closeable` interface.
NOTE: The java.io.Closeable interface extends the java.lang.AutoCloseable interface. Therefore, you can use the try-with-resources statement with any Closeable.

Why have two interfaces? The close method of the Closeable interface only throws an IOException, whereas the AutoCloseable.close method may throw any exception.

OutputStream and Writer implement the Flushable interface.
The Readable interface has a single method

    int read(CharBuffer cb)
The CharBuffer class has methods for sequential and random read/write access. It represents an in-memory buffer or a memory-mapped file. (See Section 2.5.2, “The Buffer Data Structure,” on p. 132 for details.)

The Appendable interface has two methods for appending single characters and character sequences:

- `Appendable append(char c)`
- `Appendable append(CharSequence s)`

The CharSequence interface describes basic properties of a sequence of char values. It is implemented by String, CharBuffer, StringBuilder, and StringBuffer.

Of the input/output stream classes, only Writer implements Appendable.

```java
java.io.Closeable 5.0

- void close()
  
  closes this Closeable. This method may throw an IOException.
```
java.io.Flushable 5.0

- void flush()
  flushes this Flushable.

java.lang.Readable 5.0

- int read(CharBuffer cb)
  attempts to read as many char values into cb as it can hold. Returns the number of values read, or -1 if no further values are available from this Readable.

java.lang.Appendable 5.0

- Appendable append(char c)
- Appendable append(CharSequence cs)
  appends the given code unit, or all code units in the given sequence, to this Appendable; returns this.

java.lang.CharSequence 1.4

- char charAt(int index)
  returns the code unit at the given index.
- int length()
  returns the number of code units in this sequence.
- CharSequence subSequence(int startIndex, int endIndex)
  returns a CharSequence consisting of the code units stored from index startIndex to endIndex - 1.
- String toString()
  returns a string consisting of the code units of this sequence.

2.1.3 Combining Input/Output Stream Filters

FileInputStream and FileOutputStream give you input and output streams attached to a disk file. You need to pass the file name or full path name of the file to the constructor. For example,
var fin = new FileInputStream("employee.dat");

looks in the user directory for a file named employee.dat.

**TIP:** All the classes in java.io interpret relative path names as starting from the user’s working directory. You can get this directory by a call to System.getProperty("user.dir").

**CAUTION:** Since the backslash character is the escape character in Java strings, be sure to use \ for Windows-style path names (for example, C:\Windows\win.ini). In Windows, you can also use a single forward slash (C:/Windows/win.ini) because most Windows file-handling system calls will interpret forward slashes as file separators. However, this is not recommended—the behavior of the Windows system functions is subject to change. Instead, for portable programs, use the file separator character for the platform on which your program runs. It is available as the constant string java.io.File.separator.

Like the abstract InputStream and OutputStream classes, these classes only support reading and writing at the byte level. That is, we can only read bytes and byte arrays from the object fin.

    byte b = (byte) fin.read();

As you will see in the next section, if we just had a DataInputStream, we could read numeric types:

    DataInputStream din = ...;
    double x = din.readDouble();

But just as the FileInputStream has no methods to read numeric types, the DataInputStream has no method to get data from a file.

Java uses a clever mechanism to separate two kinds of responsibilities. Some input streams (such as the FileInputStream and the input stream returned by the openStream method of the URL class) can retrieve bytes from files and other more exotic locations. Other input streams (such as the DataInputStream) can assemble bytes into more useful data types. The Java programmer has to combine the two. For example, to be able to read numbers from a file, first create a FileInputStream and then pass it to the constructor of a DataInputStream.

    var fin = new FileInputStream("employee.dat");
    var din = new DataInputStream(fin);
    double x = din.readDouble();
If you look at Figure 2.1 again, you can see the classes \texttt{FilterInputStream} and \texttt{FilterOutputStream}. The subclasses of these classes are used to add capabilities to input/output streams that process bytes.

You can add multiple capabilities by nesting the filters. For example, by default, input streams are not buffered. That is, every call to \texttt{read} asks the operating system to dole out yet another byte. It is more efficient to request blocks of data instead and store them in a buffer. If you want buffering \textit{and} the data input methods for a file, use the following rather monstrous sequence of constructors:

\begin{verbatim}
var din = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
\end{verbatim}

Notice that we put the \texttt{DataInputStream} \textit{last} in the chain of constructors because we want to use the \texttt{DataInputStream} methods, and we want them to use the buffered read method.

Sometimes you’ll need to keep track of the intermediate input streams when chaining them together. For example, when reading input, you often need to peek at the next byte to see if it is the value that you expect. Java provides the \texttt{PushbackInputStream} for this purpose.

\begin{verbatim}
var pbin = new PushbackInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
\end{verbatim}

Now you can speculatively read the next byte

\begin{verbatim}
int b = pbin.read();
\end{verbatim}

and throw it back if it isn’t what you wanted.

\begin{verbatim}
if (b != '<') pbin.unread(b);
\end{verbatim}

However, reading and unreading are the \textit{only} methods that apply to a pushback input stream. If you want to look ahead and also read numbers, then you need both a pushback input stream and a data input stream reference.

\begin{verbatim}
var din = new DataInputStream(
    pbin = new PushbackInputStream(
        new BufferedInputStream(
            new FileInputStream("employee.dat"))));
\end{verbatim}

Of course, in the input/output libraries of other programming languages, niceties such as buffering and lookahead are automatically taken care of—so it is a bit of a hassle to resort, in Java, to combining stream filters. However, the ability to mix and match filter classes to construct useful sequences of
input/output streams does give you an immense amount of flexibility. For example, you can read numbers from a compressed ZIP file by using the following sequence of input streams (see Figure 2.4):

```java
var zin = new ZipInputStream(new FileInputStream("employee.zip"));
var din = new DataInputStream(zin);
```

(See Section 2.2.3, “ZIP Archives,” on p. 85 for more on Java’s handling of ZIP files.)

**Figure 2.4** A sequence of filtered input streams

<table>
<thead>
<tr>
<th>java.io.FileInputStream 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>• FileInputStream(String name)</td>
</tr>
<tr>
<td>• FileInputStream(File file)</td>
</tr>
</tbody>
</table>

creates a new file input stream using the file whose path name is specified by the name string or the file object. (The File class is described at the end of this chapter.) Path names that are not absolute are resolved relative to the working directory that was set when the VM started.
**java.io.FileOutputStream 1.0**

- FileOutputStream(String name)
- FileOutputStream(String name, boolean append)
- FileOutputStream(File file)
- FileOutputStream(File file, boolean append)

creates a new file output stream specified by the name string or the file object. (The File class is described at the end of this chapter.) If the append parameter is true, an existing file with the same name will not be deleted and data will be added at the end of the file. Otherwise, this method deletes any existing file with the same name.

**java.io.BufferedInputStream 1.0**

- BufferedInputStream(InputStream in)

creates a buffered input stream. A buffered input stream reads bytes from a stream without causing a device access every time. When the buffer is empty, a new block of data is read into the buffer.

**java.io.BufferedOutputStream 1.0**

- BufferedOutputStream(OutputStream out)

creates a buffered output stream. A buffered output stream collects bytes to be written without causing a device access every time. When the buffer fills up or when the stream is flushed, the data are written.

**java.io.PushbackInputStream 1.0**

- PushbackInputStream(InputStream in)
- PushbackInputStream(InputStream in, int size)

constructs an input stream with one-byte lookahead or a pushback buffer of specified size.

- void unread(int b)

pushes back a byte, which is retrieved again by the next call to read.
2.1.4 Text Input and Output

When saving data, you have the choice between binary and text formats. For example, if the integer 1234 is saved in binary, it is written as the sequence of bytes 00 00 04 D2 (in hexadecimal notation). In text format, it is saved as the string “1234”. Although binary I/O is fast and efficient, it is not easily readable by humans. We first discuss text I/O and cover binary I/O in Section 2.2, “Reading and Writing Binary Data,” on p. 78.

When saving text strings, you need to consider the character encoding. In the UTF-16 encoding that Java uses internally, the string “José” is encoded as 00 4A 00 6F 00 73 00 E9 (in hex). However, many programs expect that text files use a different encoding. In UTF-8, the encoding most commonly used on the Internet, the string would be written as 4A 6F 73 C3 A9, without the zero bytes for the first three letters and with two bytes for the é character.

The OutputStreamWriter class turns an output stream of Unicode code units into a stream of bytes, using a chosen character encoding. Conversely, the InputStreamReader class turns an input stream that contains bytes (specifying characters in some character encoding) into a reader that emits Unicode code units.

For example, here is how you make an input reader that reads keystrokes from the console and converts them to Unicode:

```java
var in = new InputStreamReader(System.in);
```

This input stream reader assumes the default character encoding used by the host system. On desktop operating systems, that can be an archaic encoding such as Windows 1252 or MacRoman. You should always choose a specific encoding in the constructor for the InputStreamReader, for example:

```java
var in = new InputStreamReader(new FileInputStream("data.txt"), StandardCharsets.UTF_8);
```

See Section 2.1.8, “Character Encodings,” on p. 75 for more information on character encodings.

The Reader and Writer classes have only basic methods to read and write individual characters. As with streams, you use subclasses for processing strings and numbers.

2.1.5 How to Write Text Output

For text output, use a PrintWriter. That class has methods to print strings and numbers in text format. In order to print to a file, construct a PrintStream from a file name and a character encoding:

```java
var out = new PrintWriter("employee.txt", StandardCharsets.UTF_8);
```
To write to a print writer, use the same `print`, `println`, and `printf` methods that you used with `System.out`. You can use these methods to print numbers (int, short, long, float, double), characters, boolean values, strings, and objects.

For example, consider this code:

```java
String name = "Harry Hacker";
double salary = 75000;
out.print(name);
out.print(' ');
out.println(salary);
```

This writes the characters

```
Harry Hacker 75000.0
```

to the writer `out`. The characters are then converted to bytes and end up in the file `employee.txt`.

The `println` method adds the correct end-of-line character for the target system ("\r\n" on Windows, "\n" on UNIX) to the line. This is the string obtained by the call `System.getProperty("line.separator")`.

If the writer is set to **autoflush mode**, all characters in the buffer are sent to their destination whenever `println` is called. (Print writers are always buffered.)

By default, autoflushing is **not** enabled. You can enable or disable autoflushing by using the `PrintWriter(Writer writer, boolean autoFlush)` constructor:

```java
var out = new PrintWriter(
    new OutputStreamWriter(
        new FileOutputStream("employee.txt"), StandardCharsets.UTF_8),
    true); // autoflush
```

The `print` methods don’t throw exceptions. You can call the `checkError` method to see if something went wrong with the output stream.

**NOTE:** Java veterans might wonder whatever happened to the `PrintStream` class and to `System.out`. In Java 1.0, the `PrintStream` class simply truncated all Unicode characters to ASCII characters by dropping the top byte. (At the time, Unicode was still a 16-bit encoding.) Clearly, that was not a clean or portable approach, and it was fixed with the introduction of readers and writers in Java 1.1. For compatibility with existing code, `System.in`, `System.out`, and `System.err` are still input/output streams, not readers and writers. But now the `PrintStream` class internally converts Unicode characters to the default host encoding in the same way the `PrintWriter` does. Objects of type `PrintStream` act exactly like print writers when you use the `print` and `println` methods, but unlike print writers they allow you to output raw bytes with the `write(int)` and `write(byte[])` methods.
### java.io.PrintWriter 1.1

- PrintWriter(Writer out)
- PrintWriter(Writer writer)
  creates a new PrintWriter that writes to the given writer.
- PrintWriter(String filename, String encoding)
- PrintWriter(File file, String encoding)
  creates a new PrintWriter that writes to the given file, using the given character encoding.
- void print(Object obj)
  prints an object by printing the string resulting from toString.
- void print(String s)
  prints a string containing Unicode code units.
- void println(String s)
  prints a string followed by a line terminator. Flushes the output stream if it is in autoflush mode.
- void print(char[] s)
  prints all Unicode code units in the given array.
- void print(char c)
  prints a Unicode code unit.
- void print(int i)
- void print(long l)
- void print(float f)
- void print(double d)
- void print(boolean b)
  prints the given value in text format.
- void printf(String format, Object... args)
  prints the given values as specified by the format string. See Volume I, Chapter 3 for the specification of the format string.
- boolean checkError()
  returns true if a formatting or output error occurred. Once the output stream has encountered an error, it is tainted and all calls to checkError return true.

### 2.1.6 How to Read Text Input

The easiest way to process arbitrary text is the Scanner class that we used extensively in Volume I. You can construct a Scanner from any input stream.
Alternatively, you can read a short text file into a string like this:

```java
var content = new String(Files.readAllBytes(path), charset);
```

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

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

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

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

You can also use a scanner to read tokens—strings that are separated by a delimiter. The default delimiter is white space. You can change the delimiter to any regular expression. For example,

```java
Scanner in = . . .;
in.useDelimiter("\\PL+");
```

accepts any non-Unicode letters as delimiters. The scanner then accepts tokens consisting only of Unicode letters.

Calling the `next` method yields the next token:

```java
while (in.hasNext())
{
    String word = in.next();
    . . .
}
```

Alternatively, you can obtain a stream of all tokens as

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

In early versions of Java, the only game in town for processing text input was the `BufferedReader` class. Its `readLine` method yields a line of text, or `null` when no more input is available. A typical input loop looks like this:

```java
InputStream inputStream = . . .;
try (var in = new BufferedReader(new InputStreamReader(inputStream, charset)))
{
    String line;
    while ((line = in.readLine()) != null)
    {
        do something with line
    }
}
```
Nowadays, the BufferedReader class also has a lines method that yields a Stream<String>. However, unlike a Scanner, a BufferedReader has no methods for reading numbers.

### 2.1.7 Saving Objects in Text Format

In this section, we walk you through an example program that stores an array of Employee records in a text file. Each record is stored in a separate line. Instance fields are separated from each other by delimiters. We use a vertical bar (|) as our delimiter. (A colon (:) is another popular choice. Part of the fun is that everyone uses a different delimiter.) Naturally, we punt on the issue of what might happen if a | actually occurs in one of the strings we save.

Here is a sample set of records:

- Harry Hacker|35500|1989-10-01
- Carl Cracker|75000|1987-12-15
- Tony Tester|38000|1990-03-15

Writing records is simple. Since we write to a text file, we use the PrintWriter class. We simply write all fields, followed by either a | or, for the last field, a newline character. This work is done in the following writeData method that we add to our Employee class:

```java
public static void writeEmployee(PrintWriter out, Employee e) {
    out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
}
```

To read records, we read in a line at a time and separate the fields. We use a scanner to read each line and then split the line into tokens with the String.split method.

```java
public static Employee readEmployee(Scanner in) {
    String line = in.nextLine();
    String[] tokens = line.split("\|");
    String name = tokens[0];
    double salary = Double.parseDouble(tokens[1]);
    LocalDate hireDate = LocalDate.parse(tokens[2]);
    int year = hireDate.getYear();
    int month = hireDate.getMonthValue();
    int day = hireDate.getDayOfMonth();
    return new Employee(name, salary, year, month, day);
}
```

The parameter of the split method is a regular expression describing the separator. We discuss regular expressions in more detail at the end of this chapter. As it happens, the vertical bar character has a special meaning in
regular expressions, so it needs to be escaped with a \ character. That character needs to be escaped by another \, yielding the "\\" expression.

The complete program is in Listing 2.1. The static method

```java
void writeData(Employee[] e, PrintWriter out)
```

first writes the length of the array, then writes each record. The static method

```java
Employee[] readData(BufferedReader in)
```

first reads in the length of the array, then reads in each record. This turns out to be a bit tricky:

```java
int n = in.nextInt();
in.nextLine(); // consume newline
var employees = new Employee[n];
for (int i = 0; i < n; i++)
{
    employees[i] = new Employee();
    employees[i].readData(in);
}
```

The call to `nextInt` reads the array length but not the trailing newline character. We must consume the newline so that the `readData` method can get the next input line when it calls the `nextLine` method.

---

**Listing 2.1 textFile/TextFileTest.java**

```java
package textFile;

import java.io.*;
import java.nio.charset.*;
import java.time.*;
import java.util.*;

/**
 * @version 1.15 2018-03-17
 * @author Cay Horstmann
 */
public class TextFileTest
{
    public static void main(String[] args) throws IOException
    {
        var staff = new Employee[3];
        var staff = new Employee[3];
        staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
        staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
        staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```
Listing 2.1  (Continued)

// save all employee records to the file employee.dat
try (var out = new PrintWriter("employee.dat", StandardCharsets.UTF_8)) {
    writeData(staff, out);
}

// retrieve all records into a new array
try (var in = new Scanner(
    new FileInputStream("employee.dat"), "UTF-8")) {
    Employee[] newStaff = readData(in);
    // print the newly read employee records
    for (Employee e : newStaff)
        System.out.println(e);
}

/**
 * Writes all employees in an array to a print writer
 * @param employees an array of employees
 * @param out a print writer
 */
private static void writeData(Employee[] employees, PrintWriter out)
    throws IOException {
    // write number of employees
    out.println(employees.length);
    for (Employee e : employees)
        writeEmployee(out, e);
}

/**
 * Reads an array of employees from a scanner
 * @param in the scanner
 * @return the array of employees
 */
private static Employee[] readData(Scanner in) {
    // retrieve the array size
    int n = in.nextInt();
    in.nextLine(); // consume newline
    var employees = new Employee[n];
for (int i = 0; i < n; i++)
{
    employees[i] = readEmployee(in);
}
return employees;
}

/**
 * Writes employee data to a print writer
 * @param out the print writer
 */
public static void writeEmployee(PrintWriter out, Employee e)
{
    out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
}

/**
 * Reads employee data from a buffered reader
 * @param in the scanner
 */
public static Employee readEmployee(Scanner in)
{
    String line = in.nextLine();
    String[] tokens = line.split(\"\|\");
    String name = tokens[0];
    double salary = Double.parseDouble(tokens[1]);
    LocalDate hireDate = LocalDate.parse(tokens[2]);
    int year = hireDate.getYear();
    int month = hireDate.getMonthValue();
    int day = hireDate.getDayOfMonth();
    return new Employee(name, salary, year, month, day);
}

2.1.8 Character Encodings

Input and output streams are for sequences of bytes, but in many cases you will work with texts—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 2.1). 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.
Another common encoding is UTF-16, which encodes each Unicode code point into one or two 16-bit values (see Table 2.2). 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 the big-endian format, the more significant byte comes first: 0x21 followed by 0x22. In the 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 then discard it.

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.
There is no reliable way to automatically detect the character encoding from a stream of bytes. Some API methods let you use the “default charset”—the character encoding preferred by the operating system of the computer. Is that the same encoding that is used by your source of bytes? These bytes may well originate from a different part of the world. Therefore, you should always explicitly specify the encoding. For example, when reading a web page, check the Content-Type header.

**NOTE:** The platform 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.

**CAUTION:** The Oracle implementation of Java has a system property `file.encoding` for overriding the platform default. This is not an officially supported property, and it is not consistently followed by all parts of Oracle’s implementation of the Java library. You should not set it.

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");
```

Use the `Charset` object when reading or writing text. For example, you can turn an array of bytes into a string as

```
var str = new String(bytes, StandardCharsets.UTF_8);
```

**TIP:** As of Java 10, all methods in the `java.io` package allow you to specify a character encoding with a `Charset` object or a string. Choose the `StandardCharsets` constants, so that any spelling errors are caught at compile time.

**CAUTION:** Some methods (such as the `String(byte[])` constructor) use the default platform encoding if you don’t specify any; others (such as `Files.readAllLines`) use UTF-8.
2.2 Reading and Writing Binary Data

Text format is convenient for testing and debugging because it is humanly readable, but it is not as efficient as transmitting data in binary format. In the following sections, you will learn how to perform input and output with binary data.

2.2.1 The DataInput and DataOutput interfaces

The DataOutput interface defines the following methods for writing a number, a character, a boolean value, or a string in binary format:

- `writeChars`          `writeFloat`
- `writeByte`           `writeDouble`
- `writeInt`            `writeChar`
- `writeShort`          `writeBoolean`
- `writeLong`           `writeUTF`

For example, `writeInt` always writes an integer as a 4-byte binary quantity regardless of the number of digits, and `writeDouble` always writes a double as an 8-byte binary quantity. The resulting output is not human-readable, but it will use the same space for each value of a given type and reading it back in will be faster than parsing text.

NOTE: There are two different methods of storing integers and floating-point numbers in memory, depending on the processor you are using. Suppose, for example, you are working with a 4-byte int, such as the decimal number 1234, or 4D2 in hexadecimal (1234 = 4 × 256 + 13 × 16 + 2). This value can be stored in such a way that the first of the four bytes in memory holds the most significant byte (MSB) of the value: 00 00 04 D2. This is the so-called big-endian method. Or, we can start with the least significant byte (LSB) first: D2 04 00 00. This is called, naturally enough, the little-endian method. For example, the SPARC uses big-endian; the Pentium, little-endian. This can lead to problems. When a file is saved from C or C++ file, the data are saved exactly as the processor stores them. That makes it challenging to move even the simplest data files from one platform to another. In Java, all values are written in the big-endian fashion, regardless of the processor. That makes Java data files platform-independent.

The `writeUTF` method writes string data using a modified version of the 8-bit Unicode Transformation Format. Instead of simply using the standard UTF-8 encoding, sequences of Unicode code units are first represented in UTF-16, and then the result is encoded using the UTF-8 rules. This modified encoding
is different for characters with codes higher than \(0xFFFF\). It is used for backward compatibility with virtual machines that were built when Unicode had not yet grown beyond 16 bits.

Since nobody else uses this modification of UTF-8, you should only use the `writeUTF` method to write strings intended for a Java virtual machine—for example, in a program that generates bytecodes. Use the `writeChars` method for other purposes.

To read the data back in, use the following methods defined in the `DataInput` interface:

- `readInt`
- `readDouble`
- `readShort`
- `readChar`
- `readLong`
- `readBoolean`
- `readFloat`
- `readUTF`

The `DataInputStream` class implements the `DataInput` interface. To read binary data from a file, combine a `DataInputStream` with a source of bytes such as a `FileInputStream`:

```java
var in = new DataInputStream(new FileInputStream("employee.dat"));
```

Similarly, to write binary data, use the `DataOutputStream` class that implements the `DataOutput` interface:

```java
var out = new DataOutputStream(new FileOutputStream("employee.dat"));
```

---

### java.io.DataInput 1.0

- `boolean readBoolean()`
- `byte readByte()`
- `char readChar()`
- `double readDouble()`
- `float readFloat()`
- `int readInt()`
- `long readLong()`
- `short readShort()`

reads in a value of the given type.

- `void readFully(byte[] b)`

reads bytes into the array `b`, blocking until all bytes are read.

- `void readFully(byte[] b, int off, int len)`

places up to `len` bytes into the array `b`, starting at `off`, blocking until all bytes are read.

(Continues)
2.2.2 Random-Access Files

The RandomAccessFile class lets you read or write data anywhere in a file. Disk files are random-access, but input/output streams that communicate with a network socket are not. 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.

```java
var in = new RandomAccessFile("employee.dat", "r");
var inOut = new RandomAccessFile("employee.dat", "rw");
```

When you open an existing file as a RandomAccessFile, it does not get deleted.

A random-access file has a file pointer that indicates the position of the next byte to be read or written. The seek method can be used to set 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 in bytes.
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 from a random-access file, use methods such as `readInt/writeInt` and `readChar/writeChar` that we discussed in the preceding section.

Let’s walk through an example program that stores employee records in a random-access file. Each record will have the same size. This makes it easy to read an arbitrary record. Suppose you want to position the file pointer to the third record. Simply set the file pointer to the appropriate byte position and start reading.

```java
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
var e = new Employee();
e.readData(in);
```

If you want to modify the record and save it back into the same location, remember to set the file pointer back to the beginning of the record:

```java
in.seek((n - 1) * RECORD_SIZE);
e.writeData(out);
```

To determine the total number of bytes in a file, use the `length` method. The total number of records is the length divided by the size of each record.

```java
long nbytes = in.length(); // length in bytes
int nrecords = (int) (nbytes / RECORD_SIZE);
```

Integers and floating-point values have a fixed size in binary format, but we have to work harder for strings. We provide two helper methods to write and read strings of a fixed size.

The `writeFixedString` writes the specified number of code units, starting at the beginning of the string. If there are too few code units, the method pads the string, using zero values.

```java
public static void writeFixedString(String s, int size, DataOutput out)
throws IOException
{
    for (int i = 0; i < size; i++)
    {
        char ch = 0;
        if (i < s.length()) ch = s.charAt(i);
        out.writeChar(ch);
    }
}
```

The `readFixedString` method reads characters from the input stream until it has consumed `size` code units or until it encounters a character with a zero value.
Then, it skips past the remaining zero values in the input field. For added efficiency, this method uses the `StringBuilder` class to read in a string.

```java
public static String readFixedString(int size, DataInput in) throws IOException {
    var b = new StringBuilder(size);
    int i = 0;
    var done = false;
    while (!done && i < size) {
        char ch = in.readChar();
        i++;
        if (ch == 0) done = true;
        else b.append(ch);
    }
    in.skipBytes(2 * (size - i));
    return b.toString();
}
```

We placed the `writeFixedString` and `readFixedString` methods inside the `DataIO` helper class.

To write a fixed-size record, we simply write all fields in binary.

```java
DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
out.writeDouble(e.getSalary());
LocalDate hireDay = e.getHireDay();
out.writeInt(hireDay.getYear());
out.writeInt(hireDay.getMonthValue());
out.writeInt(hireDay.getDayOfMonth());
```

Reading the data back is just as simple.

```java
String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
double salary = in.readDouble();
int y = in.readInt();
int m = in.readInt();
int d = in.readInt();
```

Let us compute the size of each record. We will use 40 characters for the name strings. Therefore, each record will contain 100 bytes:

- 40 characters = 80 bytes for the name
- 1 `double` = 8 bytes for the salary
- 3 `int` = 12 bytes for the date

The program shown in Listing 2.2 writes three records into a data file and then reads them from the file in reverse order. To do this efficiently requires random access—we need to get to the last record first.
package randomAccess;

import java.io.*;
import java.time.*;

/**
 * @version 1.14 2018-05-01
 * @author Cay Horstmann
 */
public class RandomAccessTest {
    public static void main(String[] args) throws IOException {
        var staff = new Employee[3];

        staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
        staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
        staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

        try (var out = new DataOutputStream(new FileOutputStream("employee.dat"))) {
            // save all employee records to the file employee.dat
            for (Employee e : staff)
                writeData(out, e);
        }

        try (var in = new RandomAccessFile("employee.dat", "r")) {
            // retrieve all records into a new array
            // compute the array size
            int n = (int)(in.length() / Employee.RECORD_SIZE);
            var newStaff = new Employee[n];

            // read employees in reverse order
            for (int i = n - 1; i >= 0; i--) {
                newStaff[i] = new Employee();
                in.seek(i * Employee.RECORD_SIZE);
                newStaff[i] = readData(in);
            }

            // print the newly read employee records
            for (Employee e : newStaff)
                System.out.println(e);
        }
    }
}

(Continues)
Listing 2.2  (Continued)

```java
/**
 * Writes employee data to a data output
 * @param out the data output
 * @param e the employee
 */
public static void writeData(DataOutput out, Employee e) throws IOException {
    DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
    out.writeDouble(e.getSalary());

    LocalDate hireDay = e.getHireDay();
    out.writeInt(hireDay.getYear());
    out.writeInt(hireDay.getMonthValue());
    out.writeInt(hireDay.getDayOfMonth());
}

/**
 * Reads employee data from a data input
 * @param in the data input
 * @return the employee
 */
public static Employee readData(DataInput in) throws IOException {
    String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
    double salary = in.readDouble();
    int y = in.readInt();
    int m = in.readInt();
    int d = in.readInt();
    return new Employee(name, salary, y, m - 1, d);
}
```

**java.io.RandomAccessFile 1.0**

- RandomAccessFile(String file, String mode)
- RandomAccessFile(File file, String mode)

  opens the given file for random access. The mode string is "r" for read-only mode, "rw" for read/write mode, "rws" for read/write mode with synchronous disk writes of data and metadata for every update, and "rwd" for read/write mode with synchronous disk writes of data only.

- long getFilePointer()

  returns the current location of the file pointer.

(Continues)
2.2.3 ZIP Archives

ZIP archives store one or more files in a (usually) compressed format. Each ZIP archive has a header with information such as the name of each file and the compression method that was used. In Java, you can use a ZipInputStream to read a ZIP archive. You need to look at the individual entries in the archive. The getNextEntry method returns an object of type ZipEntry that describes the entry. Read from the stream until the end, which is actually the end of the current entry. Then call closeEntry to read the next entry. Do not close zin until you read the last entry. Here is a typical code sequence to read through a ZIP file:

```java
var zin = new ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
    read the contents of zin
    zin.closeEntry();
}
zin.close();
```

To write a ZIP file, use a ZipOutputStream. For each entry that you want to place into the ZIP file, create a ZipEntry object. Pass the file name to the ZipEntry constructor; it sets the other parameters such as file date and decompression method. You can override these settings if you like. Then, call the putNextEntry method of the ZipOutputStream to begin writing a new file. Send the file data to the ZIP output stream. When done, call closeEntry. Repeat for all the files you want to store. Here is a code skeleton:

```java
var fout = new FileOutputStream("test.zip");
var zout = new ZipOutputStream(fout);
for all files
{
    var ze = new ZipEntry(filename);
    zout.putNextEntry(ze);
    send data to zout
    zout.closeEntry();
}
zout.close();
```
NOTE: JAR files (which were discussed in Volume I, Chapter 4) are simply ZIP files with a special entry—the so-called manifest. Use the JarInputStream and JarOutputStream classes to read and write the manifest entry.

ZIP input streams are a good example of the power of the stream abstraction. When you read data stored in compressed form, you don’t need to worry that the data are being decompressed as they are being requested. Moreover, the source of the bytes in a ZIP stream need not be a file—the ZIP data can come from a network connection.

NOTE: Section 2.4.8, “ZIP File Systems,” on p. 123 shows how to access a ZIP archive without a special API, using the FileSystem class of Java 7.

<table>
<thead>
<tr>
<th>java.util.zip.ZipInputStream 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ZipInputStream(InputStream in)</td>
</tr>
<tr>
<td>creates a ZipInputStream that allows you to inflate data from the given InputStream.</td>
</tr>
<tr>
<td>• ZipEntry getNextEntry()</td>
</tr>
<tr>
<td>returns a ZipEntry object for the next entry, or null if there are no more entries.</td>
</tr>
<tr>
<td>• void closeEntry()</td>
</tr>
<tr>
<td>closes the current open entry in the ZIP file. You can then read the next entry by using getNextEntry().</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>java.util.zip.ZipOutputStream 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ZipOutputStream(OutputStream out)</td>
</tr>
<tr>
<td>creates a ZipOutputStream that you can use to write compressed data to the specified OutputStream.</td>
</tr>
<tr>
<td>• void putNextEntry(ZipEntry ze)</td>
</tr>
<tr>
<td>writes the information in the given ZipEntry to the output stream and positions the stream for the data. The data can then be written by calling the write() method.</td>
</tr>
</tbody>
</table>

(Continues)
### java.util.zip.ZipOutputStream 1.1 (Continued)

- **void closeEntry()**
  closes the currently open entry in the ZIP file. Use the putNextEntry method to start the next entry.

- **void setLevel(int level)**
  sets the default compression level of subsequent DEFLATED entries to a value from Deflater.NO_COMPRESSION to Deflater.BEST_COMPRESSION. The default value is Deflater.DEFAULT_COMPRESSION. Throws an IllegalArgumentException if the level is not valid.

- **void setMethod(int method)**
  sets the default compression method for this ZipOutputStream for any entries that do not specify a method; can be either DEFLATED or STORED.

### java.util.zip.ZipEntry 1.1

- **ZipEntry(String name)**
  constructs a ZIP entry with a given name.

- **long getCrc()**
  returns the CRC32 checksum value for this ZipEntry.

- **String getName()**
  returns the name of this entry.

- **long getSize()**
  returns the uncompressed size of this entry, or -1 if the uncompressed size is not known.

- **boolean isDirectory()**
  returns true if this entry is a directory.

- **void setMethod(int method)**
  sets the compression method for the entry to DEFLATED or STORED.

- **void setSize(long size)**
  sets the size of this entry. Only required if the compression method is STORED.

- **void setCrc(long crc)**
  sets the CRC32 checksum of this entry. Use the CRC32 class to compute this checksum. Only required if the compression method is STORED.
2.3 Object Input/Output Streams and Serialization

Using a fixed-length record format is a good choice if you need to store data of the same type. However, objects that you create in an object-oriented program are rarely all of the same type. For example, you might have an array called staff that is nominally an array of Employee records but contains objects that are actually instances of a subclass such as Manager.

It is certainly possible to come up with a data format that allows you to store such polymorphic collections—but, fortunately, we don’t have to. The Java language supports a very general mechanism, called object serialization, that makes it possible to write any object to an output stream and read it again later. (You will see in this chapter where the term “serialization” comes from.)

2.3.1 Saving and Loading Serializable Objects

To save object data, you first need to open an ObjectOutputStream object:

```java
var out = new ObjectOutputStream(new FileOutputStream("employee.dat"));
```

Now, to save an object, simply use the writeObject method of the ObjectOutputStream class as in the following fragment:

```java
var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
var boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
```
out.writeObject(harry);
out.writeObject(boss);

To read the objects back in, first get an ObjectInputStream object:

    var in = new ObjectInputStream(new FileInputStream("employee.dat"));

Then, 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();

There is, however, one change you need to make to any class that you want to save to an output stream and restore from an object input stream. The class must implement the Serializable interface:

    class Employee implements Serializable { . . . }

The Serializable interface has no methods, so you don’t need to change your classes in any way. In this regard, it is similar to the Cloneable interface that we discussed in Volume I, Chapter 6. However, to make a class cloneable, you still had to override the clone method of the Object class. To make a class serializable, you do not need to do anything else.

NOTE: You can write and read only objects with the writeObject/readObject methods. For primitive type values, use methods such as writeInt/readInt or writeDouble/readDouble. (The object input/output stream classes implement the DataInput/DataOutput interfaces.)

Behind the scenes, an ObjectOutputStream looks at all the fields of the objects and saves their contents. For example, when writing an Employee object, the name, date, and salary fields are written to the output stream.

However, there is one important situation to consider: What happens when one object is shared by several objects as part of their state?

To illustrate the problem, let us make a slight modification to the Manager class. Let’s assume that each manager has a secretary:

    class Manager extends Employee
    {
        private Employee secretary;
        . . .
    }
Each Manager object now contains a reference to an Employee object that describes the secretary. Of course, two managers can share the same secretary, as is the case in Figure 2.5 and the following code:

```javascript
var harry = new Employee("Harry Hacker", ...);
var carl = new Manager("Carl Cracker", ...);
carl.setSecretary(harry);
var tony = new Manager("Tony Tester", ...);
tony.setSecretary(harry);
```

Figure 2.5 Two managers can share a mutual employee.

Saving such a network of objects is a challenge. Of course, we cannot save and restore the memory addresses for the secretary objects. When an object is reloaded, it will likely occupy a completely different memory address than it originally did.

Instead, each object is saved with the serial number—hence the name object serialization for this mechanism. Here is the algorithm:

1. Associate a serial number with each object reference that you encounter (as shown in Figure 2.6).
2. When encountering an object reference for the first time, save the object data to the output stream.

3. If it has been saved previously, just write “same as the previously saved object with serial number $x$.”

When reading the objects back, the procedure is reversed.

1. When an object is specified in an object input stream for the first time, construct it, initialize it with the stream data, and remember the association between the serial number and the object reference.

2. When the tag “same as the previously saved object with serial number $x$” is encountered, retrieve the object reference for the sequence number.
NOTE: In this chapter, we will use serialization to save a collection of objects to a disk file and retrieve it exactly as we stored it. Another very important application is the transmittal of a collection of objects across a network connection to another computer. Just as raw memory addresses are meaningless in a file, they are also meaningless when you communicate with a different processor. By replacing memory addresses with serial numbers, serialization permits the transport of object collections from one machine to another.

Listing 2.3 is a program that saves and reloads a network of Employee and Manager objects (some of which share the same employee as a secretary). Note that the secretary object is unique after reloading—when newStaff[1] gets a raise, that is reflected in the secretary fields of the managers.

```
package objectStream;

import java.io.*;

/**
 * @version 1.11 2018-05-01
 * @author Cay Horstmann
 */

class ObjectStreamTest {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
        var carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
        carl.setSecretary(harry);
        var tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
        tony.setSecretary(harry);

        var staff = new Employee[3];
        staff[0] = carl;
        staff[1] = harry;
        staff[2] = tony;

        // save all employee records to the file employee.dat
        try (var out = new ObjectOutputStream(new FileOutputStream("employee.dat"))) {
            out.writeObject(staff);
        }
    }
}
```
```java
try (var in = new ObjectInputStream(new FileInputStream("employee.dat")))
{
    // retrieve all records into a new array
    var newStaff = (Employee[]) in.readObject();

    // raise secretary's salary
    newStaff[1].raiseSalary(10);

    // print the newly read employee records
    for (Employee e : newStaff)
        System.out.println(e);
}
```

---

### java.io.ObjectOutputStream

- **ObjectOutputStream(OutputStream out)** creates an `ObjectOutputStream` so you can write objects to the specified `OutputStream`.  
- **void writeObject(Object obj)** writes the specified object to the `ObjectOutputStream`. This method saves the class of the object, the signature of the class, and the values of any nonstatic, nontransient fields of the class and its superclasses.

---

### java.io.ObjectInputStream

- **ObjectInputStream(InputStream in)** creates an `ObjectInputStream` to read back object information from the specified `InputStream`.  
- **Object readObject()** reads an object from the `ObjectInputStream`. In particular, this method reads back the class of the object, the signature of the class, and the values of the nontransient and nonstatic fields of the class and all its superclasses. It does deserializing so that multiple object references can be recovered.

---

### 2.3.2 Understanding the Object Serialization File Format

Object serialization saves object data in a particular file format. Of course, you can use the `writeObject/readObject` methods without having to know the exact sequence of bytes that represents objects in a file. Nonetheless, we found studying the data format extremely helpful for gaining insight into the object
serialization process. As the details are somewhat technical, feel free to skip this section if you are not interested in the implementation.

Every file begins with the two-byte “magic number”

AC ED

followed by the version number of the object serialization format, which is currently

00 05

(We use hexadecimal numbers throughout this section to denote bytes.) Then, it contains a sequence of objects, in the order in which they were saved.

String objects are saved as

74 two-byte characters

length

For example, the string “Harry” is saved as

74 00 05 Harry

The Unicode characters of the string are saved in the “modified UTF-8” format.

When an object is saved, the class of that object must be saved as well. The class description contains

• The name of the class
• The serial version unique ID, which is a fingerprint of the data field types and method signatures
• A set of flags describing the serialization method
• A description of the data fields

The fingerprint is obtained by ordering the descriptions of the class, superclass, interfaces, field types, and method signatures in a canonical way, and then applying the so-called Secure Hash Algorithm (SHA) to that data.

SHA is a fast algorithm that gives a “fingerprint” of a larger block of information. This fingerprint is always a 20-byte data packet, regardless of the size of the original data. It is created by a clever sequence of bit operations on the data that makes it essentially 100 percent certain that the fingerprint will change if the information is altered in any way. (For more details on SHA, see, for example, Cryptography and Network Security, Seventh Edition by William Stallings, Prentice Hall, 2016.) However, the serialization mechanism uses only the first eight bytes of the SHA code as a class fingerprint. It is still very likely that the class fingerprint will change if the data fields or methods change.
When reading an object, its fingerprint is compared against the current fingerprint of the class. If they don’t match, it means the class definition has changed after the object was written, and an exception is generated. Of course, in practice, classes do evolve, and it might be necessary for a program to read in older versions of objects. We will discuss this in Section 2.3.5, “Versioning,” on p. 103.

Here is how a class identifier is stored:

- 72
- 2-byte length of class name
- Class name
- 8-byte fingerprint
- 1-byte flag
- 2-byte count of data field descriptors
- Data field descriptors
- 78 (end marker)
- Superclass type (70 if none)

The flag byte is composed of three bit masks, defined in java.io.ObjectStreamConstants:

```java
static final byte SC_WRITE_METHOD = 1;
    // class has a writeObject method that writes additional data
static final byte SC_SERIALIZABLE = 2;
    // class implements the Serializable interface
static final byte SC_EXTERNALIZABLE = 4;
    // class implements the Externalizable interface
```

We discuss the Externalizable interface later in this chapter. Externalizable classes supply custom read and write methods that take over the output of their instance fields. The classes that we write implement the Serializable interface and will have a flag value of 02. The serializable java.util.Date class defines its own readObject/writeObject methods and has a flag of 03.

Each data field descriptor has the format:

- 1-byte type code
- 2-byte length of field name
- Field name
- Class name (if the field is an object)

The type code is one of the following:
When the type code is L, the field name is followed by the field type. Class and field name strings do not start with the string code 74, but field types do. Field types use a slightly different encoding of their names—namely, the format used by native methods.

For example, the salary field of the Employee class is encoded as

D 00 06 salary

Here is the complete class descriptor of the Employee class:

72 00 08 Employee
   E6 D2 06 7D AE AC 18 1B 02   Fingerprint and flags
   00 03   Number of instance fields
   D 00 06 salary   Instance field type and name
   L 00 07 hireDay   Instance field type and name
   74 00 10 Ljava/util/Date;   Instance field class name: Date
   L 00 04 name   Instance field type and name
   74 00 12 Ljava/lang/String;   Instance field class name: String
   78   End marker
   70   No superclass

These descriptors are fairly long. If the same class descriptor is needed again in the file, an abbreviated form is used:

71   4-byte serial number

The serial number refers to the previous explicit class descriptor. We discuss the numbering scheme later.

An object is stored as

73   class descriptor   object data
For example, here is how an Employee object is stored:

- **Salary field value**: double
  - `40 E8 6A 00 00 00 00 00`  
  - `salary field value: double`
- **Hire day field value**: new object
  - `73`  
  - `hireDay field value: new object`
- **Existing class java.util.Date**
  - `71 00 7E 00 08`  
  - `Existing class java.util.Date`
- **External storage (details later)**
  - `77 08 00 00 00 91 1B 4E B1 80 78`  
  - `External storage (details later)`
- **Name field value**: String
  - `74 00 0C Harry Hacker`  
  - `name field value: String`

As you can see, the data file contains enough information to restore the Employee object.

Arrays are saved in the following format:

- **Class descriptor**: 4-byte number of entries
  - `75`  
  - `class descriptor 4-byte number of entries`

The array class name in the class descriptor is in the same format as that used by native methods (which is slightly different from the format used by class names in other class descriptors). In this format, class names start with an `L` and end with a semicolon.

For example, an array of three Employee objects starts out like this:

- **Array**
  - `75`  
  - `Array`
  - `72 00 0B [LEmployee;`  
  - `New class, string length, class name Employee[]`
  - `FC BF 36 11 C5 91 11 C7 02`  
  - `Fingerprint and flags`
  - `00 00`  
  - `Number of instance fields`
  - `78`  
  - `End marker`
  - `70`  
  - `No superclass`
  - `00 00 00 03`  
  - `Number of array entries`

Note that the fingerprint for an array of Employee objects is different from a fingerprint of the Employee class itself.

All objects (including arrays and strings) and all class descriptors are given serial numbers as they are saved in the output file. The numbers start at `00 7E 00 00`.

We already saw that a full class descriptor for any given class occurs only once. Subsequent descriptors refer to it. For example, in our previous example, a repeated reference to the Date class was coded as

- `71 00 7E 00 08`  
  - `71 00 7E 00 08`  

The same mechanism is used for objects. If a reference to a previously saved object is written, it is saved in exactly the same way—that is, `71` followed by
the serial number. It is always clear from the context whether a particular serial reference denotes a class descriptor or an object.

Finally, a null reference is stored as

Here is the commented output of the `ObjectRefTest` program of the preceding section. Run the program, look at a hex dump of its data file `employee.dat`, and compare it with the commented listing. The important lines toward the end of the output show a reference to a previously saved object.

```
AC ED 00 05  File header
75
72 00 00 [LEmployee;  New class, string length, class name
           Employee[] (serial #0)
FC BF 36 11 C5 91 11 C7 02  Fingerprint and flags
00 00
78
70
00 00 00 03
73
72 00 07 Manager

36 06 AE 13 63 8F 59 B7 02  Fingerprint and flags
00 01
L 00 09 secretary
74 00 0A LEmployee;
78
72 00 08 Employee

E6 D2 86 7D AE AC 18 1B 02  Fingerprint and flags
00 03
D 00 06 salary
L 00 07 hireDay
74 00 10 Ljava/util/Date;
L 00 04 name
74 00 12 Ljava/lang/String;
78
70
No superclass
```
Of course, studying these codes can be about as exciting as reading a phone book. It is not important to know the exact file format (unless you are trying to create an evil effect by modifying the data), but it is still instructive to
know that the serialized format has a detailed description of all the objects it contains, with sufficient detail to allow reconstruction of both objects and arrays of objects.

What you should remember is this:

- The serialized format contains the types and data fields of all objects.
- Each object is assigned a serial number.
- Repeated occurrences of the same object are stored as references to that serial number.

### 2.3.3 Modifying the Default Serialization Mechanism

Certain data fields should never be serialized—for example, integer values that store file handles or handles of windows that are only meaningful to native methods. Such information is guaranteed to be useless when you reload an object at a later time or transport it to a different machine. In fact, improper values for such fields can actually cause native methods to crash. Java has an easy mechanism to prevent such fields from ever being serialized: Mark them with the keyword `transient`. You also need to tag fields as transient if they belong to nonserializable classes. Transient fields are always skipped when objects are serialized.

The serialization mechanism provides a way for individual classes to add validation or any other desired action to the default read and write behavior. A serializable class can define methods with the signature

```java
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
    throws IOException;
```

Then, the data fields are no longer automatically serialized—these methods are called instead.

Here is a typical example. 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`.

```java
public class LabeledPoint implements Serializable {
    private String label;
    private transient Point2D.Double point;
    . . .
}
```
In the `writeObject` method, we 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.

```java
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:

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

Another example is the `java.util.Date` class that supplies its own `readObject` and `writeObject` methods. These methods write the date as a number of milliseconds from the epoch (January 1, 1970, midnight UTC). The `Date` class has a complex internal representation that stores both a `Calendar` object and a millisecond count to optimize lookups. The state of the `Calendar` is redundant and does not have to be saved.

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

Instead of letting the serialization mechanism save and restore object data, a class can define its own mechanism. To do this, a class must implement the `Externalizable` interface. This, in turn, requires it to define two methods:

```java
public void readExternal(ObjectInputStream in)
  throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutputStream out)
  throws IOException;
```

Unlike the `readObject` and `writeObject` methods that were described in the previous section, 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. Here is how you can implement these methods for the Employee class:

```java
public void readExternal(ObjectInput s)  
    throws IOException  
{
    name = s.readUTF();
    salary = s.readDouble();
    hireDay = LocalDate.ofEpochDay(s.readLong());
}

public void writeExternal(ObjectOutput s)  
    throws IOException  
{
    s.writeUTF(name);
    s.writeDouble(salary);
    s.writeLong(hireDay.toEpochDay());
}
```

**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.

### 2.3.4 Serializing Singletons and Typesafe Enumerations

You have to pay particular attention to serializing and deserializing objects that are assumed to be unique. This commonly happens when you are implementing singletons and typesafe enumerations.

If you use the `enum` construct of the Java language, you need not worry about serialization—it just works. However, suppose you maintain legacy code that contains an enumerated type such as

```java
public class Orientation
{
    public static final Orientation HORIZONTAL = new Orientation(1);
    public static final Orientation VERTICAL  = new Orientation(2);

    private int value;

    private Orientation(int v) { value = v; }
}
```
This idiom was common before enumerations were added to the Java language. Note that the constructor is private. Thus, no objects can be created beyond `Orientation.HORIZONTAL` and `Orientation.VERTICAL`. In particular, you can use the `==` operator to test for object equality:

```java
if (orientation == Orientation.HORIZONTAL) . . .
```

There is an important twist that you need to remember when a typesafe enumeration implements the `Serializable` interface. The default serialization mechanism is not appropriate. Suppose we write a value of type `Orientation` and read it in again:

```java
Orientation original = Orientation.HORIZONTAL;
ObjectOutputStream out = . . .;
out.write(original);
out.close();
ObjectInputStream in = . . .;
var saved = (Orientation) in.read();
```

Now the test

```java
if (saved == Orientation.HORIZONTAL) . . .
```

will fail. In fact, the `saved` value is a completely new object of the `Orientation` type that is not equal to any of the predefined constants. Even though the constructor is private, the serialization mechanism can create new objects!

To solve this problem, you need to define another special serialization method, called `readResolve`. If the `readResolve` method is defined, it is called after the object is deserialized. It must return an object which then becomes the return value of the `readObject` method. In our case, the `readResolve` method will inspect the value field and return the appropriate enumerated constant:

```java
protected Object readResolve() throws ObjectStreamException
{
    if (value == 1) return Orientation.HORIZONTAL;
    if (value == 2) return Orientation.VERTICAL;
    throw new ObjectStreamException(); // this shouldn't happen
}
```

Remember to add a `readResolve` method to all typesafe enumerations in your legacy code and to all classes that follow the singleton design pattern.

### 2.3.5 Versioning

If you use serialization to save objects, you need to consider what happens when your program evolves. Can version 1.1 read the old files? Can the users who still use 1.0 read the files that the new version is producing? Clearly, it would be desirable if object files could cope with the evolution of classes.
At first glance, it seems that this would not be possible. When a class definition changes in any way, its SHA fingerprint also changes, and you know that object input streams will refuse to read in objects with different fingerprints. However, a class can indicate that it is *compatible* with an earlier version of itself. To do this, you must first obtain the fingerprint of the *earlier* version of the class. Use the standalone `serialver` program that is part of the JDK to obtain this number. For example, running

```
serialver Employee
```

prints

```
Employee: static final long serialVersionUID = -1814239825517340645L;
```

All *later* versions of the class must define the `serialVersionUID` constant to the same fingerprint as the original.

```java
class Employee implements Serializable // version 1.1
{
    . . .
    public static final long serialVersionUID = -1814239825517340645L;
}
```

When a class has a static data member named `serialVersionUID`, it will not compute the fingerprint manually but will use that value instead.

Once that static data member has been placed inside a class, the serialization system is now willing to read in different versions of objects of that class.

If only the methods of the class change, there is no problem with reading the new object data. However, if the data fields change, you may have problems. For example, the old file object may have more or fewer data fields than the one in the program, or the types of the data fields may be different. In that case, the object input stream makes an effort to convert the serialized object to the current version of the class.

The object input stream compares the data fields of the current version of the class with those of the version in the serialized object. Of course, the object input stream considers only the nontransient and nonstatic data fields. If two fields have matching names but different types, the object input stream makes no effort to convert one type to the other—the objects are incompatible. If the serialized object has data fields that are not present in the current version, the object input stream ignores the additional data. If the current version has data fields that are not present in the serialized object, the added fields are set to their default (null for objects, zero for numbers, and false for boolean values).
Here is an example. Suppose we have saved a number of employee records on disk, using the original version (1.0) of the class. Now we change the Employee class to version 2.0 by adding a data field called department. Figure 2.7 shows what happens when a 1.0 object is read into a program that uses 2.0 objects. The department field is set to null. Figure 2.8 shows the opposite scenario: A program using 1.0 objects reads a 2.0 object. The additional department field is ignored.

**Figure 2.7** Reading an object with fewer data fields

**Figure 2.8** Reading an object with more data fields

Is this process safe? It depends. Dropping a data field seems harmless—the recipient still has all the data that it knows how to manipulate. Setting a data field to null might not be so safe. Many classes work hard to initialize all
data fields in all constructors to non-null values, so that the methods don’t have to be prepared to handle null data. It is up to the class designer to implement additional code in the readObject method to fix version incompatibilities or to make sure the methods are robust enough to handle null data.

**TIP:** Before you add a serialVersionUID field to a class, ask yourself why you made your class serializable. If serialization is used only for short-term persistence, such as distributed method calls in an application server, there is no need to worry about versioning and the serialVersionUID. The same applies if you extend a class that happens to be serializable, but you have no intent to ever persist its instances. If your IDE gives you pesky warnings, change the IDE preferences to turn them off, or add an annotation @SuppressWarnings("serial"). This is safer than adding a serialVersionUID that you may later forget to change.

### 2.3.6 Using Serialization for Cloning

There is an amusing use for the serialization mechanism: It gives you an easy way to clone an object, provided the class is serializable. Simply serialize it to an output stream and then read it back in. The result is a new object that is a deep copy of the existing object. You don’t have to write the object to a file—you can use a ByteArrayOutputStream to save the data into a byte array.

As Listing 2.4 shows, to get clone for free, simply extend the SerialCloneable class, and you are done.

You should be aware that this method, although clever, will usually be much slower than a clone method that explicitly constructs a new object and copies or clones the data fields.

#### Listing 2.4 serialClone/SerialCloneTest.java

```java
package serialClone;

import java.io.*;
import java.time.*;

public class SerialCloneTest {
    public static void main(String[] args) throws CloneNotSupportedException {
        // Implementation
    }
}
```
var harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
// clone harry
var harry2 = (Employee) harry.clone();

// mutate harry
harry.raiseSalary(10);

// now harry and the clone are different
System.out.println(harry);
System.out.println(harry2);
}
}

/**
 * A class whose clone method uses serialization.
 */
class SerialCloneable implements Cloneable, Serializable {
    public Object clone() throws CloneNotSupportedException {
        try {
            // save the object to a byte array
            var bout = new ByteArrayOutputStream();
            try (var out = new ObjectOutputStream(bout)) {
                out.writeObject(this);
            }

            // read a clone of the object from the byte array
            try (var bin = new ByteArrayInputStream(bout.toByteArray())) {
                var in = new ObjectInputStream(bin);
                return in.readObject();
            }
        }

        catch (IOException | ClassNotFoundException e) {
            var e2 = new CloneNotSupportedException();
            e2.initCause(e);
            throw e2;
        }
    }
}

/**
 * The familiar Employee class, redefined to extend the
 * SerialCloneable class.
 */
class Employee extends SerialClonable {
    private String name;
    private double salary;
    private LocalDate hireDay;
    
    public Employee(String n, double s, int year, int month, int day) {
        name = n;
        salary = s;
        hireDay = LocalDate.of(year, month, day);
    }
    
    public String getName() {
        return name;
    }
    
    public double getSalary() {
        return salary;
    }
    
    public LocalDate getHireDay() {
        return hireDay;
    }
    
    /**
     * Raises the salary of this employee.
     * @byPercent the percentage of the raise
     */
    public void raiseSalary(double byPercent) {
        double raise = salary * byPercent / 100;
        salary += raise;
    }
    
    public String toString() {
        return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay + "]";
    }
}
2.4 Working with Files

You have learned how to read and write data from a file. However, there is more to file management than reading and writing. The Path interface and Files class encapsulate the functionality required to work with the file system on the user’s machine. For example, the Files class can be used to remove or rename a file, or to find out when a file was last modified. In other words, the input/output stream classes are concerned with the contents of files, whereas the classes that we discuss here are concerned with the storage of files on a disk.

The Path interface and Files class were added in Java 7. They are much more convenient to use than the File class which dates back all the way to JDK 1.0. We expect them to be very popular with Java programmers and discuss them in depth.

2.4.1 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 a UNIX-like file system.

Path absolute = Paths.get("/home", "harry");
Path relative = Paths.get("myprog", "conf", "user.properties");

The static Paths.get 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.

The get method can get a single string containing multiple components. For example, you can read a path from a configuration file like this:

String baseDir = props.getProperty("base.dir");
    // May be a string such as /opt/myprog or c:\Program Files\myprog
Path basePath = Paths.get(baseDir);  // OK that baseDir has separators

NOTE: A path does not have to correspond to a file that actually exists. It is merely an abstract sequence of names. As you will see in the next section, when you want to create a file, you first make a path and then call a method to create the corresponding file.
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`.
- Otherwise, the result is “`p` then `q`,” according to the rules of the file system.

For example, suppose your application needs to find its working directory relative to a given base directory that is read from a configuration file, as in the preceding example.

```java
Path workRelative = Paths.get("work");
Path workPath = basePath.resolve(workRelative);
```

There is a shortcut for the `resolve` method that takes a string instead of a path:

```java
Path workPath = basePath.resolve("work");
```

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

```java
Path tempPath = workPath.resolveSibling("temp");
```

creates `/opt/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, relativizing `/home/harry` against `/home/fred/input.txt` yields `../fred/input.txt`. Here, we assume that `..` denotes the parent directory in the file system.

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

The `toAbsolutePath` method yields the absolute path of a given path, starting at a root component, such as `/home/fred/input.txt` or `c:\Users\fred\input.txt`.

The `Path` interface has many useful methods for taking paths apart. This code sample shows some of the most useful ones:

```java
Path p = Paths.get("/home", "fred", "myprog.properties");
Path parent = p.getParent(); // the path /home/fred
Path file = p.getFileName(); // the path myprog.properties
Path root = p.getRoot(); // the path /
```

As you have already seen in Volume I, you can construct a Scanner from a Path object:

```java
var in = new Scanner(Paths.get("/home/fred/input.txt"));
```
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.

**java.nio.file.Paths**

- static Path get(String first, String... more)
  makes a path by joining the given strings.

**java.nio.file.Path**

- Path resolve(Path other)
- Path resolve(String other)
  if other is absolute, returns other; otherwise, returns the path obtained by joining this and other.
- Path resolveSibling(Path other)
- Path resolveSibling(String other)
  if other is absolute, returns other; otherwise, returns the path obtained by joining the parent of this and other.
- Path relativize(Path other)
  returns the relative path that, when resolved with this, yields other.
- Path normalize()
  removes redundant path elements such as . and ..
- Path toAbsolutePath()
  returns an absolute path that is equivalent to this path.
- Path getParent()
  returns the parent, or null if this path has no parent.
- Path getFileName()
  returns the last component of this path, or null if this path has no components.
- Path getRoot()
  returns the root component of this path, or null if this path has no root components.
- toFile()
  makes a File from this path.
2.4.2 Reading and Writing Files

The Files class makes quick work of common file operations. For example, you can easily read the entire contents of a file:

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

If you want to read the file as a string, call `readAllBytes` followed by

```java
var content = new String(bytes, charset);
```

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

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

Conversely, if you want to write a string, call

```java
Files.write(path, content.getBytes(charset));
```

To append to a given file, use

```java
Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);
```

You can also write a collection of lines with

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

These simple methods are intended for dealing with text files of moderate length. If your files are large or binary, you can still use the familiar input/output streams or readers/writers:

```java
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader in = Files.newBufferedReader(path, charset);
Writer out = Files.newBufferedWriter(path, charset);
```

These convenience methods save you from dealing with `FileInputStream`, `FileOutputStream`, `BufferedReader`, or `BufferedWriter`. 
2.4.3 Creating Files and Directories

To create a new directory, call

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

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

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

You can create an empty file with

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

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

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

```java
Path newPath = Files.createTempFile(dir, prefix, suffix);
Path newPath = Files.createTempFile(prefix, suffix);
Path newPath = Files.createTempDirectory(dir, prefix);
Path newPath = 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`.

When you create a file or directory, you can specify attributes, such as owners or permissions. However, the details depend on the file system, and we won’t cover them here.

```java
java.nio.file.Files
• static Path createFile(Path path, FileAttribute<?>... attrs)
• static Path createDirectory(Path path, FileAttribute<?>... attrs)
• static Path createDirectories(Path path, FileAttribute<?>... attrs)
  creates a file or directory. The createDirectories method creates any intermediate
directories as well.
• static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)
• static Path createTempFile(Path parentDir, String prefix, String suffix, FileAttribute<?>... attrs)
• static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)
• static Path createTempDirectory(Path parentDir, String prefix, FileAttribute<?>... attrs)
  creates a temporary file or directory, in a location suitable for temporary files
or in the given parent directory. Returns the path to the created file or directory.
```

### 2.4.4 Copying, Moving, and Deleting Files

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

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

To move the file (that is, copy and delete the original), call

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

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:

```java
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:

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

You can also copy an input stream to a `Path`, which just means saving the input stream to disk. Similarly, you can copy a `Path` to an output stream. Use the following calls:

```java
Files.copy(inputStream, toPath);
Files.copy(fromPath, outputStream);
```

As with the other calls to `copy`, you can supply copy options as needed.

Finally, to delete a file, simply call

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

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

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

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

See Table 2.3 for a summary of the options that are available for file operations.

```
<table>
<thead>
<tr>
<th>java.nio.file.Files 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>• static Path copy(Path from, Path to, CopyOption... options)</td>
</tr>
<tr>
<td>• static Path move(Path from, Path to, CopyOption... options)</td>
</tr>
<tr>
<td>copies or moves from to the given target location and returns to.</td>
</tr>
<tr>
<td>• static long copy(InputStream from, Path to, CopyOption... options)</td>
</tr>
<tr>
<td>• static long copy(Path from, OutputStream to, CopyOption... options)</td>
</tr>
<tr>
<td>copies from an input stream to a file, or from a file to an output stream, returning the number of bytes copied.</td>
</tr>
<tr>
<td>• static void delete(Path path)</td>
</tr>
<tr>
<td>• static boolean deleteIfExists(Path path)</td>
</tr>
<tr>
<td>deletes the given file or empty directory. The first method throws an exception if the file or directory doesn’t exist. The second method returns false in that case.</td>
</tr>
</tbody>
</table>
```
Table 2.3 Standard Options for File Operations

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StandardOpenOption; use with newBufferedWriter, newInputStream, newOutputStream, write</td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td>Open for reading</td>
</tr>
<tr>
<td>WRITE</td>
<td>Open for writing</td>
</tr>
<tr>
<td>APPEND</td>
<td>If opened for writing, append to the end of the file</td>
</tr>
<tr>
<td>TRUNCATE_EXISTING</td>
<td>If opened for writing, remove existing contents</td>
</tr>
<tr>
<td>CREATE_NEW</td>
<td>Create a new file and fail if it exists</td>
</tr>
<tr>
<td>CREATE</td>
<td>Atomically create a new file if it doesn’t exist</td>
</tr>
<tr>
<td>DELETE_ON_CLOSE</td>
<td>Make a “best effort” to delete the file when it is closed</td>
</tr>
<tr>
<td>SPARSE</td>
<td>A hint to the file system that this file will be sparse</td>
</tr>
<tr>
<td>DSYNC or SYNC</td>
<td>Requires that each update to the file data or data and metadata be written synchronously to the storage device</td>
</tr>
<tr>
<td>StandardCopyOption; use with copy, move</td>
<td></td>
</tr>
<tr>
<td>ATOMIC_MOVE</td>
<td>Move the file atomically</td>
</tr>
<tr>
<td>COPY_ATTRIBUTES</td>
<td>Copy the file attributes</td>
</tr>
<tr>
<td>REPLACE_EXISTING</td>
<td>Replace the target if it exists</td>
</tr>
<tr>
<td>LinkOption; use with all of the above methods and exists, isDirectory, isRegularFile</td>
<td></td>
</tr>
<tr>
<td>NOFOLLOW_LINKS</td>
<td>Do not follow symbolic links</td>
</tr>
<tr>
<td>FileVisitOption; use with find, walk, walkFileTree</td>
<td></td>
</tr>
<tr>
<td>FOLLOW_LINKS</td>
<td>Follow symbolic links</td>
</tr>
</tbody>
</table>

2.4.5 Getting File Information

The following static methods return a boolean value to check a property of a path:

- exists
- isHidden
- isReadable, isWritable, isExecutable
- isRegularFile, isDirectory, isSymbolicLink

The size method returns the number of bytes in a file.

```java
long fileSize = Files.size(path);
```
The `getOwner` method returns the owner of the file, as an instance of `java.nio.file.attribute.UserPrincipal`.

All file systems report a set of basic attributes, encapsulated by the `BasicFileAttributes` interface, which partially overlaps with that information. The basic file attributes are:

- The times at which the file was created, last accessed, and last modified, as instances of the class `java.nio.file.attribute.FileTime`
- Whether the file is a regular file, a directory, a symbolic link, or none of these
- The file size
- The file key—an object of some class, specific to the file system, that may or may not uniquely identify a file

To get these attributes, call:

```java
BasicFileAttributes attributes = Files.readAttributes(path, BasicFileAttributes.class);
```

If you know that the user’s file system is POSIX-compliant, you can instead get an instance of `PosixFileAttributes`:

```java
PosixFileAttributes attributes = Files.readAttributes(path, PosixFileAttributes.class);
```

Then you can find out the group owner and the owner, group, and world access permissions of the file. We won’t dwell on the details since so much of this information is not portable across operating systems.

---

### java.nio.file.Files

- `static boolean exists(Path path)`
- `static boolean isHidden(Path path)`
- `static boolean isReadable(Path path)`
- `static boolean isWritable(Path path)`
- `static boolean isExecutable(Path path)`
- `static boolean isRegularFile(Path path)`
- `static boolean isDirectory(Path path)`
- `static boolean isSymbolicLink(Path path)`
  - checks for the given property of the file given by the path.
- `static long size(Path path)`
  - gets the size of the file in bytes.
- `A readAttributes(Path path, Class<A> type, LinkOption... options)`
  - reads the file attributes of type A.
2.4.6 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 block:

```java
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.

```java
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
```
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:
```java
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 delete the children before deleting the parent. The next section shows you how to overcome that problem.
2.4.7 Using Directory Streams

As you saw in the preceding section, the Files.walk method produces a Stream<Path> that traverses the descendants of a directory. Sometimes, you need more fine-grained control over the traversal process. In that case, use the Files.newDirectoryStream object instead. It yields a DirectoryStream. Note that this is not a subinterface of java.util.stream.Stream but an interface that is specialized for directory traversal. It is a subinterface of Iterable so that you can use directory stream in an enhanced for loop. Here is the usage pattern:

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir)) {
    for (Path entry : entries)
        Process entries
}
```

The try-with-resources block ensures that the directory stream is properly closed.

There is no specific order in which the directory entries are visited.

You can filter the files with a glob pattern:

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir, "*.java"))
```

Table 2.4 shows all glob patterns.

**Table 2.4 Glob Patterns**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Matches zero or more characters of a path component.</td>
<td>*.java matches all Java files in the current directory.</td>
</tr>
<tr>
<td>**</td>
<td>Matches zero or more characters, crossing directory boundaries.</td>
<td>**.java matches all Java files in any subdirectory.</td>
</tr>
<tr>
<td>?</td>
<td>Matches one character.</td>
<td>???.java matches all four-character (not counting the extension) Java files.</td>
</tr>
<tr>
<td>[ . . ]</td>
<td>Matches a set of characters. You can use hyphens [0-9] and negation [!0-9].</td>
<td>Test[0-9A-F].java matches Test(x).java, where (x) is one hexadecimal digit.</td>
</tr>
<tr>
<td>{ . . }</td>
<td>Matches alternatives, separated by commas.</td>
<td>*.{java,class} matches all Java and class files.</td>
</tr>
<tr>
<td>\</td>
<td>Escapes any of the above as well as .</td>
<td>** matches all files with a * in their name.</td>
</tr>
</tbody>
</table>
CAUTION: If you use the glob syntax on Windows, you have to escape backslashes twice: once for the glob syntax, and once for the Java string syntax: `Files.newDirectoryStream(dir, "C:\\\")`.

If you want to visit all descendants of a directory, call the `walkFileTree` method instead and supply an object of type `FileVisitor`. That object gets notified

- When a file is encountered: `FileVisitResult visitFile(T path, BasicFileAttributes attrs)`
- Before a directory is processed: `FileVisitResult preVisitDirectory(T dir, IOException ex)`
- After a directory is processed: `FileVisitResult postVisitDirectory(T dir, IOException ex)`
- When an error occurred trying to visit a file or directory, such as trying to open a directory without the necessary permissions: `FileVisitResult visitFileFailed(T path, IOException ex)`

In each case, you can specify whether you want to

- 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.

NOTE: The `FileVisitor` interface is a generic type, but it isn’t likely that you’ll ever want something other than a `FileVisitor<Path>`. The `walkFileTree` method is willing to accept a `FileVisitor<? super Path>`, but Path does not have an abundance of supertypes.

A convenience class `SimpleFileVisitor` implements the `FileVisitor` interface. All methods except `visitFileFailed` do nothing and continue. The `visitFileFailed` method throws the exception that caused the failure, thereby terminating the visit.

For example, here is how to print out all subdirectories of a given directory:
Files.walkFileTree(Paths.get("/"), new SimpleFileVisitor<Path>()
{
    public FileVisitResult preVisitDirectory(Path path, BasicFileAttributes attrs)
        throws IOException
    {
        System.out.println(path);
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult postVisitDirectory(Path dir, IOException exc)
    {
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult visitFileFailed(Path path, IOException exc)
        throws IOException
    {
        return FileVisitResult.SKIP_SUBTREE;
    }
});

Note that we need to override postVisitDirectory and visitFileFailed. Otherwise, the visit would fail as soon as it encounters a directory that it’s not allowed to open or a file it’s not allowed to access.

Also note that the attributes of the path are passed as a parameter to the preVisitDirectory and visitFile methods. The visitor already had to make an OS call to get the attributes, since it needs to distinguish between files and directories. This way, you don’t need to make another call.

The other methods of the FileVisitor interface are useful if you need to do some work when entering or leaving a directory. For example, when you delete a directory tree, you need to remove the current directory after you have removed all of its files. Here is the complete code for deleting a directory tree:

// Delete the directory tree starting at root
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 e) throws IOException {
    if (e != null) throw e;
    Files.delete(dir);
    return FileVisitResult.CONTINUE;
}

2.4.8 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 zipfile is the name of a ZIP file, then the call

FileSystem fs = FileSystems.newFileSystem(Paths.get(zipfile), null);

<table>
<thead>
<tr>
<th>java.nio.file.Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>static DirectoryStream&lt;Path&gt; newDirectoryStream(Path path)</td>
</tr>
<tr>
<td>static DirectoryStream&lt;Path&gt; newDirectoryStream(Path path, String glob)</td>
</tr>
<tr>
<td>gets an iterator over the files and directories in a given directory. The second method only accepts those entries matching the given glob pattern.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>java.nio.file.SimpleFileVisitor&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>static FileVisitResult visitFile(T path, BasicFileAttributes attrs)</td>
</tr>
<tr>
<td>is called when a file or directory is visited; returns one of CONTINUE, SKIP_SUBTREE, SKIP_SIBLINGS, or TERMINATE. The default implementation does nothing and continues.</td>
</tr>
<tr>
<td>static FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)</td>
</tr>
<tr>
<td>static FileVisitResult postVisitDirectory(T dir, BasicFileAttributes attrs)</td>
</tr>
<tr>
<td>are called before and after visiting a directory. The default implementation does nothing and continues.</td>
</tr>
<tr>
<td>static FileVisitResult visitFileFailed(T path, IOException exc)</td>
</tr>
<tr>
<td>is called if an exception was thrown in an attempt to get information about the given file. The default implementation rethrows the exception, which causes the visit to terminate with that exception. Override the method if you want to continue.</td>
</tr>
</tbody>
</table>
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:

```java
Files.copy(fs.getPath(sourceName), targetPath);
```

Here, `fs.getPath` is the analog of `Paths.get` for an arbitrary file system.

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

```java
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
Files.walkFileTree(fs.getPath("/"), new SimpleFileVisitor<Path>()
    {
        public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
            throws IOException
        {
            System.out.println(file);
            return FileVisitResult.CONTINUE;
        }
    });
```

That is nicer than the API described in Section 2.2.3, “ZIP Archives,” on p. 85 which required a set of new classes just to deal with ZIP archives.

### Memory-Mapped Files

Most operating systems can take advantage of a virtual memory implementation to “map” a file, or a region of a file, into memory. Then the file can be accessed as if it were an in-memory array, which is much faster than the traditional file operations.
2.5.1 Memory-Mapped File Performance

At the end of this section, you can find a program that computes the CRC32 checksum of a file using traditional file input and a memory-mapped file. On one machine, we got the timing data shown in Table 2.5 when computing the checksum of the 37MB file rt.jar in the jre/lib directory of the JDK.

Table 2.5 Timing Data for File Operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain input stream</td>
<td>110 seconds</td>
</tr>
<tr>
<td>Buffered input stream</td>
<td>9.9 seconds</td>
</tr>
<tr>
<td>Random access file</td>
<td>162 seconds</td>
</tr>
<tr>
<td>Memory-mapped file</td>
<td>7.2 seconds</td>
</tr>
</tbody>
</table>

As you can see, on this particular machine, memory mapping is a bit faster than using buffered sequential input and dramatically faster than using a RandomAccessFile.

Of course, the exact values will differ greatly from one machine to another, but it is obvious that the performance gain, compared to random access, can be substantial. For sequential reading of files of moderate size, on the other hand, there is no reason to use memory mapping.

The java.nio package makes memory mapping quite simple. Here is what you do.

First, get a channel for the file. A channel is an abstraction for a disk file that lets you access operating system features such as memory mapping, file locking, and fast data transfers between files.

```java
FileChannel channel = FileChannel.open(path, options);
```

Then, get a ByteBuffer from the channel by calling the map method of the FileChannel class. Specify the area of the file that you want to map and a mapping mode. Three modes are supported:

- `FileChannel.MapMode.READ_ONLY`: The resulting buffer is read-only. Any attempt to write to the buffer results in a ReadOnlyBufferException.
- `FileChannel.MapMode.READ_WRITE`: The resulting buffer is writable, and the changes will be written back to the file at some time. Note that other programs that have mapped the same file might not see those changes immediately. The exact behavior of simultaneous file mapping by multiple programs depends on the operating system.
• FileChannel.MapMode.PRIVATE: The resulting buffer is writable, but any changes are private to this buffer and not propagated to the file.

Once you have the buffer, you can read and write data using the methods of the ByteBuffer class and the Buffer superclass.

Buffers support both sequential and random data access. A buffer has a position that is advanced by get and put operations. For example, you can sequentially traverse all bytes in the buffer as

```java
while (buffer.hasRemaining())
{
    byte b = buffer.get();
    . . .
}
```

Alternatively, you can use random access:

```java
for (int i = 0; i < buffer.limit(); i++)
{
    byte b = buffer.get(i);
    . . .
}
```

You can also read and write arrays of bytes with the methods

```java
get(byte[] bytes)
get(byte[], int offset, int length)
```

Finally, there are methods

```java
getInt getChar
getLong getFloat
getShort getDouble
```

to read primitive-type values that are stored as binary values in the file. As we already mentioned, Java uses big-endian ordering for binary data. However, if you need to process a file containing binary numbers in little-endian order, simply call

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

To find out the current byte order of a buffer, call

```java
ByteOrder b = buffer.order();
```

⚠️ **CAUTION:** This pair of methods does not use the set/get naming convention.
To write numbers to a buffer, use one of the methods

```java
putInt            putChar
putLong           putFloat
putShort          putDouble
```

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

Listing 2.5 computes the 32-bit cyclic redundancy checksum (CRC32) of a file. That checksum is often used to determine whether a file has been corrupted. Corruption of a file makes it very likely that the checksum has changed. The `java.util.zip` package contains a class `CRC32` that computes the checksum of a sequence of bytes, using the following loop:

```java
var crc = new CRC32();
while (more bytes)
    crc.update(next byte);
long checksum = crc.getValue();
```

The details of the CRC computation are not important. We just use it as an example of a useful file operation. (In practice, you would read and update data in larger blocks, not a byte at a time. Then the speed differences are not as dramatic.)

Run the program as

```java
java memoryMap.MemoryMapTest filename
```

---

**Listing 2.5 memoryMap/MemoryMapTest.java**

```java
package memoryMap;

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
import java.util.zip.*;

/**
 * This program computes the CRC checksum of a file in four ways. <br>
 * Usage: java memoryMap.MemoryMapTest filename
 * @version 1.02 2018-05-01
 * @author Cay Horstmann
 */
```
public class MemoryMapTest
{
    public static long checksumInputStream(Path filename) throws IOException
    {
        try (InputStream in = Files.newInputStream(filename))
        {
            var crc = new CRC32();
            int c;
            while ((c = in.read()) != -1)
                crc.update(c);
            return crc.getValue();
        }
    }

    public static long checksumBufferedInputStream(Path filename) throws IOException
    {
        try (var in = new BufferedInputStream(Files.newInputStream(filename)))
        {
            var crc = new CRC32();
            int c;
            while ((c = in.read()) != -1)
                crc.update(c);
            return crc.getValue();
        }
    }

    public static long checksumRandomAccessFile(Path filename) throws IOException
    {
        try (var file = new RandomAccessFile(filename.toFile(), "r"))
        {
            long length = file.length();
            var crc = new CRC32();
            for (long p = 0; p < length; p++)
            {
                file.seek(p);
                int c = file.readByte();
                crc.update(c);
            }
            return crc.getValue();
        }
    }
}
public static long checksumMappedFile(Path filename) throws IOException
{
    try (FileChannel channel = FileChannel.open(filename))
    {
        var crc = new CRC32();
        int length = (int) channel.size();
        MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, length);
        for (int p = 0; p < length; p++)
        {
            int c = buffer.get(p);
            crc.update(c);
        }
        return crc.getValue();
    }
}

public static void main(String[] args) throws IOException
{
    System.out.println("Input Stream:");
    long start = System.currentTimeMillis();
    Path filename = Paths.get(args[0]);
    long crcValue = checksumInputStream(filename);
    long end = System.currentTimeMillis();
    System.out.println(Long.toHexString(crcValue));
    System.out.println((end - start) + " milliseconds");

    System.out.println("Buffered Input Stream:");
    start = System.currentTimeMillis();
    crcValue = checksumBufferedInputStream(filename);
    end = System.currentTimeMillis();
    System.out.println(Long.toHexString(crcValue));
    System.out.println((end - start) + " milliseconds");

    System.out.println("Random Access File:");
    start = System.currentTimeMillis();
    crcValue = checksumRandomAccessFile(filename);
    end = System.currentTimeMillis();
    System.out.println(Long.toHexString(crcValue));
    System.out.println((end - start) + " milliseconds");

    System.out.println("Mapped File:");
    start = System.currentTimeMillis();
    crcValue = checksumMappedFile(filename);
    end = System.currentTimeMillis();
    System.out.println(Long.toHexString(crcValue));
    System.out.println((end - start) + " milliseconds");
}

2.5 Memory-Mapped Files
<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.io.FileInputStream</code> 1.0</td>
<td><code>FileChannel getChannel()</code> 1.4</td>
<td>returns a channel for accessing this input stream.</td>
</tr>
<tr>
<td><code>java.io.FileOutputStream</code> 1.0</td>
<td><code>FileChannel getChannel()</code> 1.4</td>
<td>returns a channel for accessing this output stream.</td>
</tr>
<tr>
<td><code>java.io.RandomAccessFile</code> 1.0</td>
<td><code>FileChannel getChannel()</code> 1.4</td>
<td>returns a channel for accessing this file.</td>
</tr>
<tr>
<td><code>java.nio.channels.FileChannel</code> 1.4</td>
<td><code>static FileChannel open(Path path, OpenOption... options)</code> 7</td>
<td>opens a file channel for the given path. By default, the channel is opened for reading. The parameter options is one of the values WRITE, APPEND, TRUNCATE_EXISTING, CREATE in the StandardOpenOption enumeration.</td>
</tr>
<tr>
<td></td>
<td><code>MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)</code></td>
<td>maps a region of the file to memory. The parameter mode is one of the constants READ_ONLY, READ_WRITE, or PRIVATE in the FileChannel.MapMode class.</td>
</tr>
<tr>
<td><code>java.nio.Buffer</code> 1.4</td>
<td><code>boolean hasRemaining()</code></td>
<td>returns true if the current buffer position has not yet reached the buffer's limit position.</td>
</tr>
<tr>
<td></td>
<td><code>int limit()</code></td>
<td>returns the limit position of the buffer—that is, the first position at which no more values are available.</td>
</tr>
</tbody>
</table>
### java.nio.ByteBuffer 1.4

- **byte get()**
  - gets a byte from the current position and advances the current position to the next byte.
- **byte get(int index)**
  - gets a byte from the specified index.
- **ByteBuffer put(byte b)**
  - puts a byte at the current position and advances the current position to the next byte. Returns a reference to this buffer.
- **ByteBuffer put(int index, byte b)**
  - puts a byte at the specified index. Returns a reference to this buffer.
- **ByteBuffer get(byte[] destination)**
- **ByteBuffer get(byte[] destination, int offset, int length)**
  - fills a byte array, or a region of a byte array, with bytes from the buffer, and advances the current position by the number of bytes read. If not enough bytes remain in the buffer, then no bytes are read, and a BufferUnderflowException is thrown. Returns a reference to this buffer.
- **ByteBuffer put(byte[] source)**
- **ByteBuffer put(byte[] source, int offset, int length)**
  - puts all bytes from a byte array, or the bytes from a region of a byte array, into the buffer, and advances the current position by the number of bytes read. If not enough bytes remain in the buffer, then no bytes are written, and a BufferOverflowException is thrown. Returns a reference to this buffer.
- **Xxx getXxx()**
- **Xxx getXxx(int index)**
- **ByteBuffer putXxx(Xxx value)**
- **ByteBuffer putXxx(int index, Xxx value)**
  - gets or puts a binary number. *Xxx* is one of Int, Long, Short, Char, Float, or Double.
- **ByteBuffer order(ByteOrder order)**
- **ByteOrder order()**
  - sets or gets the byte order. The value for order is one of the constants BIG_ENDIAN or LITTLE_ENDIAN of the ByteOrder class.
- **static ByteBuffer allocate(int capacity)**
  - constructs a buffer with the given capacity.

(Continues)
### java.nio.ByteBuffer 1.4 (Continued)

- static ByteBuffer wrap(byte[] values)
  
  constructs a buffer that is backed by the given array.

- CharBuffer asCharBuffer()
  
  constructs a character buffer that is backed by this buffer. Changes to the character buffer will show up in this buffer, but the character buffer has its own position, limit, and mark.

### java.nio.CharBuffer 1.4

- char get()
- CharBuffer get(char[] destination)
- CharBuffer get(char[] destination, int offset, int length)

  gets one char value, or a range of char values, starting at the buffer’s position and moving the position past the characters that were read. The last two methods return this.

- CharBuffer put(char c)
- CharBuffer put(char[] source)
- CharBuffer put(char[] source, int offset, int length)
- CharBuffer put(String source)
- CharBuffer put(CharBuffer source)

  puts one char value, or a range of char values, starting at the buffer’s position and advancing the position past the characters that were written. When reading from a CharBuffer, all remaining characters are read. All methods return this.

### 2.5.2 The Buffer Data Structure

When you use memory mapping, you make a single buffer that spans the entire file or the area of the file that you’re interested in. You can also use buffers to read and write more modest chunks of information.

In this section, we briefly describe the basic operations on Buffer objects. A buffer is an array of values of the same type. The Buffer class is an abstract class with concrete subclasses ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, and ShortBuffer.

**NOTE:** The StringBuffer class is not related to these buffers.
In practice, you will most commonly use ByteBuffer and CharBuffer. As shown in Figure 2.9, a buffer has

- A capacity that never changes
- A position at which the next value is read or written
- A limit beyond which reading and writing is meaningless
- Optionally, a mark for repeating a read or write operation

![Figure 2.9 A buffer](image)

These values fulfill the condition

\[ 0 = \text{mark} = \text{position} = \text{limit} = \text{capacity} \]

The principal purpose of a buffer is a “write, then read” cycle. At the outset, the buffer’s position is 0 and the limit is the capacity. Keep calling put to add values to the buffer. When you run out of data or reach the capacity, it is time to switch to reading.

Call flip to set the limit to the current position and the position to 0. Now keep calling get while the remaining method (which returns limit – position) is positive. When you have read all values in the buffer, call clear to prepare the buffer for the next writing cycle. The clear method resets the position to 0 and the limit to the capacity.

If you want to reread the buffer, use rewind or mark/reset (see the API notes for details).

To get a buffer, call a static method such as ByteBuffer.allocate or ByteBuffer.wrap. Then, you can fill a buffer from a channel, or write its contents to a channel. For example,
ByteBuffer buffer = ByteBuffer.allocate(RECORD_SIZE);
channel.read(buffer);
channel.position(newpos);
buffer.flip();
channel.write(buffer);

This can be a useful alternative to a random-access file.

<table>
<thead>
<tr>
<th>java.nio.Buffer 1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Buffer clear()</td>
</tr>
<tr>
<td>prepares this buffer for writing by setting the position to 0 and the limit to the capacity; returns this.</td>
</tr>
<tr>
<td>• Buffer flip()</td>
</tr>
<tr>
<td>prepares this buffer for reading after writing, by setting the limit to the position and the position to 0; returns this.</td>
</tr>
<tr>
<td>• Buffer rewind()</td>
</tr>
<tr>
<td>prepares this buffer for rereading the same values by setting the position to 0 and leaving the limit unchanged; returns this.</td>
</tr>
<tr>
<td>• Buffer mark()</td>
</tr>
<tr>
<td>sets the mark of this buffer to the position; returns this.</td>
</tr>
<tr>
<td>• Buffer reset()</td>
</tr>
<tr>
<td>sets the position of this buffer to the mark, thus allowing the marked portion to be read or written again; returns this.</td>
</tr>
<tr>
<td>• int remaining()</td>
</tr>
<tr>
<td>returns the remaining number of readable or writable values—that is, the difference between the limit and position.</td>
</tr>
<tr>
<td>• int position()</td>
</tr>
<tr>
<td>• void position(int newValue)</td>
</tr>
<tr>
<td>gets and sets the position of this buffer.</td>
</tr>
<tr>
<td>• int capacity()</td>
</tr>
<tr>
<td>returns the capacity of this buffer.</td>
</tr>
</tbody>
</table>

### 2.6 File Locking

When multiple simultaneously executing programs need to modify the same file, they need to communicate in some way, or the file can easily become damaged. File locks can solve this problem. A file lock controls access to a file or a range of bytes within a file.
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.

```java
FileChannel = FileChannel.open(path);
FileLock lock = channel.lock();
```

or

```java
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 channel is closed or the `release` method is invoked on the lock.

You can also lock a portion of the file with the call

```java
FileLock lock(long start, long size, boolean shared)
```

or

```java
FileLock tryLock(long start, long size, boolean shared)
```

The shared flag is false to lock the file for both reading and writing. It is true for a shared lock, which allows multiple processes to read from the file, while preventing any process from acquiring an exclusive lock. Not all operating systems support shared locks. You may get an exclusive lock even if you just asked for a shared one. Call the `isShared` method of the `FileLock` class to find out which kind you have.

**NOTE:** If you lock the tail portion of a file and the file subsequently grows beyond the locked portion, the additional area is not locked. To lock all bytes, use a size of `Long.MAX_VALUE`.

Be sure to unlock the lock when you are done. As always, this is best done with a `try-with-resources` statement:

```java
try (FileLock lock = channel.lock())
{
    access the locked file or segment
}
```
Keep in mind that file locking is system-dependent. Here are some points to watch for:

- On some systems, file locking is merely *advisory*. If an application fails to get a lock, it may still write to a file that another application has currently locked.
- On some systems, you cannot simultaneously lock a file and map it into memory.
- File locks are held by the entire Java virtual machine. If two programs are launched by the same virtual machine (such as an applet or application launcher), they can’t each acquire a lock on the same file. The `lock` and `tryLock` methods will throw an `OverlappingFileLockException` if the virtual machine already holds another overlapping lock on the same file.
- On some systems, closing a channel releases all locks on the underlying file held by the Java virtual machine. You should therefore avoid multiple channels on the same locked file.
- Locking files on a networked file system is highly system-dependent and should probably be avoided.

### java.nio.channels.FileChannel 1.4

- **FileLock lock()**
  
  acquires an exclusive lock on the entire file. This method blocks until the lock is acquired.

- **FileLock tryLock()**

  acquires an exclusive lock on the entire file, or returns `null` if the lock cannot be acquired.

- **FileLock lock(long position, long size, boolean shared)**

  - **FileLock tryLock(long position, long size, boolean shared)**

  acquires a lock on a region of the file. The first method blocks until the lock is acquired, and the second method returns `null` if the lock cannot be acquired. The parameter `shared` is true for a shared lock, false for an exclusive lock.

### java.nio.channels.FileLock 1.4

- **void close() 1.7**

  releases this lock.
2.7 Regular Expressions

Regular expressions are used to specify string patterns. You can use regular expressions whenever you need to locate strings that match a particular pattern. For example, one of our sample programs locates all hyperlinks in an HTML file by looking for strings of the pattern `<a href="...">`.

Of course, when specifying a pattern, the `...` notation is not precise enough. You need to specify exactly what sequence of characters is a legal match, using a special syntax to describe a pattern.

In the following sections, we cover the regular expression syntax used by the Java API and discuss how to put regular expressions to work.

2.7.1 The Regular Expression Syntax

Let us start with a simple example. The regular expression

```
[Jj]ava.+
```

matches any string of the following form:

- The first letter is a J or j.
- The next three letters are ava.
- The remainder of the string consists of one or more arbitrary characters.

For example, the string "javanese" matches this particular regular expression, but the string "Core Java" does not.

As you can see, you need to know a bit of syntax to understand the meaning of a regular expression. Fortunately, for most purposes, a few straightforward constructs are sufficient.

- A **character class** is a set of character alternatives, enclosed in brackets, such as `[Jj]`, `[0-9]`, `[A-Za-z]`, or `[^0-9]`. Here the `-` denotes a range (all characters whose Unicode values fall between the two bounds), and `^` denotes the complement (all characters except those specified).
- To include a `-` inside a character class, make it the first or last item. To include a `, make it the first item. To include a `^`, put it anywhere but the beginning. You only need to escape `[ and ]`
- There are many predefined character classes such as \d (digits) or \p{Sc} (Unicode currency symbol). See Tables 2.6 and 2.7.
Table 2.6 Regular Expression Syntax

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Characters</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>. </code></td>
<td>Any character except line terminators, or any character if the DOTALL flag is set</td>
<td></td>
</tr>
<tr>
<td>`{</td>
<td>( ) [ \ ^ $`</td>
<td>The character <code>c</code></td>
</tr>
<tr>
<td><code>\x{p}</code></td>
<td>The Unicode code point with hex code <code>p</code></td>
<td><code>\x{1D546}</code></td>
</tr>
<tr>
<td><code>\u{p}{h}h{h}, \xb{h}, \xo{o}, \xoo{oo}</code></td>
<td>The UTF-16 code unit with the given hex or octal value</td>
<td><code>\u{FEFF}</code></td>
</tr>
<tr>
<td><code>\a, \e, \f, \n, \r, \t</code></td>
<td>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>)</td>
<td><code>\n</code></td>
</tr>
<tr>
<td><code>\dc, where c is in </code>[A-Z] or one of @ [ \ ] <code>_</code>?`</td>
<td>The control character corresponding to the character <code>c</code></td>
<td><code>\dh</code> is a backspace (<code>\x{8}</code>)</td>
</tr>
<tr>
<td><code>\dc, where c is not in </code>[A-Za-z0-9-]`</td>
<td>The character <code>c</code></td>
<td><code>\</code></td>
</tr>
<tr>
<td><code>\Q . . \E</code></td>
<td>Everything between the start and the end of the quotation</td>
<td><code>\Q ( ... ) \E</code> matches the string <code>( ... )</code></td>
</tr>
<tr>
<td><strong>Character Classes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>[C_1C_2 . . .], where C_i are characters, ranges c-d, or character classes</code></td>
<td>Any of the characters represented by <code>C_1</code>, <code>C_2</code>, . . .</td>
<td><code>[0-9+-]</code></td>
</tr>
<tr>
<td><code>[^ . . ]</code></td>
<td>Complement of a character class</td>
<td><code>[^\d\s]</code></td>
</tr>
<tr>
<td><code>[ . . \&amp;&amp; . . ]</code></td>
<td>Intersection of character classes</td>
<td><code>\p{L}\&amp;\{[^A-Za-z-\s]\}</code></td>
</tr>
<tr>
<td><code>\p{ . . }, \P{ . . }</code></td>
<td>A predefined character class (see Table 2.7); its complement</td>
<td><code>\p{L}</code> matches a Unicode letter, and so does <code>\p{L}</code>—you can omit braces around a single letter</td>
</tr>
<tr>
<td>Expression</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>\d, \D</td>
<td>Digits ([0-9], or \p{Digit} when the UNICODE_CHARACTER_CLASS flag is set); the complement</td>
<td>\d+ is a sequence of digits</td>
</tr>
<tr>
<td>\w, \W</td>
<td>Word characters ([a-zA-Z0-9_], or Unicode word characters when the UNICODE_CHARACTER_CLASS flag is set); the complement</td>
<td></td>
</tr>
<tr>
<td>\s, \S</td>
<td>Spaces ([\n\r\t\f\x{B}], or \p{IsWhite_Space} when the UNICODE_CHARACTER_CLASS flag is set); the complement</td>
<td>\s*,\s* is a comma surrounded by optional white space</td>
</tr>
<tr>
<td>\b, \v, \H, \V</td>
<td>Horizontal whitespace, vertical whitespace, their complements</td>
<td></td>
</tr>
</tbody>
</table>

**Sequences and Alternatives**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Any string from X, followed by any string from Y</th>
<th>[1-9][0-9]* is a positive number without leading zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>Any string from X or Y</td>
<td>http</td>
</tr>
</tbody>
</table>

**Grouping**

<table>
<thead>
<tr>
<th>(X)</th>
<th>Captures the match of X</th>
<th>'([^]*)' captures the quoted text</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>The n-th group</td>
<td>([&quot;]).*\1 matches 'Fred' or &quot;Fred&quot; but not &quot;Fred'</td>
</tr>
<tr>
<td>(?&lt;name&gt;X)</td>
<td>Captures the match of X with the given name</td>
<td>'(?&lt;id&gt;[A-Za-z0-9]+)' captures the match with name id</td>
</tr>
<tr>
<td>\k&lt;name&gt;</td>
<td>The group with the given name</td>
<td>\k&lt;id&gt; matches the group with name id</td>
</tr>
<tr>
<td>(?:X)</td>
<td>Use parentheses without capturing X</td>
<td>In (?:http</td>
</tr>
</tbody>
</table>

*(Continues)*
Table 2.6  (Continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| `(?>f_1 \ldots.:X),`  
`(?f_2 \ldots:X),`  
`(?f_1 \ldots.-X),` with  
`f_i` in `[dim\_su\_x]` | Matches, but does not capture, X with the given flags on or off (after `-`) | `(?!:{jpe?g})` is a case-insensitive match |
| Other `(\ldots)` | See the Pattern API documentation | |

**Quantifiers**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X?$</td>
<td>Optional $X$</td>
<td><code>\+?</code> is an optional + sign</td>
</tr>
<tr>
<td>$X^\ast$, $X+$</td>
<td>$0$ or more $X$, $1$ or more $X$</td>
<td><code>[1-9][0-9]+</code> is an integer $\geq 10$</td>
</tr>
<tr>
<td>$X{n}$, $X{n,}$, $X{m,n}$</td>
<td>$n$ times $X$, at least $n$ times $X$, between $m$ and $n$ times $X$</td>
<td><code>[0-7]{1,3}</code> are one to three octal digits</td>
</tr>
<tr>
<td>$Q?$, where $Q$ is a quantified expression</td>
<td>Reluctant quantifier, attempting the shortest match before trying longer matches</td>
<td><code>.*(&lt;.+?&gt;).*</code> captures the shortest sequence enclosed in angle brackets</td>
</tr>
<tr>
<td>$Q+$, where $Q$ is a quantified expression</td>
<td>Possessive quantifier, taking the longest match without backtracking</td>
<td><code>'^[^']*'</code> matches strings enclosed in single quotes and fails quickly on strings without a closing quote</td>
</tr>
</tbody>
</table>

**Boundary Matches**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>^</code>, <code>$</code></td>
<td>Beginning, end of input (or beginning, end of line in multiline mode)</td>
<td><code>^Java$</code> matches the input or line Java</td>
</tr>
<tr>
<td><code>\A</code>, <code>\Z</code>, <code>\</code></td>
<td>Beginning of input, end of input, absolute end of input (unchanged in multiline mode)</td>
<td></td>
</tr>
<tr>
<td><code>\b</code>, <code>\B</code></td>
<td>Word boundary, nonword boundary</td>
<td><code>\bJava\b</code> matches the word Java</td>
</tr>
<tr>
<td><code>\R</code></td>
<td>A Unicode line break</td>
<td></td>
</tr>
<tr>
<td><code>\G</code></td>
<td>The end of the previous match</td>
<td></td>
</tr>
</tbody>
</table>
Table 2.7 Predefined Character Class Names Used with \p

<table>
<thead>
<tr>
<th>Character Class Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>posixClass</td>
<td>posixClass 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 UNICODE_CHARACTER_CLASS flag</td>
</tr>
<tr>
<td>IsScript, sc=Script, script=Script</td>
<td>A script accepted by Character.UnicodeScript.forName</td>
</tr>
<tr>
<td>InBlock, blk=Block, block=Block</td>
<td>A block accepted by Character.UnicodeBlock.forName</td>
</tr>
<tr>
<td>Category, InCategory, gc=Category, general_category=Category</td>
<td>A one- or two-letter name for a Unicode general category</td>
</tr>
<tr>
<td>IsProperty</td>
<td>Property is one of Alphabetic, Ideographic, Letter, Lowercase, Uppercase, Titlecase, Punctuation, Control, White_Space, Digit, Hex_Digit, Join_Control, Noncharacter_Code_Point, Assigned</td>
</tr>
<tr>
<td>javaMethod</td>
<td>Invokes the method Character.isMethod (must not be deprecated)</td>
</tr>
</tbody>
</table>

- Most characters match themselves, such as the `ava` characters in the preceding example.
- The `. ` symbol matches any character (except possibly line terminators, depending on flag settings).
- Use `\` as an escape character. For example, `\.` matches a period and `\\` matches a backslash.
- `^` and `$` match the beginning and end of a line, respectively.
- If X and Y are regular expressions, then XY means “any match for X followed by a match for Y.” X | Y means “any match for X or Y.”
- You can apply quantifiers X+ (1 or more), X* (0 or more), and X? (0 or 1) to an expression X.
- By default, a quantifier matches the largest possible repetition that makes the overall match succeed. You can modify that behavior with suffixes ? (reluctant, or stingy, match: match the smallest repetition count) and + (possessive, or greedy, match: match the largest count even if that makes the overall match fail).

For example, the string `cab` matches `[a-z]*ab` but not `[a-z]*+ab`. In the first case, the expression `[a-z]*` only matches the character c, so that the characters `ab` match the remainder of the pattern. But the greedy version
[a-z]*+ matches the characters cab, leaving the remainder of the pattern unmatched.

* You can use groups to define subexpressions. Enclose the groups in ( ), for example, ([+-]?)([0-9]+). You can then ask the pattern matcher to return the match of each group or to refer back to a group with \n where n is the group number, starting with \1.

For example, here is a somewhat complex but potentially useful regular expression that describes decimal or hexadecimal integers:

```
[+-]?[0-9]+|0[Xx][0-9A-Fa-f]+  
```

Unfortunately, the regular expression syntax is not completely standardized between various programs and libraries; there is a consensus on the basic constructs but many maddening differences in the details. The Java regular expression classes use a syntax that is similar to, but not quite the same as, the one used in the Perl language. Table 2.6 shows all constructs of the Java syntax. For more information on the regular expression syntax, consult the API documentation for the Pattern class or the book *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O’Reilly and Associates, 2006).

### 2.7.2 Matching a String

The simplest use for a regular expression is to test whether a particular string matches it. Here is how you program that test in Java. First, construct a Pattern object from a string containing the regular expression. Then, get a Matcher object from the pattern and call its matches method:

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

The input of the matcher is an object of any class that implements the CharSequence interface, such as a String, StringBuilder, or CharBuffer.

When compiling the pattern, you can set one or more flags, for example:

```
Pattern pattern = Pattern.compile(expression,
Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE);
```

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 taken 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.

If you want to match elements in a collection or stream, turn the pattern into a predicate:

```java
Stream<String> strings = ...;
Stream<String> result = strings.filter(pattern.asPredicate());
```

The result contains all strings that match the regular expression.

If the regular expression contains groups, the Matcher object can reveal the group boundaries. The methods

```java
int start(int groupIndex)
int end(int groupIndex)
```

yield the starting index and the past-the-end index of a particular group.

You can simply extract the matched string by calling

```java
String group(int groupIndex)
```

Group 0 is the entire input; the group index for the first actual group is 1. Call the groupCount method to get the total group count. For named groups, use the methods

```java
int start(String groupName)
int end(String groupName)
String group(String groupName)
```
Nested groups are ordered by the opening parentheses. For example, given the pattern

```
((1-9)|1[0-2]):([0-5][0-9]))am
```

and the input

```
11:59am
```

the matcher reports the following groups

<table>
<thead>
<tr>
<th>Group Index</th>
<th>Start</th>
<th>End</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>11:59am</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>5</td>
<td>11:59</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td>59</td>
</tr>
</tbody>
</table>

Listing 2.6 prompts for a pattern, then for strings to match. It prints out whether or not the input matches the pattern. If the input matches and the pattern contains groups, the program prints the group boundaries as parentheses, for example:

```
((11):(59))am
```

### Listing 2.6  regex/RegexTest.java

```java
1 package regex;
2
3 import java.util.*;
4 import java.util.regex.*;
5
6 /**
7  * This program tests regular expression matching. Enter a pattern and strings to match,
8  * or hit Cancel to exit. If the pattern contains groups, the group boundaries are displayed
9  * in the match.
10  * @version 1.03 2018-05-01
11  * @author Cay Horstmann
12  */
13 public class RegexTest
14 {
15     public static void main(String[] args) throws PatternSyntaxException
16     {
17         var in = new Scanner(System.in);
18         System.out.println("Enter pattern: ");
19         String patternString = in.nextLine();
20         // Use the pattern to match the input
21     }
```

```
2.7.3 Finding Multiple Matches

Usually, you don’t want to match the entire input against a regular expression, but to find one or more matching substrings in the input. Use the find method of the Matcher class to find the next match. If it returns true, use the start and end methods to find the extent of the match or the group method without an argument to get the matched string.
while (matcher.find())
{
    int start = matcher.start();
    int end = matcher.end();
    String match = input.group();
    . . .
}

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(Matcher::group)
   .collect(Collectors.toList());

If you have the data in a file, 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, StandardCharsets.UTF_8);
Stream<String> words = in.findAll("\pL+")
   .map(MatchResult::group);

Listing 2.7 puts this mechanism to work. It locates all hypertext references in a web page and prints them. To run the program, supply a URL on the command line, such as

java match.HrefMatch http://horstmann.com

Listing 2.7 match/HrefMatch.java

package match;

import java.io.*;
import java.net.*;
import java.nio.charset.*;
import java.util.regex.*;

/**
 * This program displays all URLs in a web page by matching a regular expression that
 * describes the <a href=...> HTML tag. Start the program as <br>
 * java match.HrefMatch URL
 */
public class HrefMatch
{
    public static void main(String[] args)
    {
        try
        {
            // get URL string from command line or use default
            String urlString;
            if (args.length > 0) urlString = args[0];
            else urlString = "http://openjdk.java.net/";

            // read contents of URL
            InputStream in = new URL(urlString).openStream();
            var input = new String(in.readAllBytes(), StandardCharsets.UTF_8);

            // search for all occurrences of pattern
            var patternString = "<a\s+href\s*=\s*("[^\"]*"|[^\s>])*\s*>";
            Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
            pattern.matcher(input)
                .results()
                .map(MatchResult::group)
                .forEach(System.out::println);
        }
        catch (IOException | PatternSyntaxException e)
        {
            e.printStackTrace();
        }
    }
}
String[] tokens = input.split("\s*,\s*"_constraints);

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

```java
var in = new Scanner(path, StandardCharsets.UTF_8);
in.useDelimiter("\s*,\s*");
Stream<String> tokens = in.tokens();
```

### 2.7.5 Replacing Matches

The `replaceAll` method of the `Matcher` class replaces all occurrences of a regular expression with a replacement string. For example, the following instructions replace all sequences of digits with a `#` character:

```java
Pattern pattern = Pattern.compile("[0-9]+\_characters");
Matcher matcher = pattern.matcher(input);
String output = matcher.replaceAll("#\_characters");
```

The replacement string can contain references to the groups in the pattern: 
$n$ is replaced with the $n$th group, and `${name}` is replaced with the group that has the given name. Use `$\$` to include a `$` character in the replacement text.

If you have a string that may contain `\$` and `\`, and you don't want them to be interpreted as group replacements, call `matcher.replaceAll(Matcher.quoteReplacement(str))`.

If you want to carry out a more complex operation than splicing in group matches, 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:

```java
String result = Pattern.compile("\p{L}{4,}\_characters");
.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.
• Matcher matcher(CharSequence input)
  returns a matcher object that you can use to locate the matches of the pattern in the input.
• String[] split(CharSequence input)
• String[] split(CharSequence input, int limit)
• Stream<String> splitAsStream(CharSequence input)

  splits the input string into tokens, with the pattern specifying the form of the delimiters. Returns an array or stream of tokens. The delimiters are not part of the tokens. The second form has a parameter limit denoting the maximum number of strings to produce. If limit · 1 matching delimiters have been found, then the last entry of the returned array contains the remaining unsplit input. If limit is ≤ 0, then the entire input is split. If limit is 0, then trailing empty strings are not placed in the returned array.

• boolean matches()
  returns true if the input matches the pattern.
• boolean lookingAt()
  returns true if the beginning of the input matches the pattern.
• boolean find()
• boolean find(int start)
  attempts to find the next match and returns true if another match is found.
• int start()
• int end()
  returns the start or past-the-end position of the current match.
• String group()
  returns the current match.
• int groupCount()
  returns the number of groups in the input pattern.
java.util.regex.Matcher 1.4 (Continued)

- int start(int groupIndex)
- int start(String name) 8
- int end(int groupIndex)
- int end(String name) 8

returns the start or past-the-end position of a group in the current match. The group is specified by an index starting with 1, or 0 to indicate the entire match, or by a string identifying a named group.

- String group(int groupIndex)
- String group(String name) 7

returns the string matching a given group, denoted by an index starting with 1, or 0 to indicate the entire match, or by a string identifying a named group.

- String replaceAll(String replacement)
- String replaceFirst(String replacement)

returns a string obtained from the matcher input by replacing all matches, or the first match, with the replacement string. The replacement string can contain references to pattern groups as \$n. Use \$ to include a $ symbol.

- static String quoteReplacement(String str) 5.0
  quotes all \\ and $ in str.
- String replaceAll(Function<MatchResult,String> replacer) 9
  replaces every match with the result of the replacer function applied to the MatchResult.
- Stream<MatchResult> results() 9
  yields a stream of all match results.

java.util.regex.MatchResult 5

- String group()
- String group(int group)
  yields the matched string or the string matched by the given group.
- int start()
- int end()
- int start(int group)
- int end(int group)
  yields the start and end offsets of the matched string or the string matched by the given group.
You have now seen how to carry out input and output operations in Java, and had an overview of the regular expression package that was a part of the “new I/O” specification. In the next chapter, we turn to the processing of XML data.
## Numbers
- (minus sign)
  - in permissions, 558
  - in policy files, 557
  - in regular expressions, 137
  - in URLs, 263
- (underscore)
  - in native method names, 811
  - in SQL, 291, 320
  - in URLs, 263
, (comma)
  - decimal, 380, 387
  - in DTDs, 173
; (semicolon)
  - in classpath, 294
  - in method signatures, 831
  - in SQL, 295
  - not needed, in annotations, 464
: (colon)
  - as delimiter in text files, 72
  - in classpath, 294
  - in permissions, 558
  - in URLs, 252–253
!= operator (SQL), 291
? (question mark)
  - in DTDs, 173
  - in glob patterns, 120–141
  - in prepared queries, 311
  - in URLs, 263
/ (slash)
  - in method signatures, 832
  - in paths, 64, 109
  - in URLs, 252, 553
. (period)
  - decimal, 380, 387
  - in method signatures, 832
  - in regular expressions, 138, 141
  - in URLs, 263
  - leading, in file names, 552
..., in paths, 110
^ (caret), in regular expressions, 137, 140–141
~ (tilde), in URLs, 263
'. . .', in SQL, 291
'* . . .', in XML, 156
( . . )
  - in method signatures, 831
  - in regular expressions, 139, 142–144
[ (array), type code, 95, 831
[. . .]
  - in DTDs, 173, 179
  - in glob patterns, 120
  - in regular expressions, 137–138
  - in XPath, 189
{. . .}
  - in annotations, 474
  - in glob patterns, 120
  - in message formatting, 409–413
  - in regular expressions, 140
@ (at)
  - in annotations, 464
  - in URLs, 253
  - in XPath, 189
$ (dollar sign)
  - in native method names, 811
  - in regular expressions, 138, 140–141, 150
${. . .}, in policy files, 558
* (asterisk)
  - in DTDs, 173
  - in glob patterns, 120
  - in permissions, 558
  - in policy files, 557
  - in regular expressions, 140–141
\ (backslash)
  - in glob patterns, 120
  - in paths, 64, 109
  - in permissions (Windows), 558
  - in regular expressions, 137–138, 141, 150
\|, in regular expressions, 73
\& (ampersand)  
in CDATA sections, 159  
in entity references, 158  
parsing, 173  
\\\&amp; (ampersand)  
in CDATA sections, 159  
in entity references, 158  
parsing, 173  
\\\&\$ (number sign)  
in message formatting, 412–413  
in URLs, 252  
\\\&% (percent sign)  
in locales, 387  
in SQL, 291, 320  
in URLs, 263  
\\\&+ (plus sign)  
in DTDs, 173  
in regular expressions, 140–141  
in URLs, 263  
\\\&< (left angle bracket)  
in CDATA sections, 159  
in message formatting, 413  
parsing, 173  
\\\&! [CDATA[ ... ]]> in XML, 159  
\\\&. . . >, \&?! . . ?!, \&![CDATA[ . . . ]]> in XML, 159  
\\\&. . . >, in regular expressions, 139  
\\\&\textless (less than)  
in DTDs, 173  
in regular expressions, 140–141  
in URLs, 263  
\\\&\& (number sign)  
in message formatting, 412–413  
in URLs, 252  
\\\&\% (percent sign)  
in locales, 387  
in SQL, 291, 320  
in URLs, 263  
\\\&\textless (left angle bracket)  
in CDATA sections, 159  
in message formatting, 413  
parsing, 173  
\\\&\textless! [CDATA[ ... ]]> in XML, 159  
\\\&\textless. . . >, \&?! . . ?!, \&![CDATA[ . . . ]]> in XML, 159  
\\\&\textless. . . >, in regular expressions, 139  
\\\&\textless\textless (less than)  
in DTDs, 173–174  
in message formatting, 412  
in regular expressions, 139, 141  
\\\&\textless\textless \texttt{\textbackslash 0}, in regular expressions, 138  
\\\&\textless\texttt{\textbackslash 1931 CIE XYZ color specification}, 758  
\\\&\textless\texttt{2D graphics. See Java 2D API}  

A  
\\\&\texttt{\textbackslash a}, \&\texttt{\textbackslash A}, in regular expressions, 138, 140  
\\\&\texttt{\textbackslash abort method (LoginModule)}, 582  
\\\&\texttt{\textbackslash absolute method (ResultSet)}, 325, 328  
\\\&\texttt{\textbackslash AbstractCellEditor class}, 643–645  
\\\&\texttt{\textbackslash isCellEditable method}, 644  
\\\&\texttt{\textbackslash AbstractProcessor class}, 485  
\\\&\texttt{\textbackslash AbstractTableModel class}, 618  
\\\&\texttt{\textbackslash isCellEditable method}, 641  
\\\&\texttt{\textbackslash accept method (ServerSocket)}, 236, 239–240  
\\\&\texttt{\textbackslash acceptChanges method (CachedRowSet)}, 332–333  

AccessController class  
\\\&\texttt{\textbackslash getMethod method}, 572  
\\\&\texttt{\textbackslash Accumulator functions}, 42  
\\\&\texttt{\textbackslash Action listeners}  
\\\&\texttt{\textbackslash annotating}, 465–471  
\\\&\texttt{\textbackslash installing}, 466  
\\\&\texttt{\textbackslash action.properties file}, 457  
\\\&\texttt{\textbackslash ActionListener interface}  
\\\&\texttt{\textbackslash actionPerformed method}, 466  
\\\&\texttt{\textbackslash @ActionListenerFor annotation}, 465, 482  
\\\&\texttt{\textbackslash ActionListenerInstaller class}  
\\\&\texttt{\textbackslash processAnnotations method}, 466  

add method  
\\\&\texttt{\textbackslash of Area}, 714–715  
\\\&\texttt{\textbackslash of AttributeSet}, 802  
\\\&\texttt{\textbackslash of DefaultMutableTreeNode}, 655, 663  

addAttribute method (AttributesImpl), 226  

addBatch method (Statement), 346–347  

addCellEditorListener method (CellEditor), 652  

addColumn method (JTable), 630, 636  

Addition operator, identity for, 41  

addRecipient method (MimeMessage), 279  

addTreeModelListener method (TreeModel), 685, 693  

addTreeSelectionListener method (JTree), 677  

addURLs method (URLClassLoader), 532  

AES (Advanced Encryption Standard), 600  

generating keys in, 602–607  

aes/AESTest.java, 603  

aes/Util.java, 605  

Affine transformations, 730, 764  

AffineTransform class, 730–732  

constructor, 731  

getXXXInstance methods, 730–732  

setToXXX methods, 730, 732  

AffineTransformOp class, 764  

constructor, 771  

TYPE XXX fields, 764, 771  

afterLast method (ResultSet), 325, 328  

Agent code, 496–497  

Aliases, for namespaces, 179, 195  

allMatch method (Stream), 16  

allocate method (ByteBuffer), 131, 133  

AllPermission class, 550, 556, 560  

Alpha channel, 735–739  

Alpha composites, 744
Arrays (continued)

of strings, 147
type code for, 95, 831
type use annotations in, 476

Arrays class

stream method, 5, 9, 43
ArrayStoreException, 845
asCharBuffer method (ByteBuffer), 132
ASCII standard, 75
and native code, 820
in property files, 419
in regular expressions, 141
ASM library, 489–498
ASP (Active Server Pages), 261
Associative operations, 41
ATTRIBUTE declaration (DTD), 174
attribute element (XML Schema), 181
Attribute interface, 799

getCategory method, 801, 806
getName method, 806
implementing, 801
Attribute sets, 801

Attributes (XML)

enumerating, 164
for enumerated types, 175
in XML Schema, 181
legal, 174
names of, 156
namespace of, 195
values of, 156
accessing in XPath, 189
copying with XSLT, 219
default (DTDs), 175
normalizing, 175
vs. elements, 157–158, 175, 216

Attributes interface

getXxx methods, 202
AttributeSet interface, 800

add, get methods, 802, 806
remove, toArray methods, 807

AttributesImpl class

addAttribute method, 226
clear method, 226
atZone method (LocalDateTime), 367
AudioPermission class, 556
auth/AuthTest.java, 570
auth/AuthTest.policy, 571
auth/jaas.config, 571
auth/SysPropAction.java, 570

Authentication, 566–582
problems of, 592–593
role-based, 573
separating from business logic, 568
through a trusted intermediary, 593–594
AuthPermission class, 557
Autoboxing, 615
AutoCloseable interface
close method, 61
Autocommit mode (databases), 344–346
Autoflushing, 69
Autogenerated keys, 322
Auxiliary files, generated, 463
available method (InputStream), 57–58
average method (primitive streams), 44, 46–47
AWT (Abstract Window Toolkit), 693–744
AWTPermission class, 555

B

B (byte), type code, 95, 831
\b, \B, in regular expressions, 140
Bandaging, 775
Banner class

gapPageCount, layoutPages methods, 783
Banners, printing, 782–792
BASE64Encoder class, 507
BaseStream interface

iterator method, 28
parallel method, 48, 53
unordered method, 53
BasicFileAttributes interface, 117
methods of, 118
BasicPermission class, 557
BasicStroke class, 715–724
Batch updates (databases), 345–348
BCP 47 memo, 382
Bean info classes, generated, 463
beforeFirst method (ResultSet), 325, 328
between method (Duration), 355, 357
Bevel join, 716–717
Bézier curves, 703
Bicubic, bilinear interpolations, 764, 771
BIG_ENDIAN constant (ByteOrder), 131
Big-endian order, 78, 416
Binary data

converting to Unicode code units, 68
reading, 79
vs. text, 68
writing, 78
Bindings interface, 442
get, put methods, 443
Birthdays, calculating, 359
BitSet interface, 42
BLOB data type (SQL), 293, 348
Blob interface, 317
get/setBinaryStream methods, 319
getBytes method, 317–318
length method, 318
BLOBs (binary large objects), 317
creating empty, 319
placing in database, 318
Blocking
by I/O methods, 57
by network connections, 228, 232, 244–251
Blur filter, 765
BMP format, 745
body method (HttpResponse), 272, 278
BodyHandlers class
discarding method, 272
ofString method, 272–273
Book class, 782
book/Banner.java, 786
book/BookTestFrame.java, 785
book/PrintPreviewCanvas.java, 791
book/PrintPreviewDialog.java, 789
BOOLEAN data type (SQL), 293, 348
boolean type
printing, 69
streams of, 43
type code for, 95, 831
vs. C types, 817
writing in binary format, 78
Bootstrap class loader, 531
Bounding rectangle, 697
boxed method (primitive streams), 44, 46–47
Bray, Tim, 155
Breadth-first enumerations, 672
breadthFirstEnumeration method
(DefaultMutableTreeNode), 672, 676
Buffers
forms in, 261–270
response page in, 262
Buffer class, 132–134
capacity, position methods, 134
clear, flip, mark, remaining, reset, rewind methods, 133–134
hasRemaining, limit methods, 130
BufferedImage class, 726, 756
constructor, 756, 761
colorModel method, 758, 762
colorModel method, 756, 762
TYPE_BYTE_GRAY field, 760–761
TYPE_BYTE_INDEXED field, 761
TYPE_INT_ARGB field, 756–757, 761
BufferedImageOp interface, 756
filter method, 763, 771
implementing, 763
BufferedInputStream class, 67
BufferedOutputStream class, 67
BufferedReader class
readLine method, 71
Buffers, 132–134
capacity of, 133
flushing, 57, 69
in-memory, 62
limits of, 133
marks in, 133
positions in, 126, 133
traversing all bytes in, 126
vs. random access, 125
BufferUnderflowException, 131
@BugReport annotation, 482
build method
of HttpClient.Builder, 271–272, 277
of HttpRequest.Builder, 277
Business logic, 286
Butt cap, 716
ButtonFrame class, 447
addActionListener method, 465–466
buttons1/ButtonFrame.java, 452
buttons2/action.properties, 461
buttons2/ButtonFrame.java, 460
buttons3/ButtonFrame.java, 469
Byte codes, writing to memory, 455–456
Byte order mark, 416
byte type
streams of, 43
type code for, 95, 831
vs. C types, 817
BYTE_ARRAY class (DocFlavor), 794
ByteArrayClass class, 455
ByteArrayClassLoader class, 456
ByteArrayOutputStream class, 106
ByteBuffer class, 125, 132–134
allocate method, 131, 133
ByteBuffer class (continued)
   asCharBuffer method, 132
   get, getXxx methods, 126, 131
   order method, 126, 131
   put, putXxx methods, 127, 131
   wrap method, 132–133
bytecodeAnnotations/EntryLogger.java, 492
bytecodeAnnotations/EntryLoggingAgent.java, 497
Bytecodes
   engineering, 489–498
   at load time, 495–497
   with hex editor, 544–545
   verifying, 541–545
ByteOrder class
   BIG_ENDIAN, LITTLE_ENDIAN constants, 131
Byte-oriented input/output streams, 56
Bytes, reading/writing, 56–59

C (char), type code, 95, 831
C programming language
   array types in, 840–844
   bootstrap class loader in, 531
   calling Java methods from, 833–840
database access in, 284
   embedding JVM into, 849–854
   FILE* type in, 59
   pointers in, 810
   strings in, 819
types, vs. Java types, 817
\c, in regular expressions, 138
C++ programming language
   accessing JNI functions in, 820
   array types in, 840
   embedding JVM into, 849–854
   exceptions in, 845
   for native methods, 810, 813
   pointers in, 810, 835
Cached row sets, 330–335
CachedRowSet interface, 330–333
   acceptChanges method, 332–333
   execute, nextPage, populate methods, 331, 333
   get/setPageSize method, 331, 333
   get/setTableName method, 332–333
   previousPage method, 333
CachedRowSetImpl class, 507
Caesar cipher, 536–537
Calendar class, 353
   formatting objects of, 395
   weekends in, 360
call escape (SQL), 320
call method (CompilationTask), 454, 462
Call stack, during permission checking, 550
Callable interface, 454
Callback interface, 574
CallbackHandler interface
   handle method, 581
CallNonvirtualXxxMethod functions (C), 836, 839
CallStaticXxxMethod functions (C), 834–835, 839
CallXxxMethod functions (C), 833, 838–839
cancelCellEditing method (CellEditor), 644–645, 652
cancelRowUpdates method (ResultSet), 326, 329
canInsertImage method (ImageWriter), 748, 755
capacity method (Buffer), 134
Carriage return character, displaying, 164
Casts, type use annotations in, 476
catalog element (XML), 172
CatalogFeatures class
   defaults method, 172, 179
CatalogManager class
   catalogResolver method, 172, 178
Catalogs, 343
CDATA declaration (DTD), 174–175
CDATA sections (XML), 159
Cell editors (Swing), 641–642
custom, 642–652
Cell renderers (Swing), 639–652
   for tables, 622, 639
   for trees, 674–676
CellEditor interface
   add/removeCellEditorListener methods, 652
   cancelCellEditing method, 644–645, 652
ggetCellEditorValue method, 641, 644–645, 652
isCellEditable method, 652
shouldSelectCell method, 644, 652
stopCellEditing method, 644–645, 652
Cells (Swing)
editing, 641–642
selecting, 626
Certiﬁcates, 566, 589–592
and Java Plug-in, 598
managing, 596–597
publishing ﬁngerprints of, 591
root, 599
set of, 547
signing, 594–597
CertificateSigner class, 595
CGI (Common Gateway Interface), 261
Chain of trust, 593
Channels, 245
for ﬁles, 125
Channels class
newInputStream method, 251
newOutputStream method, 245, 251
char type
streams of, 43
type code for, 95, 831
vs. C types, 817
CHAR ARRAY class (DocFlavor), 794
Character classes, 137
CHARACTER data type (SQL), 293, 348
Character encodings, 68, 75–77
explicitly speciﬁed, 77
partial, 76
platform, 77, 413
Character references (XML), 158
CharacterData interface
data method, 163, 169
Characters
differences between, 402
escaping, 73, 141
in regular expressions, 141
normalizing, 403
outlines of, 733
printing, 69
writing in binary format, 78
characters method (ContentHandler), 197, 201
CharBuffer class, 62, 132
get, put methods, 132
CharSequence interface, 62, 142
charAt, length methods, 63
chars method, 43
codePoints method, 43, 47
splitting by regular expressions, 6
subSequence, toString methods, 63
Charset class
defaultCharset method, 77, 415
forName method, 77
Checkboxes (Swing), 639
checked attribute (HTML, XML), 156
Checked exceptions, 568
Checker framework, 476
checkError method (PrintWriter), 69–70
checkExit method (SecurityManager), 546, 549
checkLogin method (SimpleLoginModule), 573
checkPermission method (SecurityManager), 549–550, 559–560
checkRead method (SecurityManager), 550
Child elements (XML), 157
namespace of, 194
Child nodes (Swing), 652
adding, 655
connecting lines for, 658–659
children method (TreeNode), 671
choice element (XML Schema), 181
choice keyword (message formatting), 412
Church, Alonzo, 358
Cipher class, 600–601
doFinal method, 601, 603, 606–607
getInstance method, 600, 606
getXXXSize methods, 606
init method, 606
update method, 601, 603, 606–607
XXX_MODE modes, 600
CipherInputStream class
read method, 608
CipherOutputStream class, 607
constructor, 608
flush, write methods, 608
Ciphers
generating keys, 602–607
public keys in, 608–612
streams for, 607–608
symmetric, 600–601
Circular dependencies, in annotations, 475
Class class
forName method, 456
classLoader method, 531, 540
getFields method, 687
getProtectionDomain method, 551
classFile method, 514
implementing AnnotatedElement, 466
file extension, 530
Class files, 530
corrupted, 542–545
encrypted, 535, 537
format of, 489
Class files (continued)
loading, 530
modifying, 490–495
portability of, 416
transformers for, 496
verifying, 541–545
Class loaders, 457, 530–545
as namespaces, 534
bootstrap, 531
context, 532–534
creating, 546
extension, 531
hierarchy of, 532–534
platform, 531
separate for each web page, 534
specifying, 532
system, 531
writing, 534–541
Class path, adding JAR files to, 532
Class references, in native code, 827
Classes
adding validation to, 100
annotating, 464, 475, 479
compiling on the fly, 455
descriptions of, 94
encapsulation of, 500
externalizable, 95
inheritance trees of, 673
nonserializable, 100
platform, overriding, 531
protection domains of, 549
resolving, 530
separate for each web page, 534
serializable, 88–89
versioning, 103–106
Classifier functions, 35
ClassLoader class, 531
defineClass, findClass methods, 534, 541
extending, 456, 534
getParent method, 540
getSystemClassLoader method, 541
loadClass method, 534
ClassLoader inversion, 532
classLoader/Caesar.java, 540
classLoader/ClassLoaderTest.java, 537
CLASSPATH environment variable, 531
CLEAR composition rule, 737
clear method
of AttributesImpl, 226
of Buffer, 133–134
clearParameters method (PreparedStatement), 317
client/HttpClientTest.java, 273
Client/server applications, 287
Clients
connecting to servers, 230–232
multiple, serving, 239–243
clip method (Graphics2D), 694, 733–735, 775
Clipboard, accessing, 546
Clipping region
printing, 775
setting, 695
Clipping shapes, 694, 733–735
CLOB data type (SQL), 293, 348
Clob interface, 317
c getColumnarStream method, 318–319
c getSubString method, 318–319
length method, 319
setCharacterStream method, 319
CLOBs (character large objects), 317
creating empty, 319
placing in database, 318
close method, 89, 106
Cloning, 106–108
close method
of AutoCloseable, 61
of Connection, 301, 303, 351
of fileLock, 136
of Flushable, 59
of InputStream, 57–58
of OutputStream, 59
of ResultSet, 303
of ServerSocket, 239
of Statement, 302–303
of XMLStreamWriter, 215
Closeable interface, 59
close method, 59, 62
flush method, 60
Closed nonleaf icons, 661–662, 674
closeEntry method (ZipXxxStream), 85–87
closeOnCompletion method (Statement), 302
closePath method (Path2D), 703, 714
Closure types, 700
cmd shell, 415
Code base, 547, 553
Code generation, annotations for, 463–471, 480
Code points, 11
Code source, 547
Code units, 43
  in regular expressions, 138
Codebreakers, The (Kahn), 536
codePoints method (CharSequence), 43, 47
CodeSource class
  getXxx methods, 551
Collation, 402–409
collation/CollationTest.java, 405
CollationKey class
  compareTo method, 409
Collator class, 402
  compare, equals methods, 408
  get/setDecomposition methods, 408
  get/setStrength methods, 408
  getAvailableLocales method, 408
  getCollationKey method, 404, 408
  getInstance method, 408
collect method (Stream), 25–30, 42–43
collecting/CollectingIntoMaps.java, 32
collecting/CollectingResults.java, 26
collecting/DownstreamCollectors.java, 38
collectingAndThen method (Collectors), 36, 40
Collection interface
  parallelStream method, 2–3, 5, 48
  stream method, 2–3, 5
Collections
  iterating over elements of, 2–5
  vs. streams, 3
Collector class, 402
Collator interface, 25
Collectors, 25–40
  composing, 37
downstream, 36–40, 50
Collectors class
  collectingAndThen method, 36, 40
  counting method, 36, 40
  filtering method, 37, 40
  flatMapping method, 37, 40
  groupingBy method, 34–40
  groupingByConcurrent method, 35, 50
  joining method, 26, 29
  mapping method, 37, 40
  maxBy, minBy methods, 36, 40
  partitioningBy method, 35, 37
  reducing method, 37
  summarizingXxx methods, 26, 29, 37
  summingXxx methods, 36, 40
toCollection, toList methods, 25, 29
toConcurrentMap method, 31, 34
toMap method, 30–34
toSet method, 25, 29, 36
toUnmodifiableList method, 29
toUnmodifiableMap method, 34
toUnmodifiableSet method, 29
Color chooser, 642
Color class, 724
  constructor, 763
  getRGB method, 763
translating values into pixel data, 760
Color space conversions, 765
ColorConvertOp class, 764–765
ColorModel class, 760
  getDataElements method, 763
  getRGB method, 758, 763
Colors
  components of, 735
  composing, 736–739
  interpolating, 725
  negating, 765
  solid, 694
Columns (databases)
  accessing by number, in result set, 300
  names of, 287
  number of, 335
Columns (Swing)
  accessing, 623
  adding, 630
  detached, 615
  hiding, 630–639
  names of, 619
  rendering, 622
  resizing, 615–616, 624–625
  selecting, 626
com.sun.security.auth.module package, 568
Combo box editors, 642
Comments (XML), 159
commit method
  of Connection, 345–347
  of LoginModule, 582
commonPool method (ForkJoinPool), 51
Comparator method (ForkJoinPool), 51
Comparators interface, 14, 628
Comparators interface, 14, 402
Comparators, 628
collator method (Collator), 408
collatorTo method
  of CollationKey, 409
  of Comparable, 628
compareTo method (continued)
of String, 402
Compileable interface
compile method, 447
CompilationTask interface, 453
call method, 454, 462
compile method (Pattern), 142, 148
CompiledScript interface
eval method, 447
Compiler
annotations for, 480
invoking, 453
just-in-time, 850
compiler/CompilerTest.java, 458
Complex types, 179
complexType element (XML Schema), 180
Composite interface, 738
composite/CompositeComponent.java, 741
composite/CompositeTestFrame.java, 740
composite/Rule.java, 743
Composition rules, 694–695, 735–744
Computer Graphics: Principles and Practice
(Foley et al.), 703, 738, 758
concat method (Stream), 13
Confidential information, transferring, 600
Configuration files, 135
call method
of Socket, 233
of URLConnection, 254, 256, 260
Connection interface
close method, 301, 303, 351
commit method, 345–347
createBlob, createClob methods, 318–319
createStatement method, 299, 301, 323, 327, 345
get/setAutoCommit methods, 347
getMetaData method, 334, 343
getWarnings method, 306
prepareStatement method, 311, 317, 323, 327
releaseSavepoint method, 345, 347
rollback method, 345–347
setSavepoint method, 347
Connections (databases)
closing, 303
using row sets after, 330
debugging, 280
pooling, 350
starting new threads, 240
console method (System), 415
Constructive area geometry operations, 714
Constructor class, 466
Constructors
annotating class, 475
invoking from native code, 835
type use annotations in, 476
containsAll method (Collection), 561
Content types, 254
ContentHandler class, 197–198
characters method, 197, 201
start/endDocument methods, 201
start/endElement methods, 197–201
Context class loader, 532–534
Control points
dragging, 704
of curves, 702–703
of shapes, 704
convertXxxIndexToModel methods (JTable), 626, 636
Convolution operation, 765
ConvolveOp class, 764–766
constructor, 772
CookieHandler class
setDefault method, 267
Cookies, 267
Coordinate system
custom, 695
translating, 776
Coordinate transformations, 727–732
Copies class, 799–802
getValue method, 802
CopiesSupported class, 800
copy method (Files), 114–115
CORBA (Common Object Request Broker Architecture), 500, 531
Core Swing (Topley), 614, 652, 666
count method (Stream), 3–4, 15, 189
counting method (Collectors), 36, 40
Country codes, 35, 382
CRC32 checksum, 87, 125, 127
CRC32 class, 127
CREATE TABLE statement (SQL), 292
eexecuting, 300–301, 317
in batch updates, 346
createBindings method (ScriptEngine), 443
createBlob, createClob methods (Connection), 318–319
createDirectory, createDirectories, createFile methods (Files), 113–114
createElement method (Document), 206–207, 209
createElementNS method (Document), 207, 209
createImageXxxStream methods (ImageIO), 747, 753
createPrintJob method (PrintService), 793, 795
createStatement method (Connection), 299, 301, 323, 327, 345
createTempXxx methods (Files), 113–114
createTextNode method (Document), 207, 209
createXMLStreamReader method (XMLInputFactory), 205
createXMLStreamWriter method (XMLOutputFactory), 210, 214
createXxxRowSet methods (RowSetFactory), 330, 333
creationTime method (BasicFileAttributes), 118
Credit card numbers, transferring, 600
crypt program, 603
Cryptography and Network Security (Stallings), 584, 94
Cubic curves, 702–703
CubicCurve2D class, 697, 699, 703
CubicCurve2D.Double class, 697–698, 713
CubicCurve2D.Float class, 697
Currencies, 393–394
available, 394
formatting, 387–394
identifiers for, 393
Currency class, 393–394
getAvailableCurrencies, getCurrencyCode,
getDefaultValueFractions, getNumericXxx,
getSymbol methods, 394
getInstance method, 393–394
toString method, 394
curveTo method (Path2D.Float), 703, 713
Custom editors, 642–652
Cygwin, 814
   compiling invocation API, 854
   OpenSSL in, 596
D
D (double), type code, 95, 831
d literal (SQL), 320
\d, \D in regular expressions, 139
Dashed lines, 717–718
Data
   fingerprints of, 583–587
   signed, 587–589
Data sources (for JNDI service), 350
Data types
codes for, 95, 831
in Java vs. C, 817
mangling names of, 831
print services for, 792–794
database.properties file, 307, 349
DatabaseMetaData interface, 334–344
getJDBCXxxVersion methods, 343
getMaxConnection method, 343
getMaxStatements method, 303, 343
getSQLStateType method, 304
getTables method, 334, 343
supportsBatchUpdates method, 346, 348
supportsResultSetXxx methods, 324, 329
Databases
accessing, in C language, 284
autocommit mode of, 344–346
autonumbering keys in, 322
batch updates for, 345–348
caching prepared statements, 312
changing data with SQL, 292
connections to, 294, 296–299, 307
closing, 303, 308
in web and enterprise applications, 349–351
pooling, 350
drivers for, 285–286
duplication of data in, 289
error handling in, 346
integrity of, 344
LOBs in, 317–319
metadata for, 334–344
modifying, 330
native storage for XML in, 349
numbering columns in, 300
outer joins in, 320
populating, 306–310
saving objects to, 483
scalar functions in, 320
schemas for, 343
setting up parameters in, 331
starting, 294–295
stored procedures in, 320
structure of, 287, 334
synchronization of, 332
tools for, 336
truncated data from, 305
Databases (continued)
URLs of, 294
DataFlavor class, 501
DataInput interface
readBoolean method, 79
readChar method, 79, 81
readDouble method, 79, 89, 101
readFloat method, 79
readFully method, 79
readInt method, 79, 81, 89
readLong method, 79
readShort method, 79
readUTF method, 79–80
skipBytes method, 80
DataInputStream class, 59, 64
DataIO class
xxxFixedString methods, 81–82
DataOutput interface, 78
writeBoolean method, 78, 80
writeChar method, 78, 80
writeChars method, 78, 80–81
writeDouble method, 78, 80, 89, 101
writeFloat method, 78, 80
writeInt method, 78, 80–81, 89
writeLong method, 78, 80
writeShort method, 78, 80
writeUTF method, 78, 80
DataOutputStream class, 59
DataSource interface, 350
DataTruncation class, 305
methods of, 306
Date and Time API, 353–377
legacy code and, 376–377
Date class (java.sql), 376
valueOf method, 377
Date class (java.util), 95, 353, 376
formatting objects of, 395
months and years in, 359
readObject, writeObject methods, 101
toInstant method, 376–377
DATE data type (SQL), 293, 320, 348
dateFilter method (RowFilter), 629, 638
DateFormat class, 373, 377, 396
dateFormat/DateTimeFormatterTest.java, 397
dateFormat/EnumCombo.java, 400
Dates
computing, 359, 364–365
filtering, 629
formatting, 371–376, 380, 394–401
literals for, 319
local, 358–363
parsing, 373
datesUntil method (LocalDate), 360, 363
DateTimeFormatter class, 371–376, 394–401
format method, 371, 375, 401
legacy classes and, 376–377
ofLocalizedXxx methods, 372, 375, 394, 401
ofPattern method, 373, 376
parse method, 373
toFormat method, 373, 377
withLocale method, 372, 376, 401
DateTimeParseException, 401
DateTimeSyntax class, 802
Daylight savings time, 367–371
DayOfWeek enumeration, 360
goingDisplay method, 373, 395
dayOfWeekInMonth method (TemporalAdjusters), 365
DBeaver program, 336
DDL statement (SQL), 301, 317
Debugging
in JNI, 850
JDBC-related problems, 297
locales, 386
mail connections, 280
streams, 14
DECIMAL data type (SQL), 293, 348
Decimal separators, 380, 387
Declaration annotations, 475–477
decryption method (URLDecoder), 270
Decryption key, 536
default statement, 471
DefaultCellEditor class, 667
creator, 651
methods, 641
defaultCharset method (Charset), 77, 415
DefaultHandler class, 198
DefaultMutableTreeNode class, 654, 672–673
add method, 655, 663
creator, 663
pathFromAncestorEnumeration method, 673
setAllowsChildren, setAsksAllowsChildren
methods, 661, 663
xxxFirstEnumeration, xxxOrderEnumeration
methods, 672, 676
defaultPage method (PrinterJob), 781
DefaultRowSorter class
  setComparator, setSortable methods, 628, 637
  setRowFilter method, 629, 637
defaults method (CatalogFeatures), 172, 179
DefaultTableCellRenderer class
  isCellEditable method, 641
DefaultTableModel class
  isCellEditable method, 641
DefaultTreeCellRenderer class, 674–676
  setXxxIcon methods, 676
DefaultTreeModel class, 664, 686
  automatic notification by, 666
  getPathToRoot method, 666
  insertNodeInto method, 665, 671
  isLeaf method, 661
  nodeChanged method, 665, 671
  nodesChanged method, 671
  reload method, 666, 671
  removeNodeFromParent method, 665, 671
defaultWriteObject method (ObjectOutputStream), 101
defineClass method (ClassLoader), 535, 541
DELETE method (HttpRequest.Builder), 278
DELETE statement (SQL), 292
  executing, 300–301, 317
  in batch updates, 346
  vs. methods of ResultSet, 327
delete, deleteIfExists methods (Files), 115
DeleteGlobalRef function (C), 827
deleteRow method (ResultSet), 327, 329
Delimiters, in text files, 72
@Deprecated annotation, 479–480
Depth-first enumerations, 672
depthFirstEnumeration method
  (DefaultMutableTreeNode), 672, 676
derbyclient.jar file, 294
DES (Data Encryption Standard), 600
DestroyJavaVM function (C), 850, 854
Device coordinates, 727
Diagnostic interface, 454
  methods of, 462
DiagnosticCollector class, 454
  constructor, 462
  getDiagnostics method, 462
DiagnosticListener interface, 454
DialogCallbackHandler class, 574
digest method (MessageDigest), 585, 587
DigiCert, 591, 593
Digital fingerprints, 94, 583–587
Digital signatures, 587–589
  verifying, 589–592
Direct buffers, 843
Directories
  creating, 113–114
  current, 122
  hierarchical structure of, 652
  printing all subdirectories of, 121
  traversing, 118–123
  user’s working, 64
DirectoryStream interface, 120
discarding method (BodyHandlers), 272
distinct method (Stream), 14–15, 50
dividedBy method (Duration), 358
doAs, doAsPrivileged methods (Subject), 568–569, 572
Doc interface, 793
DocAttribute interface, 799
  implementing, 801
  printing attributes of, 803–806
DocAttributeSet interface, 800–801
DocFlavorSet interface, 793–794, 796
DocPrintJob interface
  getAttributes method, 807
  print method, 795
DOCTYPE declaration (DTD), 171
  including in output, 208
Document interface
  createElement methods, 206–207, 209
  getDocumentElement method, 160, 168
Document flavors, for print services, 793–794
DocumentBuilderFactory class
  isIgnoringElementContentWhitespace method, 179
  isNamespaceAware method, 196
  isNamespaceAwareDecorator method, 196
  setEntityResolver method, 172, 177
  setEntityResolver method, 177
DocumentBuilderFactory class
  isIgnoringElementContentWhitespace method, 179
  isNamespaceAware method, 196
  isNamespaceAwareDecorator method, 196
  newDocument method, 206, 208, 221
  parse method, 168
  setEntityResolver method, 172, 177
  setErrorHandler method, 177
DocumentBuilderFactory class
  isIgnoringElementContentWhitespace method, 179
  isNamespaceAware method, 196
  isNamespaceAwareDecorator method, 196
  newInstance method, 160, 167
  setEntityResolver method, 172, 177
  setErrorHandler method, 177
  setIgnoringElementContentWhitespace method, 176, 179
  setNamespaceAware method, 182, 195–196, 199, 207
  setValidating method, 176, 179

Index
Index

@Documented annotation, 479, 482–483
dofinal method (cipher), 601, 603, 606–607
DOM (Document Object Model) parser, 159–160, 196–197
namespace-awareness of, 195, 199
trees in:
  accessing with XPath, 188–193
  analyzing, 162–164
  building, 197, 206–216, 221
  writing, 208–210
dom/JSONConverter.java, 165
DOMResult class, 221, 226
DOMSource class, 210, 220
DOUBLE data type (SQL), 293, 348
double type
  printing, 69
  streams of, 43
  type code for, 95, 831
  vs. C types, 817
  writing in binary format, 78
DoubleBuffer class, 132
doubles method
  of Random, 44, 47, 51
  of SplittableRandom, 51
DoubleStream interface, 43–48
  methods of, 47
DoubleSummaryStatistics class, 26, 29–30, 44, 48
doubleValue method (Number), 387
Downstream collectors, 36–40, 50
draw method (graphics2D), 695–697, 715
Drawings
  creating, 693–744
  printing, 772–782
drawXxx methods (graphics), 696
DriverManager class, 296
  getConnection method, 296–297, 307, 351
  setLogWriter method, 297
DROP TABLE statement (SQL), 297
  executing, 300–301
  in batch updates, 346
dropWhile method (Stream), 13
DSA (Digital Signature Algorithm), 587–588
DST, DST_Xxx composition rules, 737
DTDHandler class, 198
DTDs (Document Type Definitions), 170–179
  element content in, 173–174
entities in, 176
  external, 171
  in XML documents, 156, 171–179
  locating, 171–172
  unambiguous, 174
  URLs for, 171
Duration class
  between method, 355, 357
  dividedBy method, 358
  getSeconds method, 355
  immutability of, 355
  isNegative, isZero methods, 358
  minus, minusXXX methods, 358
  multipliedBy method, 358
  negated method, 358
  ofXXX methods, 357
  plus, plusXXX methods, 358
toXXX methods, 355, 358
Dynamic links, 850
Dynamic web pages, 457–463
E
\e, \E, in regular expressions, 138
Echo servers, 238–239
Eclipse IDE, 509
Edge detection, 766
Editors, custom, 642–652
element element (XML Schema), 180
ELEMENT element content (DTD), 173–174
Element interface, 485
  getAttribute method, 164, 168
  getSimpleNamespace method, 486
  getTagName method, 161, 168, 195
  setAttribute, setAttributeNS methods, 207, 209
Elements (XML)
  child, 157
    accessing in XPath, 189
    namespace of, 194
  constructing, 206
  counting, in XPath, 189
  empty, 156
  legal attributes of, 174
  names of, 161, 195
  root, 157, 179
  trimming whitespace in, 163
  vs. attributes, 157–158, 175, 216
Ellipse2D class, 697, 699
Ellipse2D.Double, Ellipse2D.Float classes, 697
Ellipses, bounding rectangle of, 697
E-mails
  sending, 278–281
  terminating lines in, 279
employee/Employee.c, 829
employee/Employee.java, 828
employee/EmployeeTest.java, 828
EMPTY element content (DTD), 173
empty method
  of Optional, 20–21
  of Stream, 5, 9
Empty tags (XML), 156
Encapsulation, 500
encode method (URLEncoder), 270
Encryption, 599–612
  final block padding in, 601
  of class files, 535, 537
end method
  of Matcher, 143, 145, 149–150
  of MatchResult, 146, 150
End cap styles, 716–718
End points, 702
End tags (XML), 156
endDocument method (ContentHandler), 201
endElement method (ContentHandler), 197–201
End-of-line character. See Line feed
Enterprise applications, 349–351
Enterprise JavaBeans (EJBs), 287
Entity references (XML), 158, 176
Entity resolvers, 160, 172
ENTITY, ENTITIES attribute types (DTDs), 174–175
EntityResolver interface, 178, 198
  resolveEntity method, 172, 178
entries method (ZipFile), 88
Entrust, 593
Entry class, 629
  getXXX methods, 638–639
EntryLogger class, 496
EntryLoggingAgent.mf file, 496
enum keyword, 102
EnumCombo class, 396
enumeration element (XML Schema), 180
Enumeration interface, 88
  hasMoreElements method, 857–859
  nextElement method, 672, 857–859
Enumerations
  of nodes, in a tree, 672–673
  typesafe, 102–103
  using attributes for, 175
EnumSyntax class, 802
EOFException class, 844
Epoch, 101, 354
equals method
  of Annotation, 472
  of Collator, 408
  of Permission, 559
  of Set, 561
Error handlers
  in native code, 844–849
  installing, 177
ErrorHandler class, 198
  methods of, 177–178
Escape hatch mechanism, 663
escape keyword (SQL), 320
Escapes
  in regular expressions, 73, 141
  in SQL, 319–321
Essential XML (Box et al.), 153, 216
Euro symbol, 393, 415
eval method
  of CompiledScript, 447
  of ScriptEngine, 441–443
evaluate, evaluateExpression methods (XPath), 190, 193
Event handlers, annotating, 465–471
Event listeners, 464
Event queues (AWT), accessing, 546
EventHandler class, 467
EventListenerList class, 686
Evins, Jim, 622
evn pointer (C), 820
Exceptions
  checked, 568
  from native code, 844–849
  in C++, 845
  in SQL, 304–306
  type use annotations in, 476
ExceptionXXX functions (C), 845, 849
Exclusive lock, 136
exclusiveOr method (Area), 714–715
exec/ExecSQL.java, 308
ExecutableElement interface, 485
execute method
  of ResultSet, CachedRowSet, 331–332
  of Statement, 301, 307, 321–322
executeBatch method (Statement), 346–347
executeLargeBatch method (Statement), 347
executeQuery method
  of PreparedStatement, 312, 317
  of Statement, 300–301, 324
executeUpdate method
  of PreparedStatement, 312, 317
  of Statement, 299, 301, 322, 345
e hausmethod (HttpClient.Builder), 277
ExecutorService interface, 454
exists method (Files), 116–117
exit method (System), 546
EXIT statement (SQL), 295
exports keyword, 506, 508, 510, 521
Extension class loader, 531
extern "C", in native methods (C++), 813
External entities, 176
Externalizable interface, methods of, 101–102
F
f (float), type code, 95, 831
\f, in regular expressions, 138
Factoring algorithms, 588
f a te rror method (ErrorHandler), 177–178
Field class
  getName, getType methods, 687
  implementing AnnotatedElement, 466
Fields
  accessing from:
    another class, 546
    native code, 825–830
    annotating, 464, 479
    transient, 100
file class
  separator constant, 64
  toPath method, 111–112
File permissions, 557
File pointers, 80
File systems, POSIX-compliant, 117
file: (URI scheme), 252, 553
file.encoding property, 77
file.separator property, 558
FileChannel class
  lock method, 135–136
  open, map methods, 125, 130
  tryLock method, 135–136
FileHandler class, 415
FileInputStream class, 63–67, 550, 559
  constructor, 66
  getChannel method, 130
  read method, 56
fileKey method (BasicFileAttributes), 118
FileLock class
  close method, 136
  isShared method, 135
FileNotFoundException, 266
FileOutputStream class, 63–67
  constructor, 67
  getChannel method, 130
FilePermission class, 548, 554
FileReader class, 550
Files
  accessing, 546
  channels for, 125
  closing, 118
  configuration, 135
  copying, 114
  creating, 113–114
  deleting, 115
  encrypting/decrypting, 607
  filtering, 120, 746
  generated automatically, 463, 484
  hierarchical structure of, 652
  I/O modes of, 84
  memory-mapped, 50, 124–134
  missing, 454
  moving, 114
  random-access, 80–85
    vs. buffered, 125
  reading, 64, 112–113
    as a string, 112
    by one byte, 56–59
    permissions for, 559
  total number of bytes in, 81
  traversing, 120–123
  with multiple images, 747–755
  writing, 112–113
Files class, 109, 112–123
  copy method, 114–115
  createXXX methods, 113–114
  delete, deleteIfExists methods, 115
  exists method, 116–117
  find method, 119
  getBytes method, 112
  getOwner method, 117
  isXXX methods, 116–117
  lines method, 6, 10, 50
  list method, 118
  move method, 114–115
  newDirectoryStream method, 120, 123
newXXXStream, newBufferedXXX methods, 112–113
readAllXXX methods, 113
readAttributes method, 117
size method, 116–117
walk method, 118
walkFileTree method, 121–123
write method, 113
FileSystem class
ggetPath method, 124
FileSystems class
newFileSystem method, 123–124
FileInfo class
toInstant method, 377
FileVisitor interface, 121–122
methods of, 121
fill method (Graphics2D), 695–696, 715
Filling shapes, 694–695, 724
fillXXX methods (Graphics), 696
filter method
  of BufferedImageOp, 763, 771
  of Optional, 18–19
  of Stream, 3–11, 15
FilteredRowSet interface, 330
filtering method (Collectors), 37, 40
Filters
  for images, 763–772
  for numbers, 629
  for table rows, 628–630
  glob patterns for, 120
  implementing, 629
FilterXXXStream classes, 65
Final block padding, 601
find method
  of Files, 119
  of Matcher, 145, 149
findAll method (Scanner), 146, 151
findAny method (Stream), 16
FindClass function (C), 826, 829, 834
findClass method (ClassLoader), 534, 541
findColumn method (ResultSet), 302
findFirst method (Stream), 15–16
Fingerprints, 94, 583–587
different for a class and its objects, 97
first method (ResultSet), 325, 328
firstDayOfMonth methods (TemporalAdjuster), 365
firstValue method (HttpHeaders), 273, 278
#FIXED attribute (DTD), 175
Fixed-size records, 81–82
flatMap method
  general concept of, 12
  of Optional, 21–25
  of Stream, 12
flatMapting method (Collectors), 37, 40
flip method (Buffer), 133–134
FLOAT data type (SQL), 293, 348
float type
  printing, 69
  streams of, 43
  type code for, 95, 831
  vs. C types, 817
  writing in binary format, 78
FloatBuffer class, 132
Floating-point numbers, 380, 387–394
flush method
  of CipherOutputStream, 608
  of Closeable, 60
  of Flushable, 60, 63
  of OutputStream, 57, 59
Flushable interface, 59, 61
close method, 59
flush method, 60, 63
fn keyword (SQL), 320
Folder icons, 661–662, 674
followRedirects method (HttpClient.Builder), 271, 277
Font render context, 733
forEach method (Stream), 25, 28
forEachOrdered method (Stream), 25
Forest (Swing), 652, 660
ForkJoinPool class
  commonPool method, 51
forLanguageTag method (Locale), 386
Format class, 376
format method
  of DateTimeFormatter, 371, 375, 401
  of Format, 411
  of MessageFormat, 410–411
  of NumberFormat, 388, 392
  of String, 385
Formatting
dates, 380, 394–401
messages, 409–413
numbers, 380, 387–394
formatting/Formatting.java, 374
Forms, processing, 261–270
forName method
  of Charset, 77
forName method (continued)
  of Class, 456
ForwardingJavaFileManager class
  constructor, 463
  getFileForOutput method, 463
fprintf function (C), 833
Frame class, 457
  from method
    of Instant, 376–377
    of ZonedDateTime, 376–377
FROM statement (SQL), 290
FTP (File Transfer Protocol), 256
ftp: (URI scheme), 252, 256
Function interface, 30
@FunctionalInterface annotation, 479
G
\G, in regular expressions, 140
Garbage collection
  and arrays, 842
  and native methods, 821
GeneralPath class, 697, 699, 703
  constructor, 713
generate method (Stream), 5, 9, 43
  @Generated annotation, 479–480
generateKey method (KeyGenerator), 602, 607
Generators, converting to streams, 49
Generic types, type use annotations in, 476
get method
  of AttributeSet, 802, 806
  of Bindings, 443
  of ByteBuffer, 126, 131
  of CharBuffer, 132
  of Optional, 19–22
  of Paths, 109, 111
  of ScriptEngine, 443
  of ScriptEngineManager, 443
  of Supplier, 10
GET method (HttpRequest.Builder), 271, 278
GET request (HTML), 262, 264
  building, 271
getActions method (Permission), 559
getAddress method (InetAddress), 234–235
getAdvance method (TextLayout), 735
getAllByName method (InetAddress), 234–235
getAllowedChildren method (TreeNode), 662
getAnnotation method (AnnotatedElement), 466, 470, 484, 486
getAnnotations method (AnnotatedElement), 470
getAnnotationsByType method (AnnotatedElement), 470, 484, 486
GetArrayLength function (C), 840, 843
getAscent method (TextLayout), 735
getAsXxx methods (OptionalXxx), 44, 48
getAttribute method (Element), 164, 168
getAttributes method
  of DocPrintJob, 807
  of Node, 164, 168
  of PrintService, 807
getAddressXxx methods (XMLStreamReader), 203, 206
getAuthority method (URI), 253
getAutoCommit method (Connection), 347
getAutoCreateRowSorter method (JTable), 616, 618
getAvailableCurrencies method (Currency), 394
getAvailableLocales method
  of Collator, 408
  of NumberFormat, 383, 388, 391
getAvailableZoneIds method (ZoneId), 367
getAverage method (XxxSummaryStatistics), 26, 29, 48
getBinaryStream method (Blob), 317, 319
getBlob method (ResultSet), 317–318
getBlockSize method (Cipher), 606
GetBooleanArrayElements function (C), 842–843
GetBooleanArrayRegion function (C), 842, 844
GetBooleanField function (C), 830
getBundle method (ResourceBundle), 418–421
getByName method (InetAddress), 234–235
GetByteArrayElements function (C), 842, 843, 858
GetByteArrayRegion function (C), 842–843
getByteField function (C), 830
getBytes method
  of Blob, 317–318
  of files, 112
getCandidateLocales method
  (ResourceBundle.Control), 418
getCategory method (Attribute), 801, 806
getCellEditorValue method (CellEditor), 641, 644–645, 652
getAddressXxx method (JTable), 636
getCertificates method (CodeSource), 551
getChannel method (FileXxxStream, RandomAccessFile), 130
getChar method (ByteBuffer), 126, 131
getCharacterStream method (Clob), 318–319
GetCharArrayElements function (C), 842–843
GetCharArrayRegion function (C), 842, 844
getCharContent method (SimpleJavaFileObject), 463
GetCharField function (C), 830
getChild method (TreeModel), 685–687, 692
getchildAt method (TreeNode), 671
getchildCount method
  of TreeModel, 685–687, 692
  of TreeNode, 671
getchildNodes method (Node), 161, 168
getClassLoader method (Class), 531, 540
getClip method (Graphics), 734, 775
getclob method (ResultSet), 317–318
getcodeSource method (ProtectionDomain), 551
cGetCollationKey method (Collator), 404, 408
getColorModel method (BufferedImage), 758, 762
getColumn method (TableColumnModel), 636
getColumnType method (TableModel), 622, 635
gentication method
  of ResultSetMetaData, 335
  of TableModel, 618–619, 622
getColumnName method (JTable), 635
getColumnName method (TableModel), 619, 622
getColumnNumber method
  of Diagnostic, 462
  of SAXParseException, 178
getColumnNameAllowed method (JTable), 636
getColumnXXX methods (ResultSetMetaData), 335, 344
getCommand method (RowSet), 332
getConcurrency method (ResultSet), 324–325, 328
getConnection method (DriverManager), 296–297, 307, 351
getConnectTimeout method (URLConnection), 260
getContentType method (URLConnection), 261
getContentType, getContentType methods
  (URLConnection), 254, 257, 261, 266
getAddressLength method (URLConnection), 254, 257, 260
getAddress method
  of AccessController, 572
  of ScriptEngine, 444
getAddressClassLoader method (Thread), 533, 541
getcount method (XxxSummaryStatistics), 29, 48
getCountry method (Locale), 35, 386
getCrc method (ZipEntry), 87
generate method (Currency), 392–393
getCurrencyCode method (Currency), 394
generateCurrencyInstance method (NumberFormat), 387
getCurrencyInstance method (NumberFormat), 387, 392–393
getData method (CharacterData), 163, 169
dataElements method
  of ColorModel, 763
  of Raster, 758, 762
dataSize method (DataTruncation), 306
dataSet method
  of ResultSet, 300, 302
  of URLConnection, 254, 257, 261
dataDifferent methods
  of LocalDate, 360, 362
  of ZonedDateTime, 370
dataLength method (Period), 363
dataDeclaredAnnotations method (AnnotatedElement), 471
dataDecomposition method (Collator), 408
dataDefault method (Locale), 384, 386
dataDefaultEditor method (JTable), 650
dataDefaultFractionDigits method (Currency), 394
dataDefaultName method (NameCallback), 581
dataDefaultRenderer method (JTable), 641, 650
dataDescent method (TextLayout), 735
dataDiagnostics method (DiagnosticCollector), 462
getDataBufferXXX functions (C), 843
dataDisplayCountry, getDataDisplayLanguage methods
  (Locale), 386
dataDisplayName method
  of DayOfWeek, Month, 373, 395
  of Locale, 384, 386, 388
dataDocumentElement method (Document), 160, 168
dataDoInput, getDataOutput methods (URLConnection), 259
dataDouble method
  of ByteBuffer, 126, 131
  of ResultSet, 300, 302
dataDoubleArrayElements function (C), 842–843
dataDoubleArrayRegion function (C), 842, 844
dataDoubleField function (C), 826, 830
dataEnclosedElements method (TypeElement), 486
dataEngineXXX methods (ScriptEngineManager), 441
dataEntry method (ZipFile), 88
dataErrorCode method (SQLException), 304–305
getErrorStream method (HttpURLConnection), 266, 270
ggetErrorWriter method (ScriptContext), 444
gExpiration method (URLConnection), 254, 257, 261
gExtensions method (ScriptEngineFactory), 441
GetFieldID function (C), 826, 830
getFields method (Class), 687
getFileForOutput method
  (ForwardingJavaFileManager), 463
gFileNotFoundException method (StackTraceElement), 111
gFilePointer method (RandomAccessFile), 81, 84
getFileSuffixes method (ImageReaderWriterSpi), 755
getFillsViewportHeight method (JTable), 618
getFirstChild method (Node), 163, 168
gFloat method (ByteBuffer), 126, 131
GetFloatArrayElements function (C), 842–843
GetFloatArrayRegion function (C), 842, 844
GetFloatField function (C), 830
gFontRenderContext method (Graphics2D), 733, 735
getFontNames method (ImageReaderWriterSpi), 755
getFragment method (URI), 253
getHeaderXxx methods (URLConnection), 254–256, 260
gHeight method
  of ImageReader, 748, 754
  of PageFormat, 775, 782
gHost method (URI), 253
gHostXxx methods (InetAddress), 235
gHour method
  of LocalTime, 366
  of ZonedDateTime, 371
gIdentifier method (Entry), 638
gModifiedSince method (URLConnection), 260
gImageableXxx methods (PageFormat), 776, 782
gImageXxxByXxx methods (ImageIO), 746, 753
gIndex method (DataTruncation), 306
gIndexOfChild method (TreeModel), 685, 692
gInputStream method
  of Socket, 231–232, 236
  of URLConnection, 254, 261, 264, 266
  of ZipFile, 88
gInstance method
  of AlphaComposite, 739, 744
  of Cipher, 600, 606
  of Collator, 408
  of Currency, 393–394
  of KeyGenerator, 606
  of Locale, 402
  of MessageDigest, 584–586
getInt method
  of ByteBuffer, 126, 131
  of ResultSet, 300, 302
GetIntArrayElements function (C), 842–843
GetIntArrayRegion function (C), 842, 844
GetInterface method (Invocable), 446
getField method (URLConnection), 254, 257, 261
getFileSuffixes method (ImageReaderWriterSpi), 755
getImageableXxx methods (PageFormat), 776, 782
getImageXxxByXxx methods (ImageIO), 746, 753
getIndex method (DataTruncation), 306
getIndexOfChild method (TreeModel), 685, 692
getInputStream method of Socket, 231–232, 236
  of URLConnection, 254, 261, 264, 266
  of ZipFile, 88
getInstance method of AlphaComposite, 739, 744
  of Cipher, 600, 606
  of Collator, 408
  of Currency, 393–394
  of KeyGenerator, 606
  of Locale, 402
  of MessageDigest, 584–586
getInputStream method of URLConnection, 254, 261, 264, 266
of ZipFile, 88
get Justiça method (URLConnection), 254, 257, 261
getExtensions method (ScriptEngineFactory), 441
GetFieldID function (C), 826, 830
getFields method (Class), 687
getFileForOutput method
  (ForwardingJavaFileManager), 463
gFileNotFoundException method (StackTraceElement), 111
goFilePointer method (RandomAccessFile), 81, 84
getFileSuffixes method (ImageReaderWriterSpi), 755
getFillsViewportHeight method (JTable), 618
goFirstChild method (Node), 163, 168
goFloat method (ByteBuffer), 126, 131
GetFloatArrayElements function (C), 842–843
GetFloatArrayRegion function (C), 842, 844
GetFloatField function (C), 830
goFontRenderContext method (Graphics2D), 733, 735
getFontNames method (ImageReaderWriterSpi), 755
goFragment method (URI), 253
getNameXxx methods (URLConnection), 254–256, 260
height method
  of ImageReader, 748, 754
  of PageFormat, 775, 782
getHost method (URI), 253
getNameXxx methods (InetAddress), 235
getHour method
  of LocalTime, 366
  of ZonedDateTime, 371
goIdentifier method (Entry), 638
getModifiedSince method (URLConnection), 260
goImageableXxx methods (PageFormat), 776, 782
goImageXxxByXxx methods (ImageIO), 746, 753
goIndex method (DataTruncation), 306
goIndexOfChild method (TreeModel), 685, 692
goInputStream method
  of Socket, 231–232, 236
  of URLConnection, 254, 261, 264, 266
  of ZipFile, 88
goInstance method
  of AlphaComposite, 739, 744
  of Cipher, 600, 606
  of Collator, 408
  of Currency, 393–394
  of KeyGenerator, 606
  of Locale, 402
  of MessageDigest, 584–586
getName method
  of Attributes, 202
  of NamedNodeMap, 169
  of NodeList, 161, 169
getNameNumber method
  of Diagnostic, 462
  of SAXParseException, 178
goLocale method (MessageFormat), 411
goLocalHost method (InetAddress), 234–235
getName method
  of Attributes, 202
  of Node, 196
  of XMLStreamReader, 206
goLocation method (CodeSource), 551
getLong method (ByteBuffer), 126, 131
GetLongArrayElements function (C), 842–843
GetLongArrayRegion function (C), 842, 844
GetLongField function (C), 830
getMax method (XxxSummaryStatistics), 26, 30, 48
getMaxConnections method (DatabaseMetaData), 343
getMaxStatements method (DatabaseMetaData), 303, 343
getMessage method (Diagnostic), 462
getMetaData method of Connection, 334, 343
do ResultSet, 335, 344
getMethodCallSyntax method (ScriptEngineFactory), 445
GetMethodID function (C), 835, 838
getMimeTypes method (ScriptEngineFactory), 441
getMIMETypes method (ImageReaderWriterSpi), 755
getMin method (XxxSummaryStatistics), 30, 48
getMinute method of LocalTime, 366
do ZonedDateTime, 371
getModel method (Entry), 638
getMonth, getMonthValue methods of LocalDate, 362
do ZonedDateTime, 370
getMonths method (Period), 363
getMoreResults method (Statement), 321–322
getName method of Attribute, 806
do Field, 687
of NameCallback, 581
of Permission, 561, 566
of Principal, 573
of PrintService, 793
of UnixPrincipal, 567
of XMLStreamReader, 206
do ZipEntry, 87
of ZipFile, 88
getNames method (ScriptEngineFactory), 441
getNamespaceURI method (Node), 195–196
getNano method of LocalTime, 366
do ZonedDateTime, 371
getNextEntry method (ZipInputStream), 85–86
getNextException method (SQLException), 304–305
getNextSibling method (Node), 164, 168
getNextWarning method (SQLWarning), 306
getNodeXXX methods (Node), 164, 169, 195
genNumberInstance method (NumberFormat), 387, 392
getNumXXX methods (Currency), 394
getNumXXX methods (ImageReader), 747, 754
getObject method of ResourceBundle, 420–421
do ResultSet, 300, 302
GetObjectArrayElement function (C), 840, 843
GetObjectClass function (C), 826–827
GetObjectField function (C), 826, 830
getOffset method (ZonedDateTime), 371
goOrientation method (PageFormat), 782
getOriginatingProvider method of ImageReader, 746, 754
do ImageWriter, 755
getOutputStream method of Cipher, 606
getOutputStream method of Socket, 232, 236
of URLConnection, 254, 261, 264
goOwner method (Files), 117
goPageCount method (Banner), 784
goPageSize method (CachedRowSet), 333
goParameter method (DataTruncation), 306
goParent method of ClassLoader, 540
do Path, 111
do TreeNode, 671, 673
goParentNode method (Node), 168
getPassword method of PasswordCallback, 582
do RowSet, 332
goPath method of FileSystem, 124
do TreeSelectionEvent, 684
of URI, 253
goPaths method (TreeSelectionEvent), 678, 684
goPathToRoot method (DefaultTreeModel), 666
goPercentInstance method (NumberFormat), 387, 392
goPixel, getPixels methods (Raster), 757, 762
goPointCount method (ShapeMaker), 704
goPort method (URI), 253
goPreviousSibling method (Node), 168
goPrincipals method (Subject), 572
goPrinterJob method (PrinterJob), 773, 781
goPrintService method (StreamPrintServiceFactory), 796
goPrompt method of NameCallback, 581
do PasswordCallback, 582
goProtectionDomain method (Class), 551
getQName method (Attribute), 202
getQualifiedName method (TypeElement), 486
getQuery method (URI), 253
getRaster method (BufferedImage), 756, 762
getReader method (ScriptContext), 444
getReaderXXX methods (ImageIO), 746, 753
getReadTimeout method (URLConnection), 260
getRequestProperties method (URLConnection), 260
getResourceAsStream method (URLClassLoader), 532
getSavepoint method (Calendar), 419, 421
getRow method (Resultset), 325, 328
getRowCount method (TableModel), 618–619, 622
getRowSelectionAllowed method (JTable), 636
getRowXXX methods (JTable), 635
getSavepointXXX methods (Savepoint), 347
getScaleInstance method (AffineTransform), 730–731
getSecond method
  of LocalTime, 366
  of ZonedDateTime, 371
getsSeconds method (Duration), 355
getSelectionModel method (JTable), 636
getSelectionPath method (JTree), 665, 670, 678, 683
getSelectionPaths method (JTree), 678, 683
getShearInstance method (AffineTransform), 730, 732
getShort method (ByteBuffer), 126, 131
getShortArrayElements function (C), 842–843
getShortArrayRegion function (C), 842, 844
getShortField function (C), 830
getsimpleName method (Element), 486
getSize method (ZipEntry), 87
getSource method (Diagnostic), 462
getSQLState method (SQLException), 304–305
getSQLStateType method (SQLException), 304
getStandardFilepopup method (JavaCompiler), 461
getStaticFieldID, GetStaticXXXField functions (C), 829–830
getStaticMXXXField function (C), 834, 839
getStrength method (Collator), 408
getString method
  of ResourceBundle, 419, 421
  of ResultSet, 300, 302
getStringXXX method (ResourceBundle), 421
getStringChars, GetStringLength functions (C), 822
getStringRegion function (C), 821
getStringUTFChars function (C), 820, 822, 824, 858
getStringUTFLength, GetStringUTFRegion functions (C), 821
getStringValue method (Entry), 639
getSubject method (LoginContext), 572
getSubString method (Blob), 318–319
getSum method (XXXSummaryStatistics), 29, 48
getSuperclass function (C), 871
getSymbol method (Currency), 394
getSystemClassLoader method (ClassLoader), 541
getSystemJavaCompiler method (ToolProvider), 453
getTableCellEditorComponent method
  (TableCellEditor), 643, 645, 651
ggetCellRendererComponent method
  (TableCellRender), 639, 651
getTableName method (CachedRowSet), 333
getTables method (DatabaseMetaData), 334, 343
getTagName method (Element), 161, 168, 195
getTask method (JavaCompiler), 453–454, 461
getter/setter pairs. See Properties
getText method (XMLStreamReader), 206
ggetTimeZone method (TimeZone), 377
getTransferSize method (DataTruncation), 306
getTranslateInstance method (AffineTransform), 730, 732
getTreeCellRendererComponent method
  (TreeCellRender), 675–676
ggetType method
  of Field, 687
  of ResultSet, 324, 328
getUpdateCount method (Statement), 302, 321
getURI method (Attribute), 202
getURL method (RowSet), 332
getURLs method (URLClassLoader), 532
getUserInfo method (URI), 253
getUsername method (RowSet), 332
getValue method
  of Attributes, 202
  of Copies, 802
  of Entry, 638
  of Win32RegKey, 857
getValueAt method (TableModel), 618–619, 622
getValueCount method (Entry), 639
getVendorName, getVersion methods (IIOServiceProvider), 746, 754
getWarnings method (Connection, ResultSet, Statement), 306
getWidth method
  of ImageReader, 748, 754
  of PageFormat, 775, 782
getWriter method (ScriptContext), 444
getWriterXXX methods (ImageIO), 746, 753
getYear method
  of LocalDate, 362
  of ZonedDateTime, 370
ggetYears method (Period), 363
GIF format, 745
  animated, 747
  image manipulations on, 767
  printing, 792
GlassFish server, 350
Glob patterns, 120
GlobalSign, 593
GMail, 279–280
Gnu C compiler, 813–814
Gödel’s theorem, 542
Google Maps, 263
GradientPaint class, 725
  constructor, 725–726
  cyclic parameter, 725
grant keyword, 553, 568, 599
Graphic Java™ (Geary), 614, 652
Graphics class, 693–694
  drawXXX, fillXXX methods, 696
  get/setClip methods, 733–734, 775
Graphics2D class
  clip method, 695, 733–735, 775
  draw method, 695–697, 715
  fill method, 695–696, 715
  getFontRenderContext method, 733, 735
  rotate, scale methods, 728, 732
  setComposite method, 695, 738, 744
  setPaint method, 695, 724, 726
  setRenderingHint, setRenderingHints methods, 694
  setStroke method, 694, 715, 724
  setTransform method, 730, 732
  shear method, 728, 732
  transform method, 695, 730, 732
  translate method, 728, 732, 784
Greenwich Royal Observatory, 354, 367
Gregorian calendar reform, 363
GregorianCalendar class
toZonedDateTime method, 376–377
GroovY programming language, 440, 448
  group method
    of Matcher, 143, 145, 149–150
    of MatchResult, 146, 150
  groupCount method (Matcher), 149
Grouping, 34–35
  classifier functions of, 35
  reducing to numbers, 36
  groupingBy method (Collectors), 34–40
  groupingByConcurrent method (Collectors), 35, 50
&lt; entity reference, 158
GUI (Graphical User Interface), scripting events for, 447–452

H
\h, \H, in regular expressions, 139
Half-closing connections, 243–244
Handbook of Applied Cryptography, The (Menezes et al.), 587
handle method (CallbackHandler), 581
handleGetObject method (ResourceBundle), 421
Handles (Swing), 657, 674
hash/Digest.java, 585
hashCode method
  of Annotation, 472
  of Permission, 559
HashXXXAttributeSet classes, 773, 800
Haskell programming language, 440
hasMoreElements method (Enumeration), 857–859
hasNext method (XMLStreamReader), 205
hasRemaining method (Buffer), 130
Header information, from server, 254
header method (HttpRequest.Builder), 277
Headers (Swing tables)
  rendering, 641
  scrolling, 615
headers method (HttpResponse), 272, 278
helloNative/HelloNative.c, 813
helloNative/HelloNative.h, 812
helloNative/HelloNative.java, 811
helloNative/HelloNativeTest.java, 814
Hex editors
creating class files in, 542
modifying bytecodes with, 544
Hidden commands, in XML comments, 159
Hosts, 234–235
HTML (HyperText Markup Language)
attributes in, 158
end and empty tags in, 156
forms in, 262
generating from XML files, 216–219
mixing with JSP, 457
printing, 792
vs. XML, 155
HTTP (Hypertext Transfer Protocol), 287
redirects between HTTPS and, 267
request headers in, 255–256
http: (URI scheme), 252, 553
HttpClient class, 271–278
enabling logging for, 273
newBuilder method, 271, 273, 277
newHttpClient method, 271, 277
send method, 277
sendAsync method, 273, 277
HttpURLConnection class
getAddressStream method, 266, 270
getErrorCode method, 267
setInstanceFollowRedirects method, 267
HTTPS (Hyper Text Transfer Protocol Secure), 267
https: (URI scheme), 252
HttpURLConnection class
getAddressStream method, 266, 270
getErrorCode method, 267
setInstanceFollowRedirects method, 267
I
I (int), type code, 95, 831
I/O streams. See Input streams, Output streams
IANA (Internet Assigned Numbers Authority), 367
IBM, 153, 160
DB2 database, 293
IBM437 encoding, 415
ICC profiles, 758
Icons
in column headers, 641
in table cells, 639
in trees, 661–662, 674
ID, IDREF, IDREFS attribute types (DTDs), 174–175
Identity (do-nothing) transformation, 209
identity method (Function), 30
Identity values, 41
IDs, uniqueness of, 175, 183
IETF BCP 47, 385
ifPresent method (Optional), 17–18, 48
ifPresentOrElse method (Optional), 18
IIOMage class, 748, 755
IIOServiceProvider class
getAddressName, getVersion methods, 746, 754
IllegalArgumentException, 204, 845
IllegalStateException, 30, 747
Imageable area, 776
ImageInputStream class, 747
ImageI0 class
createImageXxxStream methods, 747, 753
determining image type, 745
getImageXxxByXxx methods, 746, 753
getReaderXxx, getWriterXxx methods, 746, 753
read, write methods, 745, 753
imageIO/ImageIOFrame.java, 749
ImageOutputStream interface, 748
imageProcessing/ImageProcessingFrame.java, 767
ImageReader class, 745
getHeigh11, getWidth methods, 748, 754
getNumXxx methods, 747, 754
getOriginatingProvider method, 746, 754
read, readThumbnail methods, 754
setInput method, 754
ImageReaderWriterSpi class
getXxx methods, 755
Images
  blurring, 765
color values of, 758
detection of, 766
filtering, 763–772
getting size of, before reading, 748
incremental rendering of, 756
manipulating, 756–772
metadata in, 748
multiple, in a file, 747–755
printing, 772–782, 792, 796
raster, 744–772
  constructing from pixels, 756–763
  readers/writers for, 745–755
rotating, 764
superimposing, 735–736
thumbnails for, 748
vector, 693–744
ImageWriter class, 745, 748
  canInsertImage method, 748, 755
  getOriginatingProvider method, 755
  setOutput method, 755
  write, writeInsert methods, 748, 755
implements specification, type use annotations, in, 476
#IMPLIED attribute (DTD), 175
implies method
  of Permission, 559–561, 566
  of ProtectionDomain, 551
import statement, 534
INCLUDE environment variable, 854
include method (RowFilter), 629, 638
Incremental rendering, 756
Indexed color model, 765
IndexOutOfBoundsException class, 747
InetAddress class, 234–235
getXxx methods, 234–235
InetAddress(InetAddressTest.java), 234
InetAddress class
  isUnresolved method, 251
Infinite trees, 688
Inheritance trees, 304
@Inherited annotation, 479, 483
init method
  of Cipher, 606
  of KeyGenerator, 606
Initialization blocks, for shared libraries, 816
initialize method
  of LoginModule, 582
  of SimpleLoginModule, 574
INPUT_STREAM class (DocFlavor), 794
Input streams, 56–77
  and Unicode, 56
  as input source, 160
  buffered, 65–67
  byte processing in, 65
  byte-oriented, 56
  chaining, 65
  closing, 57
  encoding for, 68
  filters for, 63–67
  hierarchy of, 59–63
  keeping open, 243
  objects in, 88–108
InputSource class, 178
InputStream class, 56–59, 60
  available method, 57–58
  close method, 57–58
  mark, markSupported methods, 58
  read method, 56–58
  readAllBytes, readNBytes methods, 56, 58
  reset, skip, transferTo methods, 58
InputStreamReader class, 68
INSERT statement (SQL), 292
  and autogenerated keys, 322
  executing, 300–301, 317
  in batch updates, 346
  vs. methods of ResultSet, 327
insertNodeInto method (DefaultTreeModel), 665, 671
insertRow method (ResultSet), 326, 329
Inside Java™ 2 Platform Security (Gong et al.), 530
Instance fields
  accessing from native code, 825–829
  annotating, 475
  instanceof keyword, and type use annotations, 476
Instant class, 354
  from method, 376–377
  immutability of, 355
Instant class (continued)
- legacy classes and, 376–377
- minus, minusXXX methods, 357
- now method, 355, 357
- plus, plusXXX methods, 357
Instrumentation API, 496
int type
- printing, 69
- storing, 78
- streams of, 43
- type code for, 95, 831
- vs. C types, 817
- writing in binary format, 78
IntBuffer class, 132
INTEGER data type (SQL), 293, 348
IntegerSyntax class, 802
@interface declaration, 465, 471
Interfaces
- accessing script classes with, 446
- annotating, 475, 479
- implementing in script engines, 445
- Internet Engineering Task Force, 382
- Interpolation, 764
- for gradients, 725
- strategies of, 764
- when transforming images, 764
- Interruptible sockets, 244–251
- Interruptible/InterruptibleSocketTest.java, 246
- intersect method (Area), 714–715
- ints method
- of Random, 44, 47, 51
- of SplittableRandom, 51
- IntStream interface, 43–48
- average, max, min, of, range, rangeClosed, sum,
  summaryStatistics, toArray methods, 46
- boxed method, 44, 46
- IntSummaryStatistics class, 26, 29–30, 44, 48
- intValue method (Number), 388
- Invalid pointers (C, C++), 810
- InvalidPathException, 109
- Invocable interface, 444
- getInterface method, 446
- invokeXXX methods, 445–446
- Invocation API, 849–854
- invocation/InvocationTest.c, 851
- IOException, 231
- IP addresses, 229, 234–235
- IPP (Internet Printing Protocol) 1.1, 806
- IPv6 addresses, 234
- isAfter method
- of LocalDate, 363
- of LocalTime, 366
- of ZonedDateTime, 371
- isAfterLast method (ResultSet), 325, 328
- isAnnotationPresent method (AnnotatedElement), 470
- IsAssignableFrom function (C), 858, 870
- isBefore method
- of LocalDate, 363
- of LocalTime, 366
- of ZonedDateTime, 371
- isBeforeFirst method (ResultSet), 325, 328
- isCellEditable method
- of AbstractCellEditor, 644
- of AbstractTableModel, 641
- of CellEditor, 652
- of DefaultTableModel, 641
- of Model, 622, 641
- isCharacters method (XMLStreamReader), 205
- isClosed method
- of ResultSet, 303
- of Socket, 233
- of Statement, 302
- isConnected method (Socket), 233
- isDirectory method
- of BasicFileAttributes, 118
- of ZipEntry, 87
- isEchoOn method (PasswordCallback), 582
- isEndElement method (XMLStreamReader), 205
- isExecutable method (Files), 116–117
- isFirst method (ResultSet), 325, 328
- isGroupingUsed method (NumberFormat), 392
- isHidden method (Files), 116–117
- isIgnoringElementContentWhitespace method
- (DocumentBuilderFactory), 179
- isInputShutdown method (Socket), 244
- isLast method (ResultSet), 325, 328
- isLeaf method
- of DefaultTreeModel, 661
- of TreeModel, 663, 685, 693
- of TreeNode, 661–662
- isLeapYear method (LocalDate), 363
- isNamespaceAware method
- of DocumentBuilderFactory, 196
- of SAXParserFactory, 201
- isNegative method (Duration), 358
- ISO 216 standard, 420
- ISO 639-1 standard, 382, 386
ISO 3166-1 standard, 382, 386
ISO 4217 standard, 393–394
ISO 8601 standard, 320, 480
ISO 8859-1 standard, 68, 76
isOutputShutdown method (Socket), 244
isParseIntegerOnly method (NumberFormat), 392
isPresent method (Optional), 19–22
isReadable method (Files), 116–117
isRegularFile method of BasicFileAttributes, 118
of Files, 116–117
isShared method (FileLock), 135
isStartElement method (XMLStreamReader), 205
isSymbolicLink method of BasicFileAttributes, 118
of Files, 116–117
isUnresolved method (InetSocketAddress), 251
isValidating method of DocumentBuilderFactory, 179
of SAXParserFactory, 201
isWhiteSpace method (XMLStreamReader), 205
isWritable method (Files), 116–117
isZero method (Duration), 358
item method
of NamedNodeMap, 169
of NodeList, 161, 169, 177
Iterable interface, 120
spliterator method, 10
iterate method (Stream), 5, 9, 14, 43
Iterator interface, 300
iterator method
of BaseStream, 28
of SQLException, 304–305
of Stream, 25
Iterators, 25
converting to streams, 6, 49
splittable, 9–10

J (long), type code, 95, 831
JAAS (Java Authentication and Authorization Service), 566–582
configuration files in, 567, 572
login modules in, 573–582
jaas/jaas.config, 581
jaas/JAASTest.java, 580
jaas/JAASTest.policy, 580
jaas/SimpleCallbackHandler.java, 579
jaas/SimpleLoginModule.java, 577
jaas/SimplePrincipal.java, 576
jaas.config file, 569
JAR files
adding to class path, 532
analyzing dependencies of, 524–525
automatic registration in, 296
class loaders in, 532
code base of, 547
file resources in, 514
for plugins, 532
manifest of, 86, 515
META-INF/services directory, 522
modular, 510–511
resources in, 417
signing, 591–592
jar program, 510
jar: (URI scheme), 252
jarray type (C), 858
jarsigner program, 591–592, 599
JarXxxStream classes, 86
Java 2D API, 693–744
affine transformations in, 730
colors in, 758
constructive area geometry operations in, 714
features supported in, 694
filters in, 763–772
paint in, 724
printing in, 784
rendering pipeline, 694–696
sample values in, 756
shape classes in, 697, 702
strokes in, 715
transparency in, 735–738
Java Bug Database, 765
Java EE (Java Platform, Enterprise Edition), 287
Java Platform Module System, 499–527
migration to, 515–519
Java Plug-in, loading signed code, 598
java program
--add-exports, --add-opens, --illegal-access options, 518–519
--add-modules option, 271
- javaagent option, 496
jdbc.drivers property in, 296
-module, --module-path options, 503
-noverify option, 545
security managers in, 553
java program (continued)
specifying locales in, 384
Java programming language
internationalization support in, 379
platform-independent, 78
security of, 547–551
vs. SQL, 313
Java Virtual Machine Specification, The, 489, 544
java.activation module, 514
java.awt package, 501
java.awt.AlphaComposite API, 744
java.awt.BasicStroke API, 724
java.awt.Color API, 763
java.awt.font.TextLayout API, 735
java.awt.geom package, 100
java.awt.geom.AffineTransform API, 731–732
java.awt.geom.Arc2D.Double API, 713
java.awt.geom.Area API, 715
java.awt.geom.CubicCurve2D.Double API, 713
java.awt.geom.GeneralPath API, 713
java.awt.geom.Path2D API, 714
java.awt.geom.Path2D.Float API, 713
java.awt.geom.QuadCurve2D.Double API, 713
java.awt.geom.RoundRectangle2D.Double API, 712
java.awt.GradientPaint API, 726
java.awt.Graphics API, 734
java.awt.Graphics2D API, 696, 724, 726, 732, 735, 744
java.awt.image.AffineTransformOp API, 771
java.awt.image.BufferedImage API, 761
java.awt.image.BufferedImageOp API, 771
java.awt.image.ByteLookupTable API, 771
java.awt.image.ColorModel API, 763
java.awt.image.ConvolveOp API, 772
java.awt.image.kernel API, 772
java.awt.image.LookUpOp API, 771
java.awt.image.Raster API, 762
java.awt.image.RescaleOp API, 771
java.awt.image.ShortLookupTable API, 772
java.awt.image.WritableRaster API, 762
java.awt.print.PageFormat API, 782
java.awt.print.Printable API, 781
java.awt.print.PrinterJob API, 781–782
java.awt.TexturePaint API, 726
java.corba module, 514
java.io package, 77
java.io.BufferedInputStream API, 67
java.io.BufferedOutputStream API, 67
java.io.Closeable API, 62
java.io.DataInput API, 79–80
java.io.DataOutput API, 80
java.io.File API, 112
java.io.FileInputStream API, 66, 130
java.io.FileOutputStream API, 67, 130
java.io.Flushable API, 63
java.io.InputStream API, 58
java.io.OutputStream API, 93
java.io.ObjectInputStream API, 93
java.io.ObjectOutputStream API, 93
java.io.OutputStream API, 59
java.io.PrintWriter API, 70
java.io.pushbackInputStream API, 67
java.io.RandomAccessFile API, 84–85, 130
java.lang, java.lang.annotation packages, 478
java.lang.annotation.Annotation API, 472
java.lang.Appendable API, 63
java.lang.CharSequence API, 47, 63
java.lang.Class API, 540, 551
java.lang.ClassLoader API, 540–541
java.lang.Iterable API, 10
java.lang.Readable API, 63
java.lang.reflect.AnnotatedElement API, 470–471
java.lang.SecurityManager API, 550
java.lang.System API, 816
java.lang.Thread API, 541
java.logging module, 520
java.net package
socket connections in, 232
supporting IPv6 addresses in, 234
URLs vs. URLs in, 252
java.net.http package, 271
java.net.http.HttpClient API, 277
java.net.http.HttpRequest API, 277
java.net.http.HttpResponse API, 278
java.net.http.URLConnection API, 270
java.net.InetAddress API, 235
java.net.InetSocketAddress API, 251
java.net.ServerSocket API, 239
java.net.Socket API, 232–233, 244
java.net.URL API, 259
java.net.URLClassLoader API, 541
java.net.URLConnection API, 259–261
java.net.URLDecoder API, 270
java.net.URLEncoder API, 270
javax.xml.catalog.files system property, 172
javax.xml.parsers.DocumentBuilder API, 168, 177, 208
javax.xml.parsers.DocumentBuilderFactory API, 167, 179, 196
javax.xml.parsers.SAXParser API, 201
javax.xml.parsers.SAXParserFactory API, 201
javax.xml.stream.XMLInputFactory API, 204–205
javax.xml.stream.XMLOutputFactory API, 214
javax.xml.stream.XMLStreamReader API, 205–206
javax.xml.stream.XMLStreamWriter API, 214
javax.xml.transform.dom.DOMResult API, 226
javax.xml.transform.dom.DOMSource API, 210
javax.xml.transform.sax.SAXSource API, 225
javax.xml.transform.stream.StreamResult API, 210
javax.xml.transform.stream.StreamSource API, 225
javax.xml.transform.Transformer API, 209
javax.xml.transform.TransformerFactory API, 209
javax.xml.xpath.XPath API, 193
javax.xml.xpath.XPathEvaluationResult API, 193
javax.xml.xpath.XPathFactory API, 193
JAXB (Java Architecture for XML Binding), 512
JAXP (Java API for XML Processing) library, 160
jboolean type (C), 817
jbooleanArray type (C), 840
jbyte type (C), 817
jbyteArray type (C), 840
jchar type (C), 817, 819
jcharArray type (C), 840
JCheckBox class, 641
jclass type (C), 835
JComboBox class, 641
JCommander framework, 465
JComponent class
  paint method, 639, 694
  paintComponent method, 694
  putClientProperty method, 658, 663
JDBC API, 285
  configuration of, 293–299
  debugging, 297
  design of, 284–287
  tracing, 297
  uses of, 286–287
  versions of, 283
JDBC API Tutorial and Reference (Fisher et al.), 327, 349
JDBC drivers
  escape syntax in, 319–321
  JAR files for, 294
  registering classes for, 295–296
  scrollable/updatable result sets in, 324
  types of, 285–286
JDBC/ODBC bridge, not available in Java 8, 285
JdbcTemplate interface, 330
jdeprscan program, 480
jdeps program, 524–525
JDK (Java Development Kit)
  DOM parser, 160
  keytool program, 589
  obsolete features in, 500
  serialver program, 104
  src.jar file, 850
  SunJCE ciphers, 600
jdk.incubator.http package, 271
jdouble (C), 817
jdoubleArray (C), 840
jfloat (C), 817
jfloatArray (C), 840
jint (C), 817
jintArray (C), 840
JLabel class, 674–675
Jlink program, 526
jlong (C), 817
jlongArray (C), 840
JMOD files, 526, 531
jmod program, 526
JNDI service, 350
  class loaders in, 532
  JndiLoginModule class, 568
JNI (Java Native Interface), 810–871
  accessing:
    array elements in, 840–844
    functions in C++, 820
    calling convention in, 820
    debugging mode of, 850
    error handling in, 844–849
    invoking Java methods in, 833–840
    online documentation for, 821
  JNI_CreateJavaVM function (C), 850, 854
  JNI_OnLoad, JNI_OnUnload methods (C), 816
  jni.h file, 817
  JNI_CALL, JNIEXPORT macros, 812
  JobAttributes class (obsolete), 806
  jobject type (C), 835, 840, 858
jobjectArray type (C), 840
Join styles, 716–717
joining method (Collectors), 26, 29
JoinRowSet interface, 330
JPanel class, 777
JPEG format, 745
image manipulations on, 767
printing, 792
reading, 746
jrt: syntax, 527
js.properties file, 447
JScrollPane class, 615
JSF (JavaServer Faces), 261
JShell, loading modules into, 511
jshort type (C), 817
jshortArray type (C), 840
JSON-B (JSON Binding), 512, 514
JSP (JavaServer Pages), 457–463
jstring type (C), 820, 835, 858
JTable class, 613–652
addColumn method, 630, 636
asymmetric, 622
cell editors, automatically installed, 641
constructor, 618
convertXxxIndexToModel methods, 626, 636
default rendering actions, 622
generateKey method, 602, 607
getInstance method, 606
init method, 606
KeyPairGenerator class, 609
Keys
autogenerated, 322
generating, 602–607
primary, 322
keystore keyword, 599
KeyStoreLoginModule class, 568
Keystores, 589–592, 598
referencing in policy files, 599
keytool program, 589–592
KrbSLoginModule class, 568
<table>
<thead>
<tr>
<th>L</th>
<th>L (object), type code, 95, 831</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda expressions, with streams, 11</td>
<td></td>
</tr>
<tr>
<td>Landscape orientation, 730</td>
<td></td>
</tr>
<tr>
<td>Language codes, 35, 382</td>
<td></td>
</tr>
<tr>
<td>Language Model API, 485–486</td>
<td></td>
</tr>
<tr>
<td>Language tags, 385</td>
<td></td>
</tr>
<tr>
<td>last method (ResultSet), 325, 328</td>
<td></td>
</tr>
<tr>
<td>lastXXX methods (TemporalAdjuster), 365</td>
<td></td>
</tr>
<tr>
<td>lastXXXTime methods (BasicFileAttributes), 118</td>
<td></td>
</tr>
<tr>
<td>Layout algorithm, 783</td>
<td></td>
</tr>
<tr>
<td>layoutPages method (Banner), 783</td>
<td></td>
</tr>
<tr>
<td>Lazy operations, 3, 7, 14, 147</td>
<td></td>
</tr>
<tr>
<td>LCD displays, 758</td>
<td></td>
</tr>
<tr>
<td>LD_LIBRARY_PATH environment variable, 816, 854</td>
<td></td>
</tr>
<tr>
<td>Leap seconds, 354</td>
<td></td>
</tr>
<tr>
<td>Leap years, 359, 363</td>
<td></td>
</tr>
<tr>
<td>Learn SQL The Hard Way (Shaw), 287</td>
<td></td>
</tr>
<tr>
<td>Learning SQL (Beaulieu), 287</td>
<td></td>
</tr>
<tr>
<td>Leaves (Swing), 652, 660, 685</td>
<td></td>
</tr>
<tr>
<td>icons for, 661–662, 674</td>
<td></td>
</tr>
<tr>
<td>Legacy data, converting into XML, 221</td>
<td></td>
</tr>
<tr>
<td>length method</td>
<td></td>
</tr>
<tr>
<td>of Blob, 318</td>
<td></td>
</tr>
<tr>
<td>of CharSequence, 63</td>
<td></td>
</tr>
<tr>
<td>of Clob, 319</td>
<td></td>
</tr>
<tr>
<td>of RandomAccessFile, 81, 85</td>
<td></td>
</tr>
<tr>
<td>LIB environment variable, 854</td>
<td></td>
</tr>
<tr>
<td>lib/ext directory, 531</td>
<td></td>
</tr>
<tr>
<td>LIKE statement (SQL), 291, 320</td>
<td></td>
</tr>
<tr>
<td>limit method (Stream), 12, 50, 130</td>
<td></td>
</tr>
<tr>
<td>Line feed, 69, 414</td>
<td></td>
</tr>
<tr>
<td>in e-mails, 279</td>
<td></td>
</tr>
<tr>
<td>in regular expressions, 140</td>
<td></td>
</tr>
<tr>
<td>Line2D class, 69, 699</td>
<td></td>
</tr>
<tr>
<td>Line2D.Double, Line2D.Float classes, 697</td>
<td></td>
</tr>
<tr>
<td>lines method (Files), 6, 10, 50</td>
<td></td>
</tr>
<tr>
<td>lineTo method (Path2D.Float), 703, 713</td>
<td></td>
</tr>
<tr>
<td>Linux operating system</td>
<td></td>
</tr>
<tr>
<td>compiling invocation API, 854</td>
<td></td>
</tr>
<tr>
<td>library path in, 816</td>
<td></td>
</tr>
<tr>
<td>OpenSSL in, 596</td>
<td></td>
</tr>
<tr>
<td>using GNU C compiler, 813</td>
<td></td>
</tr>
<tr>
<td>list method (Files), 118</td>
<td></td>
</tr>
<tr>
<td>ListResourceBundle class, 420</td>
<td></td>
</tr>
<tr>
<td>Lists, converting to streams, 49</td>
<td></td>
</tr>
<tr>
<td>ListSelectionModel interface</td>
<td></td>
</tr>
<tr>
<td>setSelectedMode method, 626, 637</td>
<td></td>
</tr>
<tr>
<td>LITTLE_ENDIAN constant (ByteOrder), 131</td>
<td></td>
</tr>
<tr>
<td>Little-endian order, 78, 126, 416</td>
<td></td>
</tr>
<tr>
<td>Load time, 495</td>
<td></td>
</tr>
<tr>
<td>loadClass method</td>
<td></td>
</tr>
<tr>
<td>of ClassLoader, 534</td>
<td></td>
</tr>
<tr>
<td>of URLClassLoader, 532</td>
<td></td>
</tr>
<tr>
<td>loadLibrary method (System), 814, 816</td>
<td></td>
</tr>
<tr>
<td>LOBs (large objects), 317–319</td>
<td></td>
</tr>
<tr>
<td>creating empty, 319</td>
<td></td>
</tr>
<tr>
<td>placing in database, 318</td>
<td></td>
</tr>
<tr>
<td>reading, 317</td>
<td></td>
</tr>
<tr>
<td>Local hosts, 234</td>
<td></td>
</tr>
<tr>
<td>Local names, 196</td>
<td></td>
</tr>
<tr>
<td>Local variables, annotating, 475</td>
<td></td>
</tr>
<tr>
<td>Locale class</td>
<td></td>
</tr>
<tr>
<td>datesUntil method, 360, 363</td>
<td></td>
</tr>
<tr>
<td>getDayOfMonth methods, 360, 362</td>
<td></td>
</tr>
<tr>
<td>getMonth, getMonthValue methods, 362</td>
<td></td>
</tr>
<tr>
<td>getYear method, 362</td>
<td></td>
</tr>
<tr>
<td>isAfter, isBefore, isLeapYear methods, 363</td>
<td></td>
</tr>
<tr>
<td>legacy classes and, 377</td>
<td></td>
</tr>
<tr>
<td>minus, minusXXX methods, 360, 362</td>
<td></td>
</tr>
<tr>
<td>now method, 359, 362</td>
<td></td>
</tr>
<tr>
<td>of method, 359, 362</td>
<td></td>
</tr>
<tr>
<td>parse method, 376, 395, 401</td>
<td></td>
</tr>
<tr>
<td>plus, plusXXX methods, 359–360, 362</td>
<td></td>
</tr>
<tr>
<td>toLocalDate method, 377</td>
<td></td>
</tr>
<tr>
<td>until method, 359, 362</td>
<td></td>
</tr>
<tr>
<td>weekends in, 360</td>
<td></td>
</tr>
<tr>
<td>with method, 364–365</td>
<td></td>
</tr>
<tr>
<td>withXXX methods, 362</td>
<td></td>
</tr>
<tr>
<td>localdates/LocalDates.java, 361</td>
<td></td>
</tr>
<tr>
<td>LocaleDateTime class, 366</td>
<td></td>
</tr>
<tr>
<td>atZone method, 367</td>
<td></td>
</tr>
<tr>
<td>legacy classes and, 377</td>
<td></td>
</tr>
<tr>
<td>parse method, 395, 401</td>
<td></td>
</tr>
<tr>
<td>toLocalDateTime method, 377</td>
<td></td>
</tr>
<tr>
<td>Locale class, 31, 381–386</td>
<td></td>
</tr>
<tr>
<td>constructor, 385</td>
<td></td>
</tr>
<tr>
<td>debugging, 386</td>
<td></td>
</tr>
<tr>
<td>forLanguageTag method, 386</td>
<td></td>
</tr>
<tr>
<td>getCountry method, 35, 386</td>
<td></td>
</tr>
<tr>
<td>getDefault method, 384, 386</td>
<td></td>
</tr>
<tr>
<td>getDisplayCountry, getDisplayLanguage methods, 386</td>
<td></td>
</tr>
<tr>
<td>getDisplayName method, 384, 386, 388</td>
<td></td>
</tr>
<tr>
<td>getInstance method, 402</td>
<td></td>
</tr>
<tr>
<td>getISOCountries method, 384, 386</td>
<td></td>
</tr>
<tr>
<td>getISOLanguages method, 384</td>
<td></td>
</tr>
</tbody>
</table>
Locale class (continued)
  getLanguage method, 386
  setDefault method, 384, 386
  toLanguageTag method, 383, 386
  toString method, 386
Locales, 30, 380–386
  and resources bundles, 417–418
  current, 410
  default, 372, 384
  display names of, 384
  formatting styles for, 373, 395–396
  numbers in, 388
  predefined, 383
  variants in, 381, 418
LocalTime class, 365–366
  getXXX methods, 366
  isAfter, isBefore methods, 366
  legacy classes and, 377
  minus, minusXXX methods, 365–366
  now method, 365–366
  of method, 365–366
  parse method, 395, 401
  plus, plusXXX methods, 365–366
  toLocalTime method, 377
  toXXX0fDay methods, 366
  withXXX methods, 366
lock method (FileChannel), 135–136
Locks
  for the tail portion of a file, 135
  shared, 135
  unlocking, 135
Log files, 415
Log messages, adding to classes, 490–495
@LogEntry annotation, 490
Logging, code generation for, 463
logging.properties file, 273
LoggingPermission class, 557
LoginContext class, 567
  constructor, 571
  getSubject method, 572
  login, logout methods, 567, 572
LoginException, 572
LoginModule interface
  developer’s guide for, 575
  methods of, 582
Logins
  committed, 575
  modules for, 568
  custom, 573–582
  separating from action code, 575
Long class
  MAX_VALUE constant, 135
  LONG VARCHAR data type (SQL), 348
long type
  printing, 69
  streams of, 43
  type code for, 95, 831
  vs. C types, 817
  writing in binary format, 78
LONG VARCHAR data type (SQL), 348
LongBuffer class, 132
longs method
  of Random, 44, 47, 51
  of SplittableRandom, 51
LongStream interface, 43–48
  methods of, 46
LongSummaryStatistics class, 26, 29–30, 44, 48
Look-and-feel
  displaying trees in, 658
  handles for subtrees in, 674
  selecting multiple nodes in, 678
lookingAt method (Matcher), 149
lookup method (MethodHandles), 514
Lookup tables, 420
Lookup0p class, 764–765
  constructor, 771
lookupPrintServices method (PrintServiceLookup), 793, 795
lookupStreamPrintServiceFactories method
  (StreamPrintServiceFactory), 796
LookupTable class, 765
LSB (least significant byte), 78
LSOutput interface, 208
LSSerializer interface, 208
&lt;, entity reference, 158

M
Mac OS X
  character encodings in, 413
  OpenSSL in, 596
  resources in, 417
Mail messages/headers, 278–281
mail/MailTest.java, 280
main method
  executing, 530
  setting security managers in, 553
makeShape method (ShapeMaker), 704
makeVisible method (JTree), 666, 670
Mandelbrot set, 759
Mangling names, 813, 831
Manifest files, 86
map method
  of FileChannel, 125, 130
  of HttpHeaders, 272, 278
  of Optional, 18–19
  of Stream, 11
mapping method (Collectors), 37, 40
Maps
  concurrent, 31
  of stream elements, 30–34, 50
mapToInt method (Stream), 42
mapToXxx methods (XxxStream), 44
mark method
  of Buffer, 133–134
  of InputStream, 58
Marker annotations, 473
markSupported method (InputStream), 58
Mastering Regular Expressions (Friedl), 142
match attribute (XSLT), 218
match/HrefMatch.java, 146
Matcher class
  end method, 143, 145, 149–150
  find method, 149
  group method, 143, 145, 149–150
  groupCount method, 149
  lookingAt method, 149
  matches method, 142, 149
  quoteReplacement method, 150
  replaceXxx methods, 148, 150
  results method, 146, 150
  start method, 143, 145, 149–150
matcher method (Pattern), 142, 149
matcher method (Matcher), 142, 149
MatchResult interface, 146
  methods of, 146, 150
Matrices, transformations of, 729–730
max method
  of primitive streams, 44, 46–47
  of Stream, 15–16
MAX_VALUE constant (Long), 135
maxBy method (Collectors), 36, 40
maxOccurs attribute (XML Schema), 181
MD5 algorithm, 584
Memory addresses, vs. serial numbers, 92
Memory mapping, 124–134
MemoryMap/MemoryMapTest.java, 127
MessageDigest class
  digest method, 585, 587
  extending, 584
  get Instance method, 584–586
  reset method, 587
  update method, 585–586
MessageFormat class, 409–413, 422
  applyPattern method, 411
  constructor, 410
  format method, 410–411
  get/setLocale methods, 411
  ignoring the first limit, 413
Meta-annotations, 465, 481–484
Metadata (databases), 334–344
Metal look-and-feel
  selecting multiple nodes in, 678
trees in, 657–658
Method class, 466
Method references, type use annotations in, 476
MethodHandles class
  lookup method, 514
Methods
  adding logging messages to, 490–495
  annotating, 464, 475, 479
  calling from native code, 833–840
  getters/setters, generated automatically, 489
  instance, 833–834
  mangling names of, 813, 831
  of annotation interfaces, 472
  overriding, 480
  protected, 532
  signatures of, 831–832
  static, 834–835
Microsoft
  compiler, 813–814
    invocation API in, 854
  Notepad, 76
  ODBC API, 284
  SQL Server, 293
MIME (Multipurpose Internet Mail Extensions), 745
  for print services, 794
MimeMessage class, methods of, 279
min method
  of primitive streams, 44, 46–47
  of Stream, 15–16
minBy method (Collectors), 36, 40
minoccurs attribute (XML Schema), 181
minus, minusXXX methods
  of Duration, 358
  of Instant, 357
  of LocalDate, 360, 362
  of LocalTime, 365–366
  of Period, 363
  of ZonedDateTime, 370
MissingResourceException, 418
Miter join, 716–717
Miter limit, 717
Mixed content (XML), 157
  parsing, 174
Modernist painting example, 215
Modified UTF-8, 78–80, 416
  and native code, 819–822
Module class
  getResourceAsStream method, 514
module keyword, 503
module-info.class file, 510, 515
module-info.java file, 503, 515
Modules, 499–527
  accessing, 511–514, 518
  automatic, 515–518
  declaration of, 503–504
  explicit, 517
  exporting packages, 506–510
  loading into JShell, 511
  migration to, 515–519
  naming, 501–502, 515
  not passing access rights, 506
  open, 513
  opening packages in, 513
  packages with the same names in, 510
  qualified exports of, 521
  requiring, 504–506
  service implementations and, 522
  tools for, 524–527
  unnamed, 517–518
  versioning, 501
Monads, 12
Month enumeration, 359
  getDisplayName method, 373, 395
MonthDay class, 360
move method (Files), 114–115
moveColumn method (JTable), 630, 636
moveTo method (Path2D.Float), 703, 713
moveToCurrentRow method (ResultSet), 326, 329
moveToInsertRow method (ResultSet), 326, 328
MySQL database, 293

N

  as line feed, 69, 164, 279, 414
  in regular expressions, 138–139
NameCallback class, 574
  constructor, 581
  methods of, 581
NamedNodeMap interface
  getLength, item methods, 169
names method (Win32RegKey), 857
Namespaces, 193–196
  activating processing of, 182
  aliases (prefixes) for, 179, 195
  of attributes, 195
  of child elements, 194
  using class loaders as, 534
Nashorn engine, 440–441
National Institute of Standards and Technology, 228, 584
native keyword, 810
Native methods
  and garbage collection, 821
  array elements in, 840–844
  class references in, 827
  compiling, 813
  enumerating keys with, 858
  error handling in, 844–849
  exceptions in, 845
  instance fields in, 825–829
  invoking Java constructors in, 835
  linking to Java, 816
  naming, 811–812
  overloading, 811
  reasons to use, 810
  registry access functions in, 857–871
  static, 811
  static fields in, 829–830
  strings in, 819
native2ascii program, 419
NCHAR, NCOLB data types (SQL), 348–349
negated method (Duration), 358
Nervous text applet, 582
net.properties file, 273
NetPermission class, 556
newBufferedXxx methods (Files), 112–113
newBuilder method (HttpClient, HttpRequest), 271–273, 277
NewDirectByteBuffer function (C), 843
newDirectoryStream method (Files), 120, 123
newDocument method (DocumentBuilder), 206, 208, 221
newDocumentBuilder method (DocumentBuilderFactory), 160, 167, 207
newFactory method (RowSetProvider), 330, 333
newFileSystem method (FileSystems), 123–124
NewGlobalRef function (C), 827
newHttpClient method (HttpClient), 271, 277
of SAXParserFactory, 198, 201
of TransformerFactory, 209
of XMLInputFactory, 204
of XMLOutputFactory, 210, 214
of XPathFactory, 190, 193
NewObject function (C), 835, 840, 844
newOutputStream method of Channels, 251
of Files, 112–113
of SAXParserFactory, 198, 201
of TransformerFactory, 209
of XMLInputFactory, 204
of XMLOutputFactory, 210, 214
of XPathFactory, 190, 193
NewString function (C), 822
NewStringUTF function (C), 820, 822, 824, 857
newTransformer method (TransformerFactory), 209, 225
NewXxxArray functions (C), 843, 857
next method of ResultSet, 300, 302, 322
temporalAdjusters, 365
of XMLStreamReader, 205
nextElement method (Enumeration), 672, 857–859
nextOrSame method (TemporalAdjusters), 365
nextPage method (CachedRowSet), 331, 333
NMTOKEN, NMTOKENS attribute types (DTDs), 174–175
Node interface
appendChild method, 207, 209
getAttributes method, 164, 168
cGetChildNodes method, 161, 168
getFirstChild method, 161, 168
getLastChild method, 164, 168
getLocalName method, 196
getNamespaceURI method, 195–196
getNextSibling method, 164, 168
getNodeXXX methods, 164, 169, 195
getParentNode method, 168
getPreviousSibling method, 168
subinterfaces of, 161
Node renderer, 661
nodeChanged method (DefaultTreeNode), 665, 671
NodeList interface, 161
gelLength method, 161, 169
temitem method, 161, 169, 177
Nodes (Swing), 652
adding/removing, 665
child, 652, 655
collapsed, 668
connecting lines for, 658–659
currently selected, 663
editing, 667, 686
enumerating, 672–673
expanding, 666, 668
handles for, 657, 659, 674
highlighting, 674
identifying, by tree paths, 663
making visible, 666
parent, 652, 655
rendering, 674–676
root, 652–660
row positions of, 665
searching, for a given user object, 673, 678
selecting, 677
user objects for, 654, 665
nodesChanged method (DefaultTreeNode), 671
Nondeterministic parsing, 174
noneMatch method (Stream), 16
Noninterference, of stream operations, 7
@NonNull annotation, 476–477
normalize method (Path), 110–111
Normalized color values, 757
Normalizer class, 404
   normalize method, 409
NoSuchAlgorithmException, 586, 606
NoSuchElementException, 19–20, 859
notFilter method (RowFilter), 629, 638
NotSerializableException, 100
now method
   of Instant, 355, 357
   of LocalDate, 359, 362
   of LocalTime, 365–366
   of ZonedDateTime, 370
NTLoginModule class, 568–569
NTUserPrincipal class, 569
NullPointerException, 845
vs. Optional, 15
Number class
   doubleValue method, 387
   intValue method, 388
numberFilter method (RowFilter), 629, 638
NumberFormat class, 387–394
   format method, 388, 392
   get/setXxxDigits methods, 392
   getAvailableLocales method, 383, 388, 391
   getCurrencyInstance method, 387, 392–393
   getNumberInstance method, 387, 392
   getPercentInstance method, 387, 392
   is/setGroupingUsed methods, 392
   is/setParseIntegerOnly methods, 392
   parse method, 387–388, 392
   setCurrency method, 393
numberFormat/NumberFormatTest.java, 389
Numbers
   filtering, 629
   floating-point, 380, 387–394
   formatting, 380, 387–394
      supported locales for, 388
      with C, 817
   from grouped elements, 36
   in regular expressions, 139, 141
   printing, 69
   random, 5, 12, 44
   reading:
      from files, 64
      from ZIP archives, 66
      using locales, 387
   truly random, 602
   writing in binary format, 78
NUMERIC data type (SQL), 293, 348
NVARCHAR data type (SQL), 348

0
Object class
   clone method, 89, 106
Object inspection tree, 684–693
Object serialization, 88–108
   cloning with, 106–108
   file format for, 93–100
   modifying default, 100–102
   of singletons, 102–103
   serial numbers for, 90–91
ObjectInputStream class, 89
   constructor, 93
   readObject method, 89, 93, 101
ObjectOutputStream class, 88
   constructor, 93
   defaultWriteObject method, 101
  .writeObject method, 88, 93, 100
Object-relational mappers, 512
Objects
   cloning, 106–108
   converting to streams, 6, 23
   fingerprints of, 95
   life cycle of, 480
   printing, 69
   reading from an input stream, 89
   saving:
      in database, 483
      in output streams, 88, 90
      in text format, 72–75
   serializable, 88–93
   transmitting over network, 92
   type code for, 95, 831
   versioning, 103–106
objectStream/ObjectStreamTest.java, 92
ODBC API, 284, 286
of method
   of DoubleStream, 47
   of IntStream, 43, 46
   of LocalDate, 359, 362
   of LocalTime, 365–366
   of LongStream, 46
   of Optional, 20–21
   of Period, 363
   of Stream, 5, 9
   of ZonedDateTime, 367, 370
   of ZoneId, 367
ofDateAdjuster method (TemporalAdjusters), 364
ofDays method
   of Duration, 357
... of Period, 359, 363, 368
of file, ofFileDownload methods (BodyHandlers), 272
OffsetDateTime class, 368
ofHours method (Duration), 357
ofInstant method (ZonedDateTime), 370
ofLocalizedXxx methods (DateTimeFormatter), 372, 375, 394, 401
ofMillis, ofMinutes methods (Duration), 357
ofMonths method (Period), 360, 363
ofNanos, ofSeconds methods (Duration), 357
ofNullable method
  of Optional, 20–21
  of Stream, 6, 9, 23
ofPattern method (DateTimeFormatter), 373, 376
ofString method (BodyHandlers, BodyPublishers), 272–273
ofWeeks, ofYears methods (Period), 363
oj literal (SQL), 320
open keyword, 513
open method
  of FileChannel, 125, 130
  of SocketChannel, 244, 251
openConnection method (URL), 254, 259
Opened nonleaf icons, 661–662, 674
openOutputStream method (SimpleJavaFileObject), 463
opens keyword, 513, 521
OpenSSL toolkit, 596–597
openStream method (URL), 251, 259
Operating system
  character encodings in, 77, 413
  paths in, 64, 109
  resources in, 417
Operations
  associative, 41
  lazy, 3, 7, 14, 147
  stateless, 49
Optional class, 15–22
  creating values of, 20
  empty method, 20–21
  filter method, 18–19
  flatMap method, 21–25
  for empty streams, 41
  get method, 19–22
  ifPresent method, 17–18, 48
  ifPresentOrElse method, 18
  isPresent method, 19–22
  map method, 18–19
  of, ofNullable methods, 20–21
  or method, 19
  orElse method, 15, 17, 48
  orElseGet method, 17, 48
  orElseThrow method, 17, 20
  stream method, 22–25
optional keyword, 568
optional/OptionalTest.java, 23
OptionalXxx classes, 44, 48
Oracle JVM implementation, 842
ORDER BY statement (SQL), 300
order method (ByteBuffer), 126, 131
orFilter method (RowFilter), 629, 638
org.omg.corba package, 500
org.w3c.dom package, 160
org.w3c.dom.CharacterData API, 169
org.w3c.dom.Document API, 168, 209
org.w3c.dom.Element API, 168, 209
org.w3c.dom.NamedNodeMap API, 169
org.w3c.dom.Node API, 168–169, 196, 209
org.w3c.dom.NodeList API, 169
org.xml.sax.Attributes API, 202
org.xml.sax.ContentHandler API, 201
org.xml.sax.EntityResolver API, 178
org.xml.sax.ErrorHandler API, 178
org.xml.sax.helpers.AttributesImpl API, 226
org.xml.sax.InputSource API, 178
org.xml.sax.SAXParseException API, 178
org.xml.sax.XMLReader API, 226
Orientation class, 102–103
Outer joins, 320
OutOfMemoryError, 845
output element (XSLT), 218
Output streams, 56–77
  and Unicode, 56
  buffered, 65–67
  byte processing in, 65
  byte-oriented, 56
  closing, 57, 243
  filters for, 63–67
  hierarchy of, 59–63
  objects in, 88–108
OutputStream class, 56, 60–61, 210
  close method, 59
  flush, write methods, 57, 59
OutputStreamWriter class, 68
OverlappingFileLockException, 136
Overloading, 811
@Override annotation, 479–480
P
\p, \P, in regular expressions, 138
Package class, implementing AnnotatedElement, 466
package-info.java file, 475
Packages, 500
  annotating, 475, 479
  avoiding name clashes with, 193, 534
  exporting, 506–510
  hidden, 510
  opening, 513
  split, 510
Packets, 232
Padding schemes, 601
Page setup dialog box, 777–778
Pageable interface
  implementing, 782
  objects, printing, 792
PageAttributes class (obsolete), 806
pageDialog method (PrinterJob), 776, 778, 781
PageFormat class
  getHeight, getWidth methods, 775, 782
  getImageableXXX methods, 776, 782
  getOrientation method, 782
Pages
  measurements of, 776
  multiple, printing, 782–792
  orientation of, 730, 776, 782
Paint interface, 724–726
paint method (JComponent), 639, 694
paintComponent method
  of JComponent, 694
  of StrokePanel, 719
Paper margins, 775
Paper sizes, 420, 775
parallel method (BaseStream), 48, 53
parallel/ParallelStreams.java, 51
parallelStream method (collection), 2–3, 5, 48
Parameter variables, annotating, 475
Parent nodes (Swing), 652, 655
parse method
  of DateTimeFormatter, 373
  of DocumentBuilder, 168
  of LocalDate, 376, 395, 401
  of LocalDateTime, LocalTime, 395, 401
  of NumberFormat, 387–388, 392
  of SAXParser, 198, 201
  of XMLReader, 226
  of ZonedDateTime, 376, 395, 401
Parsed character data, 173
ParseException, 388, 391
Parsers, 159–169
  checking uniqueness of IDs in, 175, 183
  pull, 202
  validating in, 170
Parsing (XML), 159–169
  nondeterministic, 174
  with XML Schema, 182
partitioningBy method (Collectors), 35, 37
PasswordCallback class, 574
  constructor, 581
  methods of, 582
Password-protected resources, 256
Path interface, 109–112
  getXXX methods, 111
  normalize, relativize, resolve, resolveSibling,
    toXXX methods, 110–111
Path2D class, 699
  append, closePath methods, 703, 714
Path2D.Double class, 679
Path2D.Float class, 697
  methods of, 703, 713
pathFromAncestorEnumeration method
  (DefaultMutableTreeNode), 673
Paths (file system), 109–112
  absolute vs. relative, 64, 109–110
  checking properties of, 116–118
  filtering, 119
  relativizing, 110
  resolving, 66, 110
  root component of, 109
  separators in, 64, 109
Paths (graphics), 703–704
Paths class, 123
  get method, 109, 111
Pattern class, 142
  compile method, 142, 148
  matcher method, 142, 149
  split method, 147, 149
  splitAsStream method, 6, 10, 147
Patterns, 137–151
#PCDATA element content (DTD), 173
PDF format, printing, 792
peek method (Stream), 14–15
PEM (Privacy Enhanced Mail), 596
Pentium processor, little-endian order in, 78
Percentages, formatting, 387–394
Performance
  of encryption algorithms, 608
  of file operations, 125–132
Period class
  getXXX, of, minus, minusXXX, plus, plusXXX,
  withXXX methods, 363
  ofXXX methods, 359–360, 363, 368
  using for daylight savings time, 368
Perl programming language, regular
  expressions in, 142
Permission class
  constructor, 566
  equals method, 559
  extending, 559
  getActions method, 559
  getName method, 561, 566
  hashCode method, 559
  implies method, 559–561, 566
Permission files, 548
permission keyword, 554, 558
Permissions, 547–551
  call stack of, 550
  class hierarchy of, 549
  commonly used classes for, 554–557
  for files, 557
  for users, 567
  implementing, 559–566
  implying other permissions, 561
  in policy files, 551–559
  mapping code sources to, 547
  order of, 559
  property, 558
  restricting to certain users, 569
  socket, 558
  targets of, 557–558
permissions/PermissionTest.java, 564
permissions/WordCheckPermission.java, 562
@Persistent annotation, 483
Personal data, transferring, 600
Picocli framework, 465
Pixels
  affine transformations on, 764
  average value of, 766
  composing, 735–744
  interpolating, 725, 764
  reading, 756
  setting individual, 756–763
Placeholders, in message formatting, 409–413
Platform class loader, 531
Plugins, loading, 532
plus, plusXXX methods
  of Duration, 358
  of Instant, 357
  of LocalDate, 359–360, 362
  of LocalTime, 365–366
  of Period, 363
  of ZonedDateTime, 370
PNG format, 745
  printing, 792
Point2D class, 698–699
Point2D.Double, Point2D.Float classes, 697
Points, in typography, 775
Policy class, 548, 552
Policy files, 551–559
  and Java Plug-in, 598
  building, 598
  locations for, 551
  parsing, 561
  platform-independent, 558
  referencing keystores in, 599
  system properties in, 558
  user roles in, 566–582
  visibility of, 562
Policy managers, 551–559
Polygons, 697, 703
Pools, for parallel streams, 51
populate method (CachedRowSet), 331, 333
Porter–Duff rules, 736–739
Portrait orientation, 776
Ports, 229
  blocking, 228
  in URIs, 253
position function (XPath), 220
position method (Buffer), 134
POSIX-compliant file systems, 117
PosixFileAttributes interface, 117
POST method (HttpRequest.Builder), 272, 278
POST request (HTML), 262, 264–266
  building, 272
post/PostTest.java, 268
@PostConstruct annotation, 479–480
PostgreSQL database, 293
  connecting to, 297
  drivers for, 294
Postorder traversal, 672
postOrderEnumeration method
  (DefaultMutableTreeNode), 672, 676
PostScript format
  printing, 792, 796
  writing to, 796
postVisitDirectory method
  of FileVisitor, 121
  of SimpleFileVisitor, 123
Predefined character classes, 138–141
@PreDestroy annotation, 479–480
Predicate functions, 35
preamble method (Instrumentation API), 496
preOrderEnumeration method
  (DefaultMutableTreeNode), 672, 676
Prepared statements, 311–317
  caching, 312
  executing, 311
PreparedStatement interface
  clearParameters method, 317
  executeXXX, setXXX methods, 311, 317
  prepareStatement method (Connection), 311, 317, 323, 327
  previous method (ResultSet), 324, 328
  previous, previousOrSame methods
    (TemporalAdjusters), 365
  previousPage method (CachedRowSet), 333
  preVisitDirectory method
    of FileVisitor, 121
    of SimpleFileVisitor, 123
Primary keys, 322
Primitive types
  arrays of, 843
  I/O in binary format in, 59
  streams of, 42–44
Principal interface
  getName method, 573
Principals (logins), 568
print method
  of DocPrintJob, 795
  of JTable, 616, 618
  of PrintWriter, 69–70, 833–834
Print dialog box, 773
  displaying page ranges in, 774, 783
  native, 774, 778
Print services, 792–795
  document flavors for, 793–794
  for images, 795
  stream, 796–799
print/PrintComponent.java, 779
print/PrintTestFrame.java, 778
Printable interface
  implementing, 772, 777
  objects, printing, 792
  print method, 773, 781, 783
Printer graphics context, 784
PrinterException, 773
PrinterJob class
  defaultPage method, 781
  getPrinterJob method, 773, 781
  pageDialog method, 776, 778, 781
  print method, 773–774, 782
  printDialog method, 773–774, 781
  setPageable method, 783
  setPrintable method, 782
printf function (C), 817
printf method (PrintWriter), 69–70, 385
printf/Printf1.c, 818
printf/Printf1.java, 818
printf/Printf1Test.java, 819
printf/Printf2.c, 824
printf/Printf2.java, 823
printf/Printf2Test.java, 823
printf/Printf3.c, 837
printf/Printf3.java, 836
printf/Printf3Test.java, 836
printf/Printf4.c, 846
printf/Printf4.java, 848
printf/Printf4Test.java, 848
Printing
  clipped areas, 775
  counting pages during, 775
  images, 772–782
  layout of, 783
  multipage documents, 782–784
  number of copies for, 799
  page orientation of, 730, 776
  paper sizes in, 775
  quality of, 802
  selecting settings for, 773
  starting, 546, 773
  text, 772–782
  using:
    banding for, 775
    transformations for, 784
Printing attributes, 799–807
  adding/retrieving, 802
  categories of, 801–802
  checking values of, 802
  hierarchy of, 800
PrintJob class (obsolete), 773
PrintJobAttribute interface, 799
  printing attributes of, 803–806
PrintJobAttributeSet interface, 800
println method
  of PrintWriter, 69–70
  of System.out, 414–415
PrintQuality class, 802
PrintRequestAttribute interface, 799
  printing attributes of, 803–806
PrintRequestAttributeSet interface, 773, 800
PrintService
  createPrintJob method, 793, 795
  getAttributes method, 807
  getName method, 793
printService/PrintServiceTest.java, 797
PrintServiceAttribute interface, 799
  printing attributes of, 803–806
PrintServiceAttributeSet interface, 800
PrintStream class, 69
PrintWriter class, 64, 68–70
  checkError method, 69–70
  constructor, 70
  print method, 69–70, 833–834
  printf method, 69–70, 385
  println method, 69–70
Private keys, 587–599, 608
PrivilegedAction interface, 575
  implementing, 568
  run method, 568, 572
PrivilegedActionException interface, 568
  run method, 572
processAnnotations method
  (ActionListenerInstaller), 466
Processing instructions (XML), 159
Processing tools, 463
Processor interface, 485
Programmer’s Day, 359
Programs. See Applications
Properties class, 154
Properties, generated automatically, 489
Property files, 154
  character encoding of, 419
  event handlers in, 447
  for resources bundles, 417–418
  for string resources, 417
  for strings, 418–419
  no passwords in, 279
Property permissions, 558
@Property annotation, 489
PropertyPermission class, 554
Protection domains, 549
ProtectionDomain class
  constructor, 551
  getCodeSource, implies methods, 551
  provides keyword, 524
Proxy objects, 466
Public certificates, keystore for, 599
PUBLIC identifier (DTD), 208
Public key ciphers, 587–594, 608–612
  performance of, 608
Public Key Cryptography Standard (PKCS)
  #5, 601
Pull parsers, 202
PushbackInputStream class, 65
  constructor, 67
  unread method, 67
put method
  of Bindings, 443
  of ByteBuffer, 131
  of CharBuffer, 132
  of ScriptEngine, 443
  of ScriptEngineManager, 443
PUT method (HttpRequest.Builder), 278
putClientProperty method (JComboBox), 658, 663
putNextEntry method (ZipOutputStream), 85–86
putXXX methods (ByteBuffer), 127, 131

Q
\0, in regular expressions, 138
QBE (query by example) tools, 289
QuadCurve2D class, 697, 699, 703
QuadCurve2D.Double class, 697–698, 713
QuadCurve2D.Float class, 697
Quadratic curves, 702–703
quadTo method (Path2D.Float), 703, 713
Qualified exports, 521
Qualified names, 195
Quantifiers, 141
Queries (databases), 290–292
  by example, 289
    executing, 300, 310–322
    multiple, 303
  populating row sets with results of, 331
  preparing, 311–317
  returning multiple results, 321–322
query/QueryTest.java, 313
"", entity reference, 158
quoteReplacement method (Matcher), 150

R
R programming language, 440, 448
\r line feed character, 69, 164, 414
in e-mails, 279
\r, \R, in regular expressions, 138, 140
Race conditions, 49
Random class, 602
methods of, 44, 47, 51
Random numbers, streams of, 5, 12, 44,
Random-access files, 80–85
randomAccess/RandomAccessTest.java, 83
RandomAccessFile class, 80–85, 125
constructor, 84
getChannel method, 130
getFilePointer method, 81, 84
length method, 81, 85
seek method, 80, 85
Randomness, 602
range, rangeClosed methods (XxxStream), 43, 46
Ranges, converting to streams, 49
Raster class
dataElements method, 758, 762
getPixel, getPixels methods, 757, 762
Raster images, 744–772
constructing from pixels, 756–763
filtering, 763–772
readers/writers for, 745–755
rasterImage/RasterImageFrame.java, 760
read method
of CipherInputStream, 608
of FileInputStream, 56
of ImageIO, 745, 753
of ImageReader, 754
of InputStream, 56–58
of ZipInputStream, 85
read/config.dtd, 187
read/config.xml, 187
read/config.xsd, 187
read/XMLReadTest.java, 183
Readable interface, 59
read method, 61, 63
ReadableByteChannel interface, 245
readAllBytes method (InputStream), 56, 58
readAllXXX methods (Files), 113
readAttributes method (Files), 117
readBoolean method (DataInput), 79
readChar method (DataInput), 79, 81
readDouble method (DataInput), 79, 89, 101
Reader class, 56, 60
read method, 59
READER class (DocFlavor), 794
readExternal method (Externalizable), 101–102
readFixedString method (DataI0), 81–82
readFloat method (DataInput), 79
readInt method (DataInput), 79, 81, 89
readLine method (Console), 71
readLong method (DataInput), 79
readNBytes method (InputStream), 56, 58
readObject method
of Date, 101
of ObjectInputStream, 89, 93, 101
ReadOnlyBufferException, 125
readResolve method (Serializable), 103
readShort method (DataInput), 79
readThumbnail method (ImageReader), 754
readUTF method (DataInput), 79–80
REAL data type (SQL), 293, 348
Receiver parameters, 478
Rectangle2D class, 697, 699
Rectangle2D.Double, Rectangle2D.Float classes,
697
RectangularShape class, 697
Redirects, of URLs, 266–267
reduce method (Stream), 41–43
reducing method (Collectors), 37
Reductions, 15, 41–43
ref attribute (XML Schema), 181
Reflection
accessing:
private members, 511–514, 546
protected methods, 532
constructing:
class trees, 678
static field names, 396
enumerating fields from a variable, 687
ReflectPermission class, 556
regex/RegexTest.java, 144
regexFilter method (RowFilter), 629, 638
Registry editor, 855, 860
Registry keys, 856–858
Regular expressions, 137–151
eapes in, 73, 141
filtering, 629
grouping in, 142–144
in DTDs, 173
predefined character classes in, 138–141
quantifiers in, 141
replacing all matches with, 148
relative method (ResultSet), 324, 328
Relativization, of an absolute URL, 253
relativize method (Path), 110–111
releaseSavepoint method (Connection), 345, 347
ReleaseStringChars function (C), 822
ReleaseStringUTFChars function (C), 821–822, 824
ReleaseXxxArrayElements functions (C), 842–843
reload method (DefaultTreeModel), 666, 671
remaining method (Buffer), 133–134
remove method (AttributeSet), 807
removeCellEditorListener method (CellEditor), 652
removeColumn method (JTable), 630, 636
removeNodeFromParent method (DefaultTreeModel), 665, 671
removeTreeModelListener method (TreeModel), 685, 693
RenderableImage interface, 792
Rendering (Swing)
cells, 639–652
columns, 622
headers, 641
nodes, 674–676
Rendering hints, 694
Rendering pipeline, 694–696
Renjin project, 440, 449
@Repeatable annotation, 479
replaceXxx methods (Matcher), 148, 150
required keyword, 568
@REQUIRED attribute (DTD), 175
requires keyword, 504, 506, 508, 510, 515, 519–521
requisite keyword, 568
RescaleOp class, 764, 771
Rescaling operation, 764
reset method
of Buffer, 133–134
of InputStream, 58
of MessageDigest, 587
resolve, resolveSibling methods (Path), 110–111
resolveEntity method (EntityResolver), 172, 178
Resolving
classes, 530
relative URLs, 253
Resource bundles, 417–421
loading, 419
locating, 417–418
lookup tables for, 420
naming, 419
searching for, 419
Resource editor, 417
@Resource annotation, 350, 479, 481
ResourceBundle class
extending, 419, 421
getBundle method, 418–421
getKeys method, 421
getObject method, 420–421
getString method, 419, 421
getStringArray method, 421
handleGetObject method, 421
ResourceBundle.Control class
c getCandidateLocales method, 418
Resources
annotations for managing, 480
hierarchy of, 418
in JAR files, 514
injection, 481
@Resources annotation, 479
Response headers, 256–257
Response page, 262
ResultSet interface, 220–221, 349
Result sets (databases)
accessing columns in, 300
analyzing, 300
closing, 303
for multiple queries, 303
iterating over rows in, 322
metadata for, 335
numbering rows in, 325
order of rows in, 300
retrieving multiple, 321–322
scrollable, 323–325
updatable, 322, 325–329
results method (Matcher), 146, 150
ResultSet interface, 330
absolute method, 325, 328
beforeFirst, afterLast methods, 325, 328
cancelRowUpdates method, 326, 329
ResultSet interface (continued)
close method, 303
concurrency values, 323, 325, 327, 329
deleteRow method, 327, 329
findColumn method, 302
first, last methods, 325, 328
getBlob, getClob methods, 317–318
getConcurrency method, 324–325, 328
getDate, getDouble, getInt, getObject, getString methods, 300, 302
getRow method, 325, 328
type values, 323, 325, 327, 329
updateObject method, 302
insertRow method, 326, 329
isClosed method, 303
isFirst, isLast, isBeforeFirst, isAfterLast methods, 325, 328
iteration protocol, 300
moveToCurrentRow method, 326, 329
moveToInsertRow method, 326, 328
next method, 300, 302, 322
previous method, 324, 328
relative method, 324, 328
updateObject method, 302
updateXxx methods, 326, 329
ResultSetMetaData interface, 335
cached, 330–335
Row sets (databases), 329–333
caching, 330–335
iterating through, 325
modifying, 330
page size of, 331
Rows (databases), 287
deleting/inserting, 327
interacting with, 325
order of, in result set, 300
Rows (Swing), 615–626
filtering, 628–630
height of, 625
hiding, 630
margins of, 625
position, in a node, 665
resizing, 625
Rows (Swing) (continued)
sorting, 615, 626
upgrading, 326
RowSet interface, 329–332
methods of, 331–332
Rivest, Ronald, 584
Role-based authentication, 573
rollback method (Connection), 345–347
Root certificate, 599
Root component (file system), 109
Root environment (XML), 157
... referencing schemas in, 179
Root node (Swing), 652–660
handles for, 659–660
separating children of, 659
Rotation, 727–728
and interpolating pixels, 764
with center point, 729
Round cap, 716
Round join, 716–717
Rounded rectangles, 699–700
RowFilter interface, 486
RowFilter class, 628–630
RowFilter.Entry class, 629
ROWID data type (SQL), 348–349
RowId interface, 349
Rows (databases), 287
deleting/inserting, 327
interacting with, 325
order of, in result set, 300
retrieving, 349
selecting, 290
upgrading, 326
Rows (Swing), 615–626
filtering, 628–630
height of, 625
hiding, 630
margins of, 625
position, in a node, 665
resizing, 625
selecting, 615, 626
sorting, 616, 627–628
RowSet interface, 329–332
methods of, 331–332
RGB color model, 735, 758
RFC 821 standard, 279
RFC 822 standard, 372
RFC 1123 standard, 372
RFC 2396 standard, 253
RFC 2616 standard, 255
RFC 2911 standard, 806
...
RowSetFactory interface
   createXxxRowSet methods, 330, 333
RowSetProvider class
   newFactory method, 330, 333
RSA algorithm, 588, 609
RSA Security, 601
rsa/RSAjTest.java, 610
rt.jar file, 527, 531–532
Ruby programming language, 440
run method
   of PrivilegedAction, 568, 572
   of PrivilegedExceptionAction, 572
   of Tool, 461
Runnable interface, 240
Runtime class
   exit method, 546
runtimeAnnotations/ ActionListenerFor.java, 470
runtimeAnnotations/ ActionListenerInstaller.java, 467
RuntimePermission class, 555

S
S (short), type code, 95, 831
\s, \S, in regular expressions, 139
@SafeVarargs annotation, 479
Sample values, 756
Sandbox, 547–551
Save points (databases), 345
Savepoint interface
   getSavepointXxx methods, 347
SAX (Simple API for XML) parser, 159, 197–202
   activating namespace processing in, 199
sax/SAXTest.java, 199
SAXParseException class
   getXXNumber methods, 178
SAXParser class
   parse method, 198, 201
SAXParserFactory class
   is/setNamespaceAware methods, 201
   is/setValidating methods, 201
   newInstance, newSAXParser methods, 198, 201
   setFeature method, 199
SAXResult class, 221
SAXSource class, 220–221
   constructor, 225
Scalar functions, 319–320
scale method (Graphics2D), 728, 732
Scaling, 727–728
Scanner class, 70
   constructor, 245
   findAll method, 146, 151
   tokens method, 6, 10
   useLocale method, 385, 389
Scheduling applications
   and time zones, 359, 367
   computing dates for, 364–365
   schema element (XML Schema), 182
Schemas, 343
Script engines, 440–441
   adding variable bindings to, 442
   implementing Java interfaces, 445
   invoking, 441
   invoking functions in, 444–446
script/ScriptTest.java, 449
ScriptContext interface, 443
   getXxx/setXxx methods of, 444
ScriptEngine interface
   createBindings method, 443
   eval method, 441–443
   get, put methods, 443
   getApplicationContext method, 444
ScriptEngineFactory interface
   getExtensions method, 441
   getMethodCallSyntax method, 445
   getMimeTypes method, 441
   getNames method, 441
ScriptEngineManager class
   get, put methods, 443
   getEngineXxx methods, 441
Scripting languages, 440–452
   advantages of, 440
   supported, 440
Scripts
   accessing classes in, 446
   compiling, 446
   executing, 442, 447
   invoking, 441
   redirecting I/O of, 444
   using Java method call syntax in, 445
Scroll pane (Swing)
   with tables, 615
   with trees, 666, 668
scrollPathToVisible method (JTree), 666, 670
Secret key, generating, 603
SecretKey interface, 602
SecretKeySpec class, 607
Secure random generator, 603
SecureRandom class
  setSeed method, 602
Securing Java (McGraw/Felten), 550
Security
  bytecode verification, 541–545
  class loaders, 530–545
  code signing, 597–599
  different levels of, 583
digital signatures, 582–599
  encryption, 599–612
  user authentication, 566–582
Security managers, 546–566
Security policy, 547
SecurityException class
  checkExit method, 546, 549
  checkPermission method, 549–550, 559–560
  checkRead method, 550
SecurityPermission class, 556
“Seek forward only” mode (ImageInputStream), 747
seek method (RandomAccessFile), 80, 85
select attribute (XSLT), 219
SELECT statement (SQL), 290–291
executing, 300
for LOBs, 317
multiple, in a query, 321
not supported in batch updates, 346
Selection models, 626
send method (HttpClient), 277
sendAsync method (HttpClient), 273, 277
separator constant (File), 64
Separators (file system), 64, 109
sequence element (XML Schema), 181
Sequences, producing, 5
Serial numbers, 90–91
  vs. memory addresses, 92
serialClone/SerialCloneTest.java, 106
Serializable class, 106
Serializable interface, 89, 95, 483
  readResolve method, 103
@Serializable annotation, 483
SerializablePermission class, 556
Serialization, 88–108
  cloning with, 106–108
  file format for, 93–100
  modifying default, 100–102
  of singletons, 102–103
  serial numbers for, 90–91
serialver program, 104
serialVersionUID constant, 104
server/EchoServer.java, 237
Servers
  accessing, 251–270
  connecting clients to, 230–232
  implementing, 236–251
  invoking programs, 261
Server-side programs, 261–270
  redirecting URLs in, 266–267
ServerSocket class, 236–251
  accept method, 236, 239–240
  close method, 239
  constructor, 239
Service loading, 522–524
Service provider interfaces, 746
SERVICE_FORMATTED class (DocFlavor), 794
ServiceLoader class, 522
Servlets, 261, 457–463
Session class
  setDebug method, 280
Set interface
  containsAll, equals methods, 561
set/Item.java, 494
set/ItemTest.java, 495
setAllowsChildren, setAsksAllowsChildren methods
  (DefaultMutableTreeNode), 661, 663
setAllowUserInteraction method (URLConnection), 254
setAttribute, setAttributeNS methods (Element), 207, 209
setAutoCommit method (Connection), 347
setAutoCreateRowSorter method (JTable), 616, 618, 627
setAutoResizeMode method (JTable), 625, 635
setBinaryStream method (Blob), 319
SetBooleanArrayRegion function (C), 842
SetBooleanField function (C), 830
SetByteArrayRegion function (C), 842, 844, 857
SetCharField function (C), 830
setCellEditor method (TableColumn), 642, 651
setCellRenderer method (TableColumn), 651
setCellSelectionEnabled method (JTable), 626, 636
setCharacterStream method (Clob), 319
SetCharArrayRegion function (C), 842, 844
SetCharField function (C), 830
setClip method (Graphics), 733–734, 775
setClosedIcon method (DefaultTreeCellRenderer), 676
setColumnSelectionAllowed method (JTable), 626, 636
setCommand method (RowSet), 331–332
setComparator method (DefaultRowSorter), 628, 637
setComposite method (Graphics2D), 695, 738, 744
setConnectTimeout method (URLConnection), 254, 260
setContentHandler method (XMLReader), 226
setContextClassLoader method (Thread), 533, 541
setCurrency method (NumberFormat), 393
setDataElements method (WritableRaster), 759, 762
setDate method (PreparedStatement), 311, 317
setDebug method (Session), 280
setDecomposition method (Collator), 408
setDefault method of CookieHandler, 267
of Locale, 384, 386
setDefaultNamespace method (XMLStreamWriter), 214
setDefaultRenderer method (JTable), 640
setDoInput method (URLConnection), 254–255, 259
setDoOutput method (URLConnection), 254–255, 259, 264, 266
setDouble method (PreparedStatement), 311, 317
SetDoubleArrayRegion function (C), 842, 844
SetDoubleField function (C), 826, 830
setEntityResolver method (DocumentBuilder), 172, 177
setErrorHandler method (DocumentBuilder), 177
setErrorWriter method (ScriptContext), 444
setFeature method (SAXParserFactory), 199
setFillsViewportHeight method (JTable), 618
SetFloatArrayRegion function (C), 842, 844
SetFloatField function (C), 830
setMaxWidth method (TableColumn), 624, 637
setName method (NameCallback), 581
of SAXParserFactory, 201
SetObjectArrayElement function (C), 840, 843, 845
SetObjectField function (C), 826, 830
setMaxWidth method (TableColumn), 624, 637
setOpenIcon method (DefaultTreeCellRenderer), 676
setOutput method (ImageWriter), 755
setOutputProperty method (Transformer), 209
setPreferredWidth method (TableColumn), 624, 637
setScale method (ZipOutputStream), 254, 255, 260
setScaleXxxDigits, setMinimumXxxDigits methods (NumberFormat), 392
setScaleInverted method (XMLStreamWriter), 214
setScaleXxxYxxDigits method (NumberFormat), 392
setPassword method of PasswordCallback, 582
of RowSet, 331–332
setProperty method (XMLInputFactory), 203–204
setPrefix method (XMLStreamWriter), 214
setPreferredWidth method (TableColumn), 624, 637
setPrefix method (XMLInputFactory), 203–204
setReader method (ScriptContext), 444
setReadTimeout method (URLConnection), 254, 260
setRenderingHint, setRenderingHints methods (Graphics2D), 694
setRequestProperty method (URLConnection), 254–255, 260
setResizable method (TableColumn), 624, 637
setRootVisible method (JTree), 660, 662
setRowFilter method (DefaultRowSorter), 629, 637
setRowHeight, setRowMargin methods (JTable), 625, 635
setRowSelectionAllowed method (JTable), 626, 636
setRowSorter method (JTable), 627, 636
Sets, comparing, 561
setSavepoint method (Connection), 347
setSecurityManager method (System), 553
setSeed method (SecureRandom), 602
setSelectionMode method (ListSelectionModel), 626, 637
SetShortArrayRegion function (C), 842, 844
SetShortField function (C), 830
setShowsRootHandles method (JTree), 659, 662
setSize method (ZipEntry), 87
setSortable method (DefaultRowSorter), 627, 637
setSoTimeout method (Socket), 232–233
SetStaticXxxField functions (C), 829–830
setStrength method (Collator), 408
setString method (PreparedStatement), 311, 317
setStringConverter method (TableRowSorter), 637
setStroke method (Graphics2D), 694, 715, 724
setSubject method (MimeMessage), 279
setTableName method (CachedRowSet), 332–333
setText method (MimeMessage), 279
setToXxx methods (AffineTransform), 730, 732
setTransform method (Graphics2D), 730, 732
setURL method (RowSet), 331–332
setUseCaches method (URLConnection), 254
setUsername method (RowSet), 331–332
setUserObject method (MutableTreeNode), 654, 662
setValidating method
  of DocumentBuilderFactory, 176, 179
  of SAXParserFactory, 201
setValue method (Win32RegKey), 857–858
setValueAt method (TableModel), 622, 645
setWidth method (TableColumn), 624, 637
setWriter method (ScriptContext), 444
SGML (Standard Generalized Markup Language), 155
SHA-1 algorithm, 94, 583
Shape interface, 697, 715
shape/ShapeTest.java, 705
ShapeMaker class
  getPointCount, makeShape methods, 704
ShapePanel class, 704
Shapes
  clipping, 694, 733–735
  combining, 695, 714–715
  control points of, 704
  drawing, 694–697
  filling, 694–695, 724
  rendering, 696
  transforming, 694
Shared libraries, 816, 854
shear method (Graphics2D), 728, 732
Shear, 727–728
Shift-JIS standard, 76
short type
  printing, 69
  streams of, 43
  type code for, 95, 831
  vs. C types, 817
  writing in binary format, 78
ShortBuffer class, 132
ShortLookupTable class, 765, 772
shouldSelectCell method (CellEditor), 644, 652
shutdownXXX methods (Socket), 244
Side files, 484
Signatures, 587–589
  generating, 832
  mangling, 831–832
Simple types, 179
SimpleDoc class, 793, 795
SimpleFileVisitor class, 121
  visitFile, visitFailed methods, 121, 123
  xxxVisitDirectory methods, 123
SimpleJavaFileObject class, 455
  getCharContent, openOutputStream methods, 463
SimpleLoginModule class
  checkLogin, initialize methods, 573
SimpleScriptContext class, 443
simpleType element (XML Schema), 180
Single value annotations, 473
Singletons, serializing, 102–103
size method
  of BasicFileAttributes, 118
of Files, 116–117
skip method
  of InputStream, 58
  of Stream, 12
skipBytes method (DataInput), 80
SMALLINT data type (SQL), 293, 348
SMTP (Simple Mail Transport Protocol), 278–281
SOAP (Simple Object Access Protocol), 501
Socket class
  connect method, 233
  constructor, 232–233
  getInputStream method, 231–232, 236
  getOutputStream method, 232, 236
  isClosed, isConnected methods, 233
  isXxxShutdown, shutdownXxx methods, 244
  setSoTimeout method, 232–233
Socket permissions, 558
Socket/SocketTest.java, 231
SocketChannel class, 244
  open method, 231, 546
timeouts, 232–233
SocketTimeoutException, 232, 260
sort method (Collections), 402
sorted method (Stream), 14–15
Source files
  character encoding of, 416
  reading from memory, 454
Source interface, 220, 349
Source-level annotations, 484–489
Space. See Whitespace
SPARC processor, big-endian order in, 78
split method
  of Pattern, 147, 149
  of String, 72, 147
Split packages, 510
splitAsStream method (Pattern), 6, 10, 147
splitter method (Iterable), 10
Splitters class
  spliteratorUnknownSize method, 6, 9
SplittableRandom class, methods of, 51
sprint, sprintf functions (C), 823
SQL (Structured Query Language), 287–293
  changing data inside databases, 292
  commands in, 295
  data types in, 293, 348–349
  equality testing in, 291
  escapes in, 319–321
  exceptions in, 304–306
  executing statements in, 299–303
  keywords in, 290
  reading instructions from a file, 306
  strings in, 291
  vs. Java, 313
  warnings in, 304
  wildcards in, 291
SQLException class, 304–306, 324
  and rolling back, 345
  and save points, 347
  getXxx, iterator methods, 304–305
SQLPermission class, 557
SQLWarning class, 304, 324
  getNextWarning method, 306
SQLXML data type (SQL), 348–349
Square cap, 716
Square root, computing, 21
SQuirreL program, 336
SRC, SRC_Xxx composition rules, 737–739
src.jar file, 850
sRGB standard, 758
Standard extensions, 531
StandardCharsets class, 77
StandardJavaFileManager interface, 453–455
  getJavaFileObjectsFromXxx methods, 462
start method
  of Matcher, 143, 145, 149–150
  of MatchResult, 146, 150
startDocument method (ContentHandler), 201
startElement method (ContentHandler), 197–201
Stateless operations, 49
Statement interface, 299–303
  addBatch method, 346–347
  close, closeOnCompletion methods, 302–303
  execute method, 301, 307, 321–322
  executeBatch method, 346–347
  executeLargeBatch method, 347
  executeQuery method, 300–301, 324
  executeUpdate method, 299, 301, 322, 345
  getMoreResults method, 321–322
  getResultSet method, 302
Statement interface (continued)
- getUpdateCount method, 302, 321
- getWarnings method, 306
- isClosed method, 302
- RETURN_GENERATED_KEYS field, 322
- using for multiple queries, 303

Statements (databases)
- closing, 303
- complex, 313
- concurrently open, 303
- executing, 299–303
- grouping into transactions, 344–349
- in batch updates, 346
- multiple, 303
- prepared, 311–317
- truncations in, 305

Static fields, in native code, 829–830
Static initialization blocks, 814
Static methods, calling from native code, 834–835

statusCode method (HttpResponse), 272, 278

StAX parser, 202–206, 210–215
- namespace processing in, 203
- no indented output in, 211
- stax/StAXTest.java, 203
- StAXSource class, 220
- stopCellEditing method (CellEditor), 644–645, 652

Stored procedures, 319–320

Stream interface
- allMatch, anyMatch methods, 16
- collect method, 25–30, 42–43
- concat method, 13
- count method, 3–4, 15
- distinct method, 14–15, 50
- dropWhile method, 13
- empty method, 5, 9
- filter method, 3–11, 15
- findAny method, 16
- findFirst method, 15–16
- flatMap method, 12
- forEach method, 25, 28
- forEachOrdered method, 25
- generate method, 5, 9, 43
- iterate method, 5, 9, 14, 43
- iteror method, 25
- limit method, 12, 50
- map method, 11
- mapToInt method, 42
- max, min methods, 15–16
- noneMatch method, 16
- of method, 5, 9
- ofNullable method, 6, 9, 23
- peek method, 14–15
- reduce method, 41–43
- skip method, 12
- sorted method, 14–15
- takeWhile method, 13
- toArray method, 25, 28
- unordered method, 50

stream method
- of Arrays, 5, 9, 43
- of Collection, 2–3, 5
- of Optional, 22–25
- of StreamSupport, 6, 10

Streaming parsers, 159, 196–206
StreamPrintService class, 796
StreamPrintServiceFactory class, 796
- getPrintService method, 796
- lookupStreamPrintServiceFactories method, 796

StreamResult class, 210, 221

Streams, 1–7
- collecting elements of, 25–31
- computing values from, 41–43
- converting to/from arrays, 5, 25, 49
- creating, 5–10
- debugging, 14
- empty, 5, 15, 41
- encrypted, 607–608
- filtering, 22
- finite, 6
- flattening, 12, 22
- for print services, 796–799
- infinite, 3, 5, 12, 14
- input, 160
- intermediate operations for, 3
- noninterference of, 7
- of primitive type values, 42–44
- of random numbers, 44
- parallel, 2, 16, 25, 31, 35, 41, 48–53
- processed lazily, 3, 7, 14
- reductions of, 15
- removing duplicates from, 14
- returned by Files.lines, 50
- sorted, 14, 49
- splitting/combining, 12–13
- summarizing, 26, 44
- terminal operation for, 3, 15
transformations of, 11–12, 44
vs. collections, 3
streams/CountLongWords.java, 4
streams/CreatingStreams.java, 7
streams/PrimitiveTypeStreams.java, 44
StreamSource class, 220
constructor, 225
transform method, 221
StreamSupport class
stream method, 6, 10
String class, 62
cator, 402
format method, 385
split method, 72, 147
tolowerCase method, 11	trim method, 163, 388
STRING class (DocFlavor), 794
String parameters, 819–825
StringBuilder class, 62, 82
Strings
converting to code points, 11
encoding, 379, 416
fixed-size, I/O of, 81–82
in native code, 819–825
in SQL, 291
internationalizing, 417–418
ordering, 402
patterns for, 137–151
printing, 69
sorting, 402
splitting, 6
transforming to lower/uppercase, 11
writing in binary format, 78
StringSource class, 454
Stroke interface, 715
stroke/StrokeTest.java, 719
StrokePanel class, 719
Strokes, 694, 715–724
 dash patterns of, 717–718
 end cap styles of, 716–718
 join styles of, 716–717
 setting, 694
 thickness of, 716
Stylesheets (XSLT), 216–226
Subject class
doAs, doAsPrivileged methods, 568–569, 572
getPrincipals method, 572
Subjects (logins), 568
subSequence method (CharSequence), 63
subtract method (Area), 714–715
Subtraction operator, not associative, 41
Subtrees (Swing), 657, 674
 adding nodes to, 666
collapsed and expanded, 658
Suetonius, Gaius Tranquillus, 536
sufficient keyword, 568
sum, summaryStatistics methods (primitive streams), 44, 46–47
summarizingXxx methods (Collectors), 26, 29, 37
summingXxx methods (Collectors), 36, 40
SunJCE ciphers, 600
Superclasses, type use annotations in, 476
Supplier interface
get method, 10
@SupportedAnnotationTypes annotation, 485
SupportedValuesAttribute interface, 799
supportsBatchUpdates method (DatabaseMetaData), 346, 348
supportsResultSetXxx methods (DatabaseMetaData), 324, 329
@SuppressWarnings annotation, 479–480
SVG (Scalable Vector Graphics), 215–216
Swing, 613–693
generating dynamic code for, 457
tables, 613–652
trees, 652–693
Symmetric ciphers, 600–601
 performance of, 608
SyncProviderException interface, 332–333
System class
 console method, 415
 loadLibrary method, 814, 816
 setSecurityManager method, 553
System class loader, 531
SYSTEM identifier (DTD), 171, 208
System properties, in policy files, 558
System.err class, 69, 453
System.in class, 69
 and character encoding, 415
System.out class, 69, 453
 and character encoding, 415
println method, 414–415
T
t literal (SQL), 320
\t, in regular expressions, 138
Table cell renderers, 622, 639
Table index values, 626
Table models (Swing), 614, 618–622
updating after cells were edited, 645
table/TableTest.java, 616
TableCellEditor interface
getTableCellEditorComponent method, 643, 645, 651
implementing, 643, 645
tableCellRender/ColorTableCellEditor.java, 649
tableCellRender/ColorTableCellRenderer.java, 648
tableCellRender/PlanetTableModel.java, 647
tableCellRender/TableCellRenderFrame.java, 646
TableCellRenderer interface
getTableCellRendererComponent method, 639, 651
implementing, 639
TableCell models (Swing), 614, 618–622
TableColumn class, 624–625, 630
constructor, 637
setCellEditor method, 642, 651
setCellRenderer method, 651
setHeaderXXX methods, 641, 651
setResizable, setWidth, setXXXWidth methods, 624, 637
TableColumnModel interface, 623
getColumn method, 636
TableModel interface, 627
getColumn method, 618
getColumnClass method, 622, 635
getColumnCount method, 619, 622
getValueAt method, 619, 622
getXXXCount methods, 618–619, 622
implementing, 618
getCellEditable method, 622, 641
setCellEditable method, 622, 645
tableModel/InvestmentTable.java, 620
tableRowColumn/PlanetTableFrame.java, 631
TableRowSorter class, 627
setStringConverter method, 637
Tables (databases), 287
changing data in, 292
creating, 292
duplication of data in, 289
inspecting, 289
metadata for, 334
multiple, selecting data from, 291
removing, 297
Tables (Swing), 613–652
cells in:
editing, 641–642
rendering, 639
selecting, 626
columns in:
accessing, 623
adding, 630
hiding, 630–639
naming, 619
rearranging, 615
rendering, 622
resizing, 615–616, 624–625
selecting, 626
constructing, 615, 619
headers in, 615
rendering, 641
printing, 616
relationship between classes of, 624
rows in:
filtering, 628–630
hiding, 630
margins of, 625
resizing, 625
selecting, 615, 626
sorting, 616, 627–628
scrolling, 615
TableStringConverter class
toString method, 628, 637
takeWhile method (Stream), 13
@Target annotation, 465, 479, 481
TCP (Transmission Control Protocol), 232
telnet activating/connecting, 228
several windows communicating simultaneously, 240–241
template element (XSLT), 218
Temporal interface, 357
TemporalAdjuster interface, 364
TemporalAdjusters class, 364–365
dayOfWeekInMonth, firstDayOfWeek, lastXXX methods, 365
next, nextOrDefault methods, 365
ofDateAdjuster method, 364
previous, previousOrSame methods, 365
TemporalAmount interface, 357–358, 362–363
test/TestDB.java, 298
@Test annotation, 464
@TestCase, @TestCases annotations, 484
Text, 68
  encoding of, 75–77
  generating from XML files, 219–221
  output, 68–70
  printing, 772–782, 792
  reading, 70–72
  saving objects in, 72–75
  transmitting through sockets, 236–251
vs. binary data, 68

Text fields
  editing, 641
  validating input in, 560–566
Text files, encoding of, 414–415
Text nodes (XML)
  constructing, 207
  retrieving from XML, 163
TextCallbackHandler class, 574
  textField/TextFileTest.java, 73
TextLayout class
  getXXX methods, 735
TextStyle enumeration, 396
TextSyntax class, 802
TexturePaint class, 725–726
this keyword, 826
  annotating, 477–478
Thread class
  get/setContextClassLoader methods, 533, 541
  threaded/ThreadedEchoServer.java, 241
ThreadedEchoHandler class, 239–243
Threads
  blocking, 57, 244–251
  executing scripts in, 442
  Internet connections with, 239–243
  race conditions in, 49
  referencing class loaders in, 533–534
Three-tier model, 286–287
Throw, ThrowNew functions (C), 844–845, 849
Throwable class, 844
Thumbnails, 748
Time
  current, 354
  formatting, 371–376, 394–401
  instances of, 365
  literals for, 319
  local, 365–366
  measuring, 355
  parsing, 373
  zoned, 367–371, 394
  Time class, 376
    valueOf method, 377
    Time of day service, 228
    TIME, TIMESTAMP data types (SQL), 293, 320, 348
    timeline/TimeLine.java, 356
    Timeouts, 232–233
    Timestamp class, 376
toInstant, valueOf methods, 377
Timestamps, 371
  using instants as, 355
TimeZone class
  getTimeZone, toZoneId methods, 377
toAbsolutePath method (Path), 110–111
toArray method
  of AttributeSet, 807
  of primitive streams, 44, 46–47
  of Stream, 25
  of streams, 28
toCollection method (Collectors), 25, 29
toConcurrentMap method (Collectors), 31, 34
toDays method (Duration), 355, 358
toFile method (Path), 111
toFormat method (DateTimeFormatter), 373, 377
toHours method (Duration), 355, 358
toInstant method
  of Date, 376–377
  of FileTime, 377
  of Timestamp, 377
  of ZonedDateTime, 367, 371
tokens method (Scanner), 6, 10
toLanguageTag method (Locale), 383, 386
toList method (Collectors), 25, 29
toLocalXXX methods
  of LocalXXX, 377
  of ZonedDateTime, 371
toLowerCase method (String), 11
toMap method (Collectors), 30–34
toMillis, toMinutes, toNanos methods (Duration), 355, 358
toNanoOfDay method (LocalTime), 366
Tool interface
  run method, 461
ToolProvider class
  getSystemJavaCompiler method, 453
tools.jar file, 527
toPath method (File), 111–112
Top-level windows, opening, 546
toSecondOfDay method (LocalTime), 366
toSeconds method (Duration), 355, 358
toSet method (Collectors), 25, 29, 36
toString method
implementing with annotations, 486–489 of Annotation, 472
of CharSequence, 63
of Currency, 394
of Locale, 386
of ToStringConverter, 628, 637
of Variable, 687
toUnmodifiableList method (Collectors), 29
toUnmodifiableMap method (Collectors), 34
toUnmodifiableSet method (Collectors), 29
toZonedDateTime method (GregorianCalendar), 376–377
toZoneId method (TimeZone), 377
Transactions, 344–349
committing, 344
error handling in, 346
rolling back, 344
transferTo method (InputStream), 58
transform method
of Graphics2D, 695, 730, 732
of StreamSource, 221
of Transformer, 209, 220
transform/makehtml.xsl, 221
transform/makeprop.xsl, 222
transform/TransformTest.java, 223
Transformations, 694, 727–732
affine, 730, 764
composing, 728–729
fundamental types of, 727–728
matrices for, 729–730
order of, 728
setting, 695
using for printing, 784
Transformer class
setOutputProperty method, 209
transform method, 209, 220
TransformerFactory class
newInstance method, 209
newTransformer method, 209, 225
transient keyword, 100
transitive keyword, 519–521
translate method (Graphics2D), 728, 732, 784
Translation, 727–728
Transparency, 735–744
Traversal order, 672
Tree events, 677–684
Tree models
constructing, 654, 685
custom, 684–693
default, 654
Tree parsers, 159
Tree paths, 663–671
constructing, 666, 673
Tree selection listeners, 677
tree/SimpleTreeFrame.java, 656
TreeCellRenderer interface, 674–676
getTreeCellRendererComponent method, 675–676
implementing, 674
treeEdit/TreeEditFrame.java, 668
TreeMap class, 31
TreeModel interface, 654, 664
add/removeTreeModelListener method, 685, 693
getChild, getChildCount, getRoot methods, 685–687, 692
getIndexOfChild method, 685, 692
implementing, 654
isLeaf method, 663, 685, 693
valueForPathChanged method, 686, 693
treeModel/ObjectInspectorFrame.java, 688
treeModel/ObjectTreeModel.java, 689
treeModel/Variable.java, 691
TreeModelEvent class, 693
TreeModelListener interface, 685
treeNodeXxx, treeStructureChanged methods, 685, 693
TreeNode interface, 654, 664
children, getChildXxx methods, 671
getAllowsChild method, 662
getParent method, 671, 673
isLeaf method, 661–662
TreePath class, 665
gGetLastPathComponent method, 665, 671
treeRender/ClassNameTreeCellRenderer.java, 683
treeRender/ClassTreeFrame.java, 679
Trees (Swing), 652–693
adding listeners to, 677
background color for, 674
connecting lines in, 658–659
displaying, 654–671
editing, 663–671, 686
handles in, 657, 659, 674
hierarchy of classes for, 655
indexes in, 665
infinite, 688
leaves in, 652, 660–662, 674, 685
nodes in, 652, 662, 674, 685
paired with other components, 677
rendering, 674–676
scrolling to newly added nodes, 666, 668
structure of, 652
subtrees in, 657–658
traversals for, 672–673
view of, 665–666
with horizontal lines, 659

TreeSelectionEvent class
getPath method, 684
getPaths method, 678, 684

TreeSelectionListener interface
implementing, 677–684
valueChanged method, 677, 679, 684

TreeSelectionModel interface, 677
trim method (String), 163, 388

Troubleshooting, See Debugging

True Odds: How Risks Affect Your Everyday Life (Walsh), 583

tryLock method (FileChannel), 135–136
try-with-resources statement, 61, 120
closing files with, 118
for database connections, 303
with locks, 135
ts literal (SQL), 320

Type bounds, type use annotations in, 476
Type definitions, 179
anonymous, 181
nesting, 181

Type parameters, annotating, 475
Type use annotations in, 476

TYPE_BICUBIC, TYPE_BILINEAR fields
(AffineTransformOp), 764, 771
TYPE_BYTE_GRAY field (BufferedImage), 760–761
TYPE_BYTE_INDEXED field (BufferedImage), 761
TYPE_INT_ARGB field (BufferedImage), 756–757, 761
TYPE_NEAREST_NEIGHBOR field (AffineTransformOp), 764, 771

TypeElement interface, 485–486
Types. See Data types
Typesafe enumerations, 102–103

U
\u, in regular expressions, 138
UDP (User Datagram Protocol), 232
UIDManager class, 640
Unicode standard, 43
and input/output streams, 56
and native code, 820
character order in, 402
converting to binary data, 68
in property files, 419
in regular expressions, 141
normalization forms in, 403
using for all strings, 379

Units of measurement, 158
UNIX operating system
authentication in, 566
authentications in, 567
file names starting with a period in, 552
line feed in, 69, 414
paths in, 109
specifying locales in, 384
UnixLoginModule class, 568
UnixNumericGroupPrincipal class, 568
UnixPrincipal class, 567–568
getName method, 567
UnknownHostException, 231
unordered method (BaseStream), 50, 53
Unparsed external entities, 176
unread method (PushbackInputStream), 67
UnsatisfiedLinkError, 811
until method (LocalDate), 359, 362

update method
of Cipher, 601, 603, 606–607
of MessageDigest, 585–586
UPDATE statement (SQL), 292, 312, 325
executing, 300–301, 317
in batch updates, 346
truncations in, 305
vs. methods of ResultSet, 327
updateObject method (ResultSet), 302
updateXxx methods (ResultSet), 326, 329

URI class, 271
getXxx methods, 253
no resource accessing with, 252
uri method (HttpRequest.Builder), 271–273, 277

URIs (Uniform Resource Identifiers), 252
absolute vs. relative, 252–253
base, 253
URIs (Uniform Resource Identifiers)  
(continued)
  hierarchical, 252
  namespace, 193–196
  opaque vs. nonopaque, 252
  schemes for, 252
  with HTTP, 271
URISyntax class, 802
URL class (DocFlavor), 794
URL class (java.lang.Object), 251–253, 271
accept schemes for, 252
openConnection method, 254, 259
openStream method, 251, 259
URLClassLoader class
  addURLs, getURLs methods, 532
  constructor, 541
  loadClass method, 532
URLConnection class, 251, 254–261, 271
  connect method, 254, 256, 260
  getConnectTimeout method, 260
  getContent method, 261
  getContentType, getContentType methods, 254, 257, 261, 266
  getDate, getExpiration, getLastModified methods, 254, 257, 261
  getDoInput, getDoOutput methods, 259
  getHeaderXxx methods, 254–256, 260
  getIfModifiedSince method, 260
  getInputStream method, 254, 261, 259
  getReadTimeout method, 260
  getRequestProperty method, 260
  setAllowUserInteraction method, 254
  setConnectTimeout method, 254, 260
  setDoInput method, 254–255, 259
  setDoOutput method, 254–255, 259, 264, 266
  setIfModifiedSince method, 254–255, 260
  setReadTimeout method, 254, 260
  setRequestProperty method, 254–255, 260
  setUseCaches method, 254
urlConnection/URLConnectionTest.java, 257
URLDecoder class
  decode method, 270
URLEncoder class
  encode method, 270
URLs (Uniform Resource Locators), 252
  attaching parameters to, 263
  connections via, 251
  encoding, 263
  for databases, 294
  for local files, 553
  for namespace identifiers, 194
  redirecting, 266–267
  relative vs. absolute, 599
  for DTDs, 171
URNs (Uniform Resource Names), 252
US Letter paper, 775
useLocale method (Scanner), 385, 389
User coordinates, 727
User objects, 654, 665
User-Agent request parameter, 267
Users
  authentication of, 566–582
  permissions for, 569
  preferences of, 135
uses keyword, 523–524
UTC (Coordinated Universal Time), 367
UTF-8 standard, 75–77, 78
  for text files, 414–415
  modified, 78–80, 416, 819–822
UTF-16 standard, 43, 68, 76, 78
  and native code, 819
  in regular expressions, 138
V
V (void), type code, 831
\v, \V, in regular expressions, 139
Validation, 169–188
  activating, 176
  adding to classes, 100
  value method (XPathEvaluationResult), 191, 193
  valueChanged method (TreeSelectionListener), 677, 679, 684
  valueForPathChanged method (TreeModel), 686, 693
  value-of element (XSLT), 219
  value0f method (date/time legacy classes), 377
VARCHAR data type (SQL), 293, 348
VarHandle class, 514
Variable class, 686
  toString method, 687
  Variable handles, 514
  VariableElement interface, 485
Variables
  annotating, 464, 476
binding, 442
fields of, 687
initializing, 542
scope of, 443
Variants, in locales, 381, 418
Vendor name, of a reader, 746
Verifier/VerifierTest.java, 545
Verifiers, 541–545
Version number, of a reader, 746
Versioning, 103–106
view/ViewDB.java, 336
visitFile, visitFileFailed methods of FileVisitor, 121
of SimpleFileVisitor, 121, 123
Visual representation, 286

W
\w, \W, in regular expressions, 139
walk method (File), 118
walkFileTree method (Files), 121–123
warning method (ErrorHandler), 177–178
Warnings
in SQL, 304
suppressing, 480
WBMP format, 745
WeakReference object, 688
Web applications, connection management in, 349–351
Web containers, 480
Web crawlers, 198
with SAX parser, 199
with STAX parser, 203
Web pages
dynamic, 457–463
separating applet class loaders for, 534
WebRowSet interface, 330
Weekends, 360
WHERE statement (SQL), 291
Whitespace
ignoring, while parsing, 163
in e-mail URIs, 263
in regular expressions, 139, 141
Wildcards, type use annotations in, 476
Wilde, Oscar, 380
win32reg/Win32RegKey.c, 862
win32reg/Win32RegKey.java, 860
win32reg/Win32RegKeyTest.java, 870
Win32RegKey class, 856, 859
get/setValue methods, 857–858
names method, 857
Win32RegKeyNameEnumeration class, 858–859
Windows operating system
activating telnet in, 228
authentication in, 566, 569
character encodings in, 413
classpath in, 294
compiling invocation API, 854
dynamic linking in, 850
glob syntax in, 121
line feed in, 69, 414
paths in, 64, 109
permissions in, 558
registry, accessing from native code, 855–871
resources in, 417
using Microsoft compiler, 813–814
Windows look-and-feel, trees in, 658
with method (Temporal), 364–365
withLocale method (DateTimeFormatter), 372, 376, 401
withXXX methods
of LocalDate, 362
of LocalTime, 366
of Period, 363
of ZonedDateTime, 370
WordCheckPermission class, 560–566
Words, in regular expressions, 139
Working directory, 64
wrap method (ByteBuffer), 132–133
WritableByteChannel interface, 245
WritableRaster class, 756
setDataElements method, 759, 762
setPixel, setPixels methods, 756, 762
write method
of CipherOutputStream, 608
of Files, 113
of ImageI0, 745, 753
of ImageWriter, 748, 755
of OutputStream, 57, 59
write/XMLWriteTest.java, 211
writeAttribute method (XMLStreamWriter), 210, 214
writeBoolean method (DataOutput), 78, 80
writeByte method (DataOutput), 78, 80
writeCData method (XMLStreamWriter), 215
writeChar method (DataOutput), 78, 80–81
writeCharacters method (XMLStreamWriter), 210, 215
writeChars method (DataOutput), 78, 80
writeComment method (XMLStreamWriter), 215
writeDouble method (DataOutput), 78, 80, 89, 101
writeDTD method (XMLStreamWriter), 215
writeEmptyElement method (XMLStreamWriter), 211, 214
writeEndElement methods (XMLStreamWriter), 210, 214
writeExternal method (Externalizable), 101–102
writeFixedString method (DataIO), 81–82
writeFloat method (DataOutput), 78, 80
writeInsert method (ImageWriter), 748, 755
writeInt method (DataOutput), 78, 80–81, 89
writeLong method (DataOutput), 78, 80
writeObject method of Date, 101
of ObjectOutputstream, 88, 93, 100
Writer class, 56, 60–61
write method, 59
writeShort method (DataOutput), 78, 80
writeStartElement methods (XMLStreamWriter), 210, 214
writeUTF method (DataOutput), 78, 80

X
X.509 format, 590
\x, in regular expressions, 138
XHTML (Extensible Hypertext Markup Language), 156, 198
XML (Extensible Markup Language)
annotated version of the standard, 155
case sensitivity of, 156
databases, 349
namespaces in, 193–196
vs. HTML, 155
XML binding, 513–514
XML catalogs, 172
XML documents
DTDs in, 156, 171–179
format of, 155
generating, 206–216
from non-XML legacy data, 221
HTML files from, 216–219
plain text from, 219–221
with StAX, 210–215
locating information in, 188–193
malformed, 211
parsing, 159–169
structure of, 156–160, 170
validating, 169–188
with/without namespaces, 206–207
XML Schema, 170, 179–182, 194
attributes in, 181
case sensitivity of, 156
end and empty tags in, 156
element names in, 179
repeated elements in, 179
validating, 169–188
with/without namespaces, 179
XML catalogs, 172–175
XMLInputFactory class
createXMLStreamReader method, 205
newInstance method, 204
setProperty method, 203–204
xmlns attribute (XSLT), 195
XML_OUTPUT_FACTORY class
createXMLStreamWriter method, 210, 214
newInstance method, 210, 214
XMLReader interface
implementing, 220
parse method, 226
setContentHandler method, 226
XMLStreamReader interface
getAttributeXXX methods, 203, 206
getName, getLocalName methods, 206
getText method, 206
hasNext, next methods, 205
isXXX methods, 205
XMLStreamWriter interface, 210
close method, 215
not autocloseable, 211
setDefaultNamespace, setPrefix methods, 214
writeAttribute method, 210, 214
writeData method, 215
writeCharacters method, 210, 215
writeComment method, 215
writeDTD method, 215
writeEmptyElement method, 211, 214
writeEndElement methods, 210, 214
writeStartElement methods, 210, 214
XOR composition rule, 737
XPath (XML Path Language), 188–193
elements/attributes in, 189
evaluating expressions in, 190–193
XPath interface
evaluate, evaluateExpression methods, 190, 193
xpath/XPathTest.java, 191
XPathEvaluationResult interface
  type method, 193
  value method, 191, 193
XPathFactory class
  newInstance method, 190, 193
  newPath method, 193
XPathNodes class, 190
xs:, xsd: prefixes (XSL Schema), 180
xsd:attribute element (XML Schema), 181
xsd:choice element (XML Schema), 181
xsd:complexType element (XML Schema), 180
xsd:element element (XML Schema), 180
xsd:enumeration element (XML Schema), 180
xsd:schema element (XML Schema), 182
xsd:sequence element (XML Schema), 181
xsd:simpleType element (XML Schema), 180
xsl:apply-templates element (XSLT), 218
xsl:output element (XSLT), 218
xsl:template element (XSLT), 218
xsl:value-of element (XSLT), 219
XSLT (Extensible Stylesheet Language Transformations), 208, 216–226
copying attribute values in, 219
templates in, 218
XSLT processor, 216
Y
Year, YearMonth classes, 360
Z
Z (boolean), type code, 95, 831
\z, \Z, in regular expressions, 140
ZIP archives, 85–88
  for JMOD files, 526
  reading, 85
  numbers from, 66
  writing, 85
Zip code lookup, 264
ZipEntry class, methods of, 87
ZipFile class, methods of, 88
ZipInputStream class, 59, 85
closeEntry, getNextEntry methods, 85–86
constructor, 86
read method, 85
ZipOutputStream class, 59, 85
closeEntry method, 85, 87
constructor, 86
putNextEntry method, 85–86
setLevel, setMethod methods, 87
ZonedDateTime class, 367–371
  from method, 376–377
  getXXX methods, 370–371
  isAfter, isBefore methods, 371
  legacy classes and, 376–377
  minus, minusXXX methods, 370
  now method, 370
  of, ofInstant methods, 367, 370
  parse method, 376, 395, 401
  plus, plusXXX methods, 370
  toInstant method, 367, 371
toLocalXXX methods, 371
  withXXX methods, 370
zonedtimes/ZonedTimes.java, 369
ZoneId class, 377
  getAvailableZoneIds, of methods, 367