

C O R E
JAVA®

Volume II—Advanced Features

TENTH EDITION



CAY S. HORSTMANN



Core Java®

Volume II—Advanced Features

Tenth Edition

This page intentionally left blank



Core Java®

Volume II—Advanced Features

Tenth Edition

Cay S. Horstmann



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

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

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

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

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

Visit us on the Web: informit.com/ph

Library of Congress Catalog Number: 2016952666

Copyright © 2017 Oracle and/or its affiliates. All rights reserved.
500 Oracle Parkway, Redwood Shores, CA 94065

Portions © 2017 Cay S. Horstmann

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

Oracle America Inc. does not make any representations or warranties as to the accuracy, adequacy or completeness of any information contained in this work, and is not responsible for any errors or omissions.

ISBN-13: 978-0-13-417729-8

ISBN-10: 0-13-417729-0

Text printed in the United States of America.

1 16

Contents

<i>Preface</i>	xv
<i>Acknowledgments</i>	xix
Chapter 1: The Java SE 8 Stream Library	1
1.1 From Iterating to Stream Operations	2
1.2 Stream Creation	5
1.3 The <code>filter</code> , <code>map</code> , and <code>flatMap</code> Methods	9
1.4 Extracting Substreams and Concatenating Streams	10
1.5 Other Stream Transformations	11
1.6 Simple Reductions	12
1.7 The Optional Type	13
1.7.1 How to Work with Optional Values	14
1.7.2 How Not to Work with Optional Values	15
1.7.3 Creating Optional Values	16
1.7.4 Composing Optional Value Functions with <code>flatMap</code>	16
1.8 Collecting Results	19
1.9 Collecting into Maps	24
1.10 Grouping and Partitioning	28
1.11 Downstream Collectors	29
1.12 Reduction Operations	33
1.13 Primitive Type Streams	36
1.14 Parallel Streams	41
Chapter 2: Input and Output	47
2.1 Input/Output Streams	48
2.1.1 Reading and Writing Bytes	48
2.1.2 The Complete Stream Zoo	51
2.1.3 Combining Input/Output Stream Filters	55

2.2	Text Input and Output	60
2.2.1	How to Write Text Output	60
2.2.2	How to Read Text Input	62
2.2.3	Saving Objects in Text Format	63
2.2.4	Character Encodings	67
2.3	Reading and Writing Binary Data	69
2.3.1	The <code>DataInput</code> and <code>DataOutput</code> interfaces	69
2.3.2	Random-Access Files	72
2.3.3	ZIP Archives	77
2.4	Object Input/Output Streams and Serialization	80
2.4.1	Saving and Loading Serializable Objects	80
2.4.2	Understanding the Object Serialization File Format	85
2.4.3	Modifying the Default Serialization Mechanism	92
2.4.4	Serializing Singletons and Typesafe Enumerations	94
2.4.5	Versioning	95
2.4.6	Using Serialization for Cloning	98
2.5	Working with Files	100
2.5.1	Paths	101
2.5.2	Reading and Writing Files	104
2.5.3	Creating Files and Directories	105
2.5.4	Copying, Moving, and Deleting Files	106
2.5.5	Getting File Information	108
2.5.6	Visiting Directory Entries	110
2.5.7	Using Directory Streams	111
2.5.8	ZIP File Systems	115
2.6	Memory-Mapped Files	116
2.6.1	Memory-Mapped File Performance	116
2.6.2	The Buffer Data Structure	124
2.6.3	File Locking	126
2.7	Regular Expressions	128
Chapter 3: XML		143
3.1	Introducing XML	144
3.1.1	The Structure of an XML Document	146

3.2	Parsing an XML Document	149
3.3	Validating XML Documents	162
3.3.1	Document Type Definitions	163
3.3.2	XML Schema	172
3.3.3	A Practical Example	175
3.4	Locating Information with XPath	190
3.5	Using Namespaces	196
3.6	Streaming Parsers	199
3.6.1	Using the SAX Parser	199
3.6.2	Using the StAX Parser	205
3.7	Generating XML Documents	208
3.7.1	Documents without Namespaces	209
3.7.2	Documents with Namespaces	209
3.7.3	Writing Documents	210
3.7.4	An Example: Generating an SVG File	211
3.7.5	Writing an XML Document with StAX	214
3.8	XSL Transformations	222
Chapter 4: Networking		233
4.1	Connecting to a Server	233
4.1.1	Using Telnet	233
4.1.2	Connecting to a Server with Java	236
4.1.3	Socket Timeouts	238
4.1.4	Internet Addresses	239
4.2	Implementing Servers	241
4.2.1	Server Sockets	242
4.2.2	Serving Multiple Clients	245
4.2.3	Half-Close	249
4.3	Interruptible Sockets	250
4.4	Getting Web Data	257
4.4.1	URLs and URIs	257
4.4.2	Using a <code>URLConnection</code> to Retrieve Information	259
4.4.3	Posting Form Data	267
4.5	Sending E-Mail	277

Chapter 5: Database Programming	281
5.1 The Design of JDBC	282
5.1.1 JDBC Driver Types	283
5.1.2 Typical Uses of JDBC	284
5.2 The Structured Query Language	285
5.3 JDBC Configuration	291
5.3.1 Database URLs	292
5.3.2 Driver JAR Files	292
5.3.3 Starting the Database	293
5.3.4 Registering the Driver Class	294
5.3.5 Connecting to the Database	294
5.4 Working with JDBC Statements	297
5.4.1 Executing SQL Statements	298
5.4.2 Managing Connections, Statements, and Result Sets	301
5.4.3 Analyzing SQL Exceptions	302
5.4.4 Populating a Database	305
5.5 Query Execution	309
5.5.1 Prepared Statements	309
5.5.2 Reading and Writing LOBs	316
5.5.3 SQL Escapes	318
5.5.4 Multiple Results	319
5.5.5 Retrieving Autogenerated Keys	320
5.6 Scrollable and Updatable Result Sets	321
5.6.1 Scrollable Result Sets	321
5.6.2 Updatable Result Sets	324
5.7 Row Sets	328
5.7.1 Constructing Row Sets	329
5.7.2 Cached Row Sets	329
5.8 Metadata	333
5.9 Transactions	344
5.9.1 Programming Transactions with JDBC	344
5.9.2 Save Points	345
5.9.3 Batch Updates	345
5.10 Advanced SQL Types	347
5.11 Connection Management in Web and Enterprise Applications	349

Chapter 6: The Date and Time API	351
6.1 The Time Line	352
6.2 Local Dates	355
6.3 Date Adjusters	358
6.4 Local Time	360
6.5 Zoned Time	361
6.6 Formatting and Parsing	365
6.7 Interoperating with Legacy Code	369
Chapter 7: Internationalization	371
7.1 Locales	372
7.2 Number Formats	378
7.3 Currencies	384
7.4 Date and Time	385
7.5 Collation and Normalization	393
7.6 Message Formatting	400
7.6.1 Formatting Numbers and Dates	400
7.6.2 Choice Formats	402
7.7 Text Input and Output	404
7.7.1 Text Files	405
7.7.2 Line Endings	405
7.7.3 The Console	405
7.7.4 Log Files	406
7.7.5 The UTF-8 Byte Order Mark	406
7.7.6 Character Encoding of Source Files	407
7.8 Resource Bundles	408
7.8.1 Locating Resource Bundles	409
7.8.2 Property Files	410
7.8.3 Bundle Classes	411
7.9 A Complete Example	413
Chapter 8: Scripting, Compiling, and Annotation Processing	429
8.1 Scripting for the Java Platform	430
8.1.1 Getting a Scripting Engine	430
8.1.2 Script Evaluation and Bindings	431
8.1.3 Redirecting Input and Output	434

8.1.4	Calling Scripting Functions and Methods	435
8.1.5	Compiling a Script	437
8.1.6	An Example: Scripting GUI Events	437
8.2	The Compiler API	443
8.2.1	Compiling the Easy Way	443
8.2.2	Using Compilation Tasks	443
8.2.3	An Example: Dynamic Java Code Generation	449
8.3	Using Annotations	455
8.3.1	An Introduction into Annotations	455
8.3.2	An Example: Annotating Event Handlers	457
8.4	Annotation Syntax	462
8.4.1	Annotation Interfaces	462
8.4.2	Annotations	464
8.4.3	Annotating Declarations	466
8.4.4	Annotating Type Uses	467
8.4.5	Annotating this	468
8.5	Standard Annotations	470
8.5.1	Annotations for Compilation	471
8.5.2	Annotations for Managing Resources	472
8.5.3	Meta-Annotations	472
8.6	Source-Level Annotation Processing	475
8.6.1	Annotation Processors	476
8.6.2	The Language Model API	476
8.6.3	Using Annotations to Generate Source Code	477
8.7	Bytecode Engineering	481
8.7.1	Modifying Class Files	481
8.7.2	Modifying Bytecodes at Load Time	486
Chapter 9: Security	491	
9.1	Class Loaders	492
9.1.1	The Class Loading Process	492
9.1.2	The Class Loader Hierarchy	494
9.1.3	Using Class Loaders as Namespaces	496
9.1.4	Writing Your Own Class Loader	497
9.1.5	Bytecode Verification	504

9.2	Security Managers and Permissions	509
9.2.1	Permission Checking	509
9.2.2	Java Platform Security	510
9.2.3	Security Policy Files	514
9.2.4	Custom Permissions	522
9.2.5	Implementation of a Permission Class	524
9.3	User Authentication	530
9.3.1	The JAAS Framework	531
9.3.2	JAAS Login Modules	537
9.4	Digital Signatures	546
9.4.1	Message Digests	547
9.4.2	Message Signing	550
9.4.3	Verifying a Signature	553
9.4.4	The Authentication Problem	556
9.4.5	Certificate Signing	558
9.4.6	Certificate Requests	560
9.4.7	Code Signing	561
9.5	Encryption	567
9.5.1	Symmetric Ciphers	567
9.5.2	Key Generation	569
9.5.3	Cipher Streams	574
9.5.4	Public Key Ciphers	575
Chapter 10: Advanced Swing		581
10.1	Lists	582
10.1.1	The <code>JList</code> Component	582
10.1.2	List Models	588
10.1.3	Inserting and Removing Values	593
10.1.4	Rendering Values	595
10.2	Tables	599
10.2.1	A Simple Table	600
10.2.2	Table Models	604
10.2.3	Working with Rows and Columns	608
10.2.3.1	Column Classes	609
10.2.3.2	Accessing Table Columns	610
10.2.3.3	Resizing Columns	611

10.2.3.4	Resizing Rows	612
10.2.3.5	Selecting Rows, Columns, and Cells	612
10.2.3.6	Sorting Rows	614
10.2.3.7	Filtering Rows	615
10.2.3.8	Hiding and Displaying Columns	617
10.2.4	Cell Rendering and Editing	626
10.2.4.1	Rendering Cells	626
10.2.4.2	Rendering the Header	627
10.2.4.3	Editing Cells	628
10.2.4.4	Custom Editors	629
10.3	Trees	639
10.3.1	Simple Trees	640
10.3.2	Editing Trees and Tree Paths	650
10.3.3	Node Enumeration	659
10.3.4	Rendering Nodes	661
10.3.5	Listening to Tree Events	664
10.3.6	Custom Tree Models	671
10.4	Text Components	681
10.4.1	Change Tracking in Text Components	682
10.4.2	Formatted Input Fields	685
10.4.2.1	Integer Input	686
10.4.2.2	Behavior on Loss of Focus	687
10.4.2.3	Filters	688
10.4.2.4	Verifiers	690
10.4.2.5	Other Standard Formatters	691
10.4.2.6	Custom Formatters	693
10.4.3	The JSpinner Component	703
10.4.4	Displaying HTML with the JEditorPane	712
10.5	Progress Indicators	719
10.5.1	Progress Bars	719
10.5.2	Progress Monitors	722
10.5.3	Monitoring the Progress of Input Streams	726
10.6	Component Organizers and Decorators	731
10.6.1	Split Panes	732
10.6.2	Tabbed Panes	735

10.6.3	Desktop Panes and Internal Frames	741
10.6.3.1	Displaying Internal Frames	741
10.6.3.2	Cascading and Tiling	744
10.6.3.3	Vetoing Property Settings	748
10.6.3.4	Dialogs in Internal Frames	750
10.6.3.5	Outline Dragging	751
10.6.4	Layers	760
Chapter 11: Advanced AWT	765	
11.1	The Rendering Pipeline	766
11.2	Shapes	769
11.2.1	The Shape Class Hierarchy	769
11.2.2	Using the Shape Classes	772
11.3	Areas	786
11.4	Strokes	788
11.5	Paint	797
11.6	Coordinate Transformations	799
11.7	Clipping	805
11.8	Transparency and Composition	807
11.9	Rendering Hints	817
11.10	Readers and Writers for Images	823
11.10.1	Obtaining Readers and Writers for Image File Types	824
11.10.2	Reading and Writing Files with Multiple Images	825
11.11	Image Manipulation	834
11.11.1	Constructing Raster Images	835
11.11.2	Filtering Images	842
11.12	Printing	851
11.12.1	Graphics Printing	852
11.12.2	Multiple-Page Printing	862
11.12.3	Print Preview	864
11.12.4	Print Services	874
11.12.5	Stream Print Services	878
11.12.6	Printing Attributes	879
11.13	The Clipboard	887
11.13.1	Classes and Interfaces for Data Transfer	888
11.13.2	Transferring Text	888

11.13.3 The Transferable Interface and Data Flavors	892
11.13.4 Building an Image Transferable	894
11.13.5 Transferring Java Objects via the System Clipboard	898
11.13.6 Using a Local Clipboard to Transfer Object References	902
11.14 Drag and Drop	903
11.14.1 Data Transfer Support in Swing	904
11.14.2 Drag Sources	909
11.14.3 Drop Targets	912
11.15 Platform Integration	921
11.15.1 Splash Screens	921
11.15.2 Launching Desktop Applications	927
11.15.3 The System Tray	932
Chapter 12: Native Methods	939
12.1 Calling a C Function from a Java Program	940
12.2 Numeric Parameters and Return Values	947
12.3 String Parameters	949
12.4 Accessing Fields	956
12.4.1 Accessing Instance Fields	956
12.4.2 Accessing Static Fields	960
12.5 Encoding Signatures	961
12.6 Calling Java Methods	963
12.6.1 Instance Methods	963
12.6.2 Static Methods	964
12.6.3 Constructors	965
12.6.4 Alternative Method Invocations	966
12.7 Accessing Array Elements	970
12.8 Handling Errors	974
12.9 Using the Invocation API	980
12.10 A Complete Example: Accessing the Windows Registry	985
12.10.1 Overview of the Windows Registry	985
12.10.2 A Java Platform Interface for Accessing the Registry	987
12.10.3 Implementation of Registry Access Functions as Native Methods	988
<i>Index</i>	1002

Preface

To the Reader

The book you have in your hands is the second volume of the tenth edition of *Core Java®*, fully updated for Java SE 8. 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 to 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 8 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.

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.

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 is 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 in 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.

Chapter 9 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 10 contains all the Swing material that didn't make it into Volume I, especially the important but complex tree and table components. We show the basic uses of editor panes, the Java implementation of a "multiple document" interface, progress indicators used in multithreaded programs, and "desktop integration features" such as splash screens and support for the system tray. Again, we focus on the most useful constructs that you are likely to encounter in practical programming because an encyclopedic coverage of the entire Swing library would fill several volumes and would only be of interest to dedicated taxonomists.

Chapter 11 covers the Java 2D API, which you can use to create realistic drawings and special effects. The chapter also covers some advanced features of the AWT (Abstract Windowing Toolkit) that seemed too specialized for coverage in Volume I but should, nonetheless, be part of every programmer's toolkit. These features include printing and the APIs for cut-and-paste and drag-and-drop.

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 SE 8 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.

Application Programming Interface 1.2

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>.

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 Prentice Hall 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
September 2016*

The Java SE 8 Stream Library

In this chapter

- 1.1 From Iterating to Stream Operations, page 2
- 1.2 Stream Creation, page 5
- 1.3 The `filter`, `map`, and `flatMap` Methods, page 9
- 1.4 Extracting Substreams and Concatenating Streams, page 10
- 1.5 Other Stream Transformations, page 11
- 1.6 Simple Reductions, page 12
- 1.7 The Optional Type, page 13
- 1.8 Collecting Results, page 19
- 1.9 Collecting into Maps, page 24
- 1.10 Grouping and Partitioning, page 28
- 1.11 Downstream Collectors, page 29
- 1.12 Reduction Operations, page 33
- 1.13 Primitive Type Streams, page 36
- 1.14 Parallel Streams, page 41

Streams provide a view of data that lets you specify computations at a higher conceptual level than with collections. With a stream, you specify what you want to have done, not how to do it. You leave the scheduling of operations to the implementation. For example, suppose you want to compute the average of a certain property. You specify the source of data and the property, and the stream library

can then optimize the computation, for example by using multiple threads for computing sums and counts and combining the results.

In this chapter, you will learn how to use the Java stream library, which was introduced in Java SE 8, to process collections in a “what, not how” style.

1.1 From Iterating to Stream Operations

When you process a collection, you usually iterate over its elements and do some work with each of them. For example, suppose we want to count all long words in a book. First, let’s put them into a list:

```
String contents = new String(Files.readAllBytes(  
    Paths.get("alice.txt"), StandardCharsets.UTF_8)); // Read file into string  
List<String> words = Arrays.asList(contents.split("\\PL+"));  
// Split into words; nonletters are delimiters
```

Now we are ready to iterate:

```
long count = 0;  
for (String w : words)  
{  
    if (w.length() > 12) count++;  
}
```

With streams, the same operation looks like this:

```
long count = words.stream()  
.filter(w -> w.length() > 12)  
.count();
```

The stream version is easier to read than the loop because you do not have to scan the code for evidence of filtering and counting. The method names tell you right away what the code intends to do. Moreover, while the loop prescribes the order of operations in complete detail, a stream is able to schedule the operations any way it wants, as long as the result is correct.

Simply changing `stream` into `parallelStream` allows the stream library to do the filtering and counting in parallel.

```
long count = words.parallelStream()  
.filter(w -> w.length() > 12)  
.count();
```

Streams follow the “what, not how” principle. In our stream example, we describe what needs to be done: get the long words and count them. We don’t specify in which order, or in which thread, this should happen. In contrast, the loop at the beginning of this section specifies exactly how the computation should work, and thereby forgoes any chances of optimization.

A stream seems superficially similar to a collection, allowing you to transform and retrieve data. But there are significant differences:

1. A stream does not store its elements. They may be stored in an underlying collection or generated on demand.
2. Stream operations don’t mutate their source. For example, the `filter` method does not remove elements from a new stream, but it yields a new stream in which they are not present.
3. Stream operations are *lazy* when possible. This means they are not executed until their result is needed. For example, if you only ask for the first five long words instead of all, the `filter` method will stop filtering after the fifth match. As a consequence, you can even have infinite streams!

Let us have another look at the example. The `stream` and `parallelStream` methods yield a *stream* for the `words` list. The `filter` method returns another stream that contains only the words of length greater than twelve. The `count` method reduces that stream to a result.

This workflow is typical when you work with streams. You set up a pipeline of operations in three stages:

1. Create a stream.
2. Specify *intermediate operations* for transforming the initial stream into others, possibly in multiple steps.
3. Apply a *terminal operation* to produce a result. This operation forces the execution of the lazy operations that precede it. Afterwards, the stream can no longer be used.

In the example in Listing 1.1, the stream is created with the `stream` or `parallelStream` method. The `filter` method transforms it, and `count` is the terminal operation.

In the next section, you will see how to create a stream. The subsequent three sections deal with stream transformations. They are followed by five sections on terminal operations.

Listing 1.1 streams/CountLongWords.java

```
1 package streams;
2
3 import java.io.IOException;
4 import java.nio.charset.StandardCharsets;
5 import java.nio.file.Files;
6 import java.nio.file.Paths;
7 import java.util.Arrays;
8 import java.util.List;
9
10 public class CountLongWords
11 {
12     public static void main(String[] args) throws IOException
13     {
14         String contents = new String(Files.readAllBytes(
15             Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
16         List<String> words = Arrays.asList(contents.split("\\PL+"));
17
18         long count = 0;
19         for (String w : words)
20         {
21             if (w.length() > 12) count++;
22         }
23         System.out.println(count);
24
25         count = words.stream().filter(w -> w.length() > 12).count();
26         System.out.println(count);
27
28         count = words.parallelStream().filter(w -> w.length() > 12).count();
29         System.out.println(count);
30     }
31 }
```

java.util.stream.Stream<T> 8

- `Stream<T> filter(Predicate<? super T> p)`
yields a stream containing all elements of this stream fulfilling `p`.
- `long count()`
yields the number of elements of this stream. This is a terminal operation.

java.util.Collection<E> 1.2

- default Stream<E> stream()
- default Stream<E> parallelStream()

yields a sequential or parallel stream of the elements in this collection.

1.2 Stream Creation

You have already seen that you can turn any collection into a stream with the `stream` method of the `Collection` interface. If you have an array, use the static `Stream.of` method instead.

```
Stream<String> words = Stream.of(contents.split("\\PL+"));
// split returns a String[] array
```

The `of` method has a varargs parameter, so you can construct a stream from any number of arguments:

```
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

Use `Arrays.stream(array, from, to)` to make a stream from array elements between positions `from` (inclusive) and `to` (exclusive).

To make a stream with no elements, use the static `Stream.empty` method:

```
Stream<String> silence = Stream.empty();
// Generic type <String> is inferred; same as Stream.<String>empty()
```

The `Stream` interface has two static methods for making infinite streams. The `generate` method takes a function with no arguments (or, technically, an object of the `Supplier<T>` interface). Whenever a stream value is needed, that function is called to produce a value. You can get a stream of constant values as

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

or a stream of random numbers as

```
Stream<Double> randoms = Stream.generate(Math::random);
```

To produce infinite sequences, such as 0 1 2 3 . . . , use the `iterate` method instead. It takes a “seed” value and a function (technically, a `UnaryOperator<T>`) and repeatedly applies the function to the previous result. For example,

```
Stream<BigInteger> integers
    = Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

The first element in the sequence is the seed `BigInteger.ZERO`. The second element is `f(seed)`, or 1 (as a big integer). The next element is `f(f(seed))`, or 2, and so on.

NOTE: A number of methods in the Java API yield streams. For example, the `Pattern` class has a method `splitAsStream` that splits a `CharSequence` by a regular expression. You can use the following statement to split a string into words:

```
Stream<String> words = Pattern.compile("\\PL+").splitAsStream(contents);
```

The static `Files.lines` method returns a `Stream` of all lines in a file:

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

The example program in Listing 1.2 shows the various ways of creating a stream.

Listing 1.2 streams/CreatingStreams.java

```
1 package streams;
2
3 import java.io.IOException;
4 import java.math.BigInteger;
5 import java.nio.charset.StandardCharsets;
6 import java.nio.file.Files;
7 import java.nio.file.Path;
8 import java.nio.file.Paths;
9 import java.util.List;
10 import java.util.regex.Pattern;
11 import java.util.stream.Collectors;
12 import java.util.stream.Stream;
13
14 public class CreatingStreams
15 {
16     public static <T> void show(String title, Stream<T> stream)
17     {
18         final int SIZE = 10;
19         List<T> firstElements = stream
20             .limit(SIZE + 1)
21             .collect(Collectors.toList());
```

```
22     System.out.print(title + ": ");
23     for (int i = 0; i < firstElements.size(); i++)
24     {
25         if (i > 0) System.out.print(", ");
26         if (i < SIZE) System.out.print(firstElements.get(i));
27         else System.out.print("... ");
28     }
29     System.out.println();
30 }
31
32 public static void main(String[] args) throws IOException
33 {
34     Path path = Paths.get("../gutenberg/alice30.txt");
35     String contents = new String(Files.readAllBytes(path),
36         StandardCharsets.UTF_8);
37
38     Stream<String> words = Stream.of(contents.split("\\PL+"));
39     show("words", words);
40     Stream<String> song = Stream.of("gently", "down", "the", "stream");
41     show("song", song);
42     Stream<String> silence = Stream.empty();
43     show("silence", silence);
44
45     Stream<String> echos = Stream.generate(() -> "Echo");
46     show("echos", echos);
47
48     Stream<Double> randoms = Stream.generate(Math::random);
49     show("randoms", randoms);
50
51     Stream<BigInteger> integers = Stream.iterate(BigInteger.ONE,
52         n -> n.add(BigInteger.ONE));
53     show("integers", integers);
54
55     Stream<String> wordsAnotherWay = Pattern.compile("\\PL+").splitAsStream(
56         contents);
57     show("wordsAnotherWay", wordsAnotherWay);
58
59     try (Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8))
60     {
61         show("lines", lines);
62     }
63 }
64 }
```

java.util.stream.Stream 8

- static <T> Stream<T> of(T... values)
yields a stream whose elements are the given values.
- static <T> Stream<T> empty()
yields a stream with no elements.
- static <T> Stream<T> generate(Supplier<T> s)
yields an infinite stream whose elements are constructed by repeatedly invoking the function s.
- static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
yields an infinite stream whose elements are seed, f invoked on seed, f invoked on the preceding element, and so on.

java.util.Arrays 1.2

- static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive) 8
yields a stream whose elements are the specified range of the array.

java.util.regex.Pattern 1.4

- Stream<String> splitAsStream(CharSequence input) 8
yields a stream whose elements are the parts of the input that are delimited by this pattern.

java.nio.file.Files 7

- static Stream<String> lines(Path path) 8
- static Stream<String> lines(Path path, Charset cs) 8
yields a stream whose elements are the lines of the specified file, with the UTF-8 charset or the given charset.

***java.util.function.Supplier*<T>** 8

- T get()
supplies a value.

1.3 The filter, map, and flatMap Methods

A stream transformation produces a stream whose elements are derived from those of another stream. You have already seen the `filter` transformation that yields a stream with those elements that match a certain condition. Here, we transform a stream of strings into another stream containing only long words:

```
List<String> wordList = . . .;
Stream<String> longWords = wordList.stream().filter(w -> w.length() > 12);
```

The argument of `filter` is a `Predicate<T>`—that is, a function from `T` to `boolean`.

Often, you want to transform the values in a stream in some way. Use the `map` method and pass the function that carries out the transformation. For example, you can transform all words to lowercase like this:

```
Stream<String> lowercaseWords = words.stream().map(String::toLowerCase);
```

Here, we used `map` with a method reference. Often, a lambda expression is used instead:

```
Stream<String> firstLetters = words.stream().map(s -> s.substring(0, 1));
```

The resulting stream contains the first letters of all words.

When you use `map`, a function is applied to each element, and the result is a new stream with the results. Now, suppose you have a function that returns not just one value but a stream of values:

```
public static Stream<String> letters(String s)
{
    List<String> result = new ArrayList<>();
    for (int i = 0; i < s.length(); i++)
        result.add(s.substring(i, i + 1));
    return result.stream();
}
```

For example, `letters("boat")` is the stream `["b", "o", "a", "t"]`.

NOTE: With the `IntStream.range` method in Section 1.13, “Primitive Type Streams,” on p. 36, you can implement this method much more elegantly.

Suppose you map the `letters` method on a stream of strings:

```
Stream<Stream<String>> result = words.stream().map(w -> letters(w));
```

You will get a stream of streams, like `[. . . ["y", "o", "u", "r"], ["b", "o", "a", "t"], . . .]`. To flatten it out to a stream of letters `[. . . "y", "o", "u", "r", "b", "o", "a", "t", . . .]`, use the `flatMap` method instead of `map`:

```
Stream<String> flatResult = words.stream().flatMap(w -> letters(w))
// Calls letters on each word and flattens the results
```

NOTE: You will find a `flatMap` method in classes other than streams. It is a general concept in computer science. Suppose you have a generic type G (such as `Stream`) and functions f from some type T to $G<U>$ and g from U to $G<V>$. Then you can compose them—that is, first apply f and then g , by using `flatMap`. This is a key idea in the theory of *monads*. But don't worry—you can use `flatMap` without knowing anything about monads.

`java.util.stream.Stream` 8

- `Stream<T> filter(Predicate<? super T> predicate)`
yields a stream containing the elements of this stream that fulfill the predicate.
- `<R> Stream<R> map(Function<? super T,>? extends R> mapper)`
yields a stream containing the results of applying `mapper` to the elements of this stream.
- `<R> Stream<R> flatMap(Function<? super T,>? extends Stream<? extends R>> mapper)`
yields a stream obtained by concatenating the results of applying `mapper` to the elements of this stream. (Note that each result is a stream.)

1.4 Extracting Substreams and Concatenating Streams

The call `stream.limit(n)` returns a new stream that ends after n elements (or when the original stream ends, if it is shorter). This method is particularly useful for cutting infinite streams down to size. For example,

```
Stream<Double> randoms = Stream.generate(Math::random).limit(100);
```

yields a stream with 100 random numbers.

The call `stream.skip(n)` does the exact opposite: It discards the first n elements. This is handy when splitting text into words since, due to the way the `split` method works, the first element is an unwanted empty string. We can make it go away by calling `skip`:

```
Stream<String> words = Stream.of(contents.split("\\PL+")).skip(1);
```

You can concatenate two streams with the static `concat` method of the `Stream` class:

```
Stream<String> combined = Stream.concat(
    letters("Hello"), letters("World"));
// Yields the stream ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]
```

Of course the first stream should not be infinite—otherwise the second one will never get a chance.

`java.util.stream.Stream` 8

- `Stream<T> limit(long maxSize)`
yields a stream with up to `maxSize` of the initial elements from this stream.
- `Stream<T> skip(long n)`
yields a stream whose elements are all but the initial `n` elements of this stream.
- `static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)`
yields a stream whose elements are the elements of `a` followed by the elements of `b`.

1.5 Other Stream Transformations

The `distinct` method returns a stream that yields elements from the original stream, in the same order, except that duplicates are suppressed. The stream must obviously remember the elements that it has already seen.

```
Stream<String> uniqueWords =
    Stream.of("merrily", "merrily", "merrily", "gently").distinct();
// Only one "merrily" is retained
```

For sorting a stream, there are several variations of the `sorted` method. One works for streams of `Comparable` elements, and another accepts a `Comparator`. Here, we sort strings so that the longest ones come first:

```
Stream<String> longestFirst =
    words.stream().sorted(Comparator.comparing(String::length).reversed());
```

As with all stream transformations, the `sorted` method yields a new stream whose elements are the elements of the original stream in sorted order.

Of course, you can sort a collection without using streams. The `sorted` method is useful when the sorting process is part of a stream pipeline.

Finally, the `peek` method yields another stream with the same elements as the original, but a function is invoked every time an element is retrieved. That is handy for debugging:

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
    .peek(e -> System.out.println("Fetching " + e))
    .limit(20).toArray();
```

When an element is actually accessed, a message is printed. This way you can verify that the infinite stream returned by `iterate` is processed lazily.

For debugging, you can have `peek` call a method into which you set a breakpoint.

java.util.stream.Stream 8

- `Stream<T> distinct()`
yields a stream of the distinct elements of this stream.
- `Stream<T> sorted()`
- `Stream<T> sorted(Comparator<? super T> comparator)`
yields as stream whose elements are the elements of this stream in sorted order.
The first method requires that the elements are instances of a class implementing `Comparable`.
- `Stream<T> peek(Consumer<? super T> action)`
yields a stream with the same elements as this stream, passing each element to `action` as it is consumed.

1.6 Simple Reductions

Now that you have seen how to create and transform streams, we will finally get to the most important point—getting answers from the stream data. The methods that we cover in this section are called *reductions*. Reductions are *terminal operations*. They reduce the stream to a non-stream value that can be used in your program.

You have already seen a simple reduction: The `count` method returns the number of elements of a stream.

Other simple reductions are `max` and `min` that return the largest or smallest value. There is a twist—these methods return an `Optional<T>` value that either wraps the answer or indicates that there is none (because the stream happened to be empty). In the olden days, it was common to return `null` in such a situation. But that can lead to null pointer exceptions when it happens in an incompletely tested program. The `Optional` type is a better way of indicating a missing return value. We discuss the `Optional` type in detail in the next section. Here is how you can get the maximum of a stream:

```
Optional<String> largest = words.max(String::compareToIgnoreCase);
System.out.println("largest: " + largest.orElse "");
```

The `findFirst` returns the first value in a nonempty collection. It is often useful when combined with `filter`. For example, here we find the first word that starts with the letter Q, if it exists:

```
Optional<String> startsWithQ = words.filter(s -> s.startsWith("Q")).findFirst();
```

If you are OK with any match, not just the first one, use the `findAny` method. This is effective when you parallelize the stream, since the stream can report any match it finds instead of being constrained to the first one.

```
Optional<String> startsWithQ = words.parallel().filter(s -> s.startsWith("Q")).findAny();
```

If you just want to know if there is a match, use `anyMatch`. That method takes a predicate argument, so you won't need to use `filter`.

```
boolean aWordStartsWithQ = words.parallel().anyMatch(s -> s.startsWith("Q"));
```

There are methods `allMatch` and `noneMatch` that return true if all or no elements match a predicate. These methods also benefit from being run in parallel.

java.util.stream.Stream 8

- `Optional<T> max(Comparator<? super T> comparator)`
- `Optional<T> min(Comparator<? super T> comparator)`

yields a maximum or minimum element of this stream, using the ordering defined by the given comparator, or an empty `Optional` if this stream is empty. These are terminal operations.

- `Optional<T> findFirst()`
- `Optional<T> findAny()`

yields the first, or any, element of this stream, or an empty `Optional` if this stream is empty. These are terminal operations.

- `boolean anyMatch(Predicate<? super T> predicate)`
- `boolean allMatch(Predicate<? super T> predicate)`
- `boolean noneMatch(Predicate<? super T> predicate)`

returns true if any, all, or none of the elements of this stream match the given predicate. These are terminal operations.

1.7 The Optional Type

An `Optional<T>` object is a wrapper for either an object of type `T` or no object. In the former case, we say that the value is *present*. The `Optional<T>` type is intended as a

safers alternative for a reference of type `T` that either refers to an object or is `null`. However, it is only safer if you use it right. The next section shows you how.

1.7.1 How to Work with Optional Values

The key to using `Optional` effectively is to use a method that either *produces an alternative* if the value is not present, or *consumes the value* only if it is present.

Let us look at the first strategy. Often, there is a default that you want to use when there was no match, perhaps the empty string:

```
String result = optionalString.orElse("");
// The wrapped string, or "" if none
```

You can also invoke code to compute the default:

```
String result = optionalString.orElseGet(() -> Locale.getDefault().getDisplayName());
// The function is only called when needed
```

Or you can throw an exception if there is no value:

```
String result = optionalString.orElseThrow(IllegalStateException::new);
// Supply a method that yields an exception object
```

You have just seen how to produce an alternative if no value is present. The other strategy for working with optional values is to consume the value only if it is present.

The `ifPresent` method accepts a function. If the optional value exists, it is passed to that function. Otherwise, nothing happens.

```
optionalValue.ifPresent(v -> Process v);
```

For example, if you want to add the value to a set if it is present, call

```
optionalValue.ifPresent(v -> results.add(v));
```

or simply

```
optionalValue.ifPresent(results::add);
```

When calling `ifPresent`, no value is returned from the function. If you want to process the function result, use `map` instead:

```
Optional<Boolean> added = optionalValue.map(results::add);
```

Now `added` has one of three values: `true` or `false` wrapped into an `Optional`, if `optionalValue` was present, or an empty `Optional` otherwise.

NOTE: This `map` method is the analog of the `map` method of the `Stream` interface that you have seen in Section 1.3, “The `filter`, `map`, and `flatMap` Methods,” on p. 9. Simply imagine an optional value as a stream of size zero or one. The result also has size zero or one, and in the latter case, the function has been applied.

java.util.Optional 8

- `T orElse(T other)`
yields the value of this `Optional`, or `other` if this `Optional` is empty.
- `T orElseGet(Supplier<? extends T> other)`
yields the value of this `Optional`, or the result of invoking `other` if this `Optional` is empty.
- `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`
yields the value of this `Optional`, or throws the result of invoking `exceptionSupplier` if this `Optional` is empty.
- `void ifPresent(Consumer<? super T> consumer)`
if this `Optional` is nonempty, passes its value to `consumer`.
- `<U> Optional<U> map(Function<? super T, ? extends U> mapper)`
yields the result of passing the value of this `Optional` to `mapper`, provided this `Optional` is nonempty and the result is not `null`, or an empty `Optional` otherwise.

1.7.2 How Not to Work with Optional Values

If you don’t use `Optional` values correctly, you get no benefit over the “something or `null`” approach of the past.

The `get` method gets the wrapped element of an `Optional` value if it exists, or throws a `NoSuchElementException` if it doesn’t. Therefore,

```
Optional<T> optionalValue = . . .;
optionalValue.get().someMethod();
```

is no safer than

```
T value = . . .;
value.someMethod();
```

The `isPresent` method reports whether an `Optional<T>` object has a value. But

```
if (optionalValue.isPresent()) optionalValue.get().someMethod();
```

is no easier than

```
if (value != null) value.someMethod();
```

java.util.Optional 8

- `T get()`
yields the value of this `Optional`, or throws a `NoSuchElementException` if it is empty.
- `boolean isPresent()`
returns `true` if this `Optional` is not empty.

1.7.3 Creating Optional Values

So far, we have discussed how to consume an `Optional` object someone else created. If you want to write a method that creates an `Optional` object, there are several static methods for that purpose, including `Optional.of(result)` and `Optional.empty()`. For example,

```
public static Optional<Double> inverse(Double x)
{
    return x == 0 ? Optional.empty() : Optional.of(1 / x);
}
```

The `ofNullable` method is intended as a bridge from possibly `null` values to optional values. `Optional.ofNullable(obj)` returns `Optional.of(obj)` if `obj` is not `null` and `Optional.empty()` otherwise.

java.util.Optional 8

- `static <T> Optional<T> of(T value)`
- `static <T> Optional<T> ofNullable(T value)`
yields an `Optional` with the given value. If `value` is `null`, the first method throws a `NullPointerException` and the second method yields an empty `Optional`.
- `static <T> Optional<T> empty()`
yields an empty `Optional`.

1.7.4 Composing Optional Value Functions with flatMap

Suppose you have a method `f` yielding an `Optional<T>`, and the target type `T` has a method `g` yielding an `Optional<U>`. If they were normal methods, you could compose them by calling `s.f().g()`. But that composition doesn't work since `s.f()` has type `Optional<T>`, not `T`. Instead, call

```
Optional<U> result = s.f().flatMap(T::g);
```

If `s.f()` is present, then `g` is applied to it. Otherwise, an empty `Optional<U>` is returned.

Clearly, you can repeat that process if you have more methods or lambdas that yield `Optional` values. You can then build a pipeline of steps, simply by chaining calls to `flatMap`, that will succeed only when all steps do.

For example, consider the safe `inverse` method of the preceding section. Suppose we also have a safe square root:

```
public static Optional<Double> squareRoot(Double x)
{
    return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
}
```

Then you can compute the square root of the inverse as

```
Optional<Double> result = inverse(x).flatMap(MyMath::squareRoot);
```

or, if you prefer,

```
Optional<Double> result = Optional.of(-4.0).flatMap(MyMath::inverse).flatMap(MyMath::squareRoot);
```

If either the `inverse` method or the `squareRoot` returns `Optional.empty()`, the result is empty.

NOTE: You have already seen a `flatMap` method in the `Stream` interface (see Section 1.3, “The `filter`, `map`, and `flatMap` Methods,” on p. 9). That method was used to compose two methods that yield streams, by flattening out the resulting stream of streams. The `Optional.flatMap` method works in the same way if you interpret an optional value as a stream of size zero or one.

The example program in Listing 1.3 demonstrates the `Optional` API.

Listing 1.3 optional/OptionalTest.java

```
1 package optional;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
6 import java.util.*;
7
8 public class OptionalTest
9 {
10     public static void main(String[] args) throws IOException
11     {
```

(Continues)

Listing 1.3 (Continued)

```
12     String contents = new String(Files.readAllBytes(
13         Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
14     List<String> wordList = Arrays.asList(contents.split("\\PL+"));
15
16     Optional<String> optionalValue = wordList.stream()
17         .filter(s -> s.contains("fred"))
18         .findFirst();
19     System.out.println(optionalValue.orElse("No word") + " contains fred");
20
21     Optional<String> optionalString = Optional.empty();
22     String result = optionalString.orElse("N/A");
23     System.out.println("result: " + result);
24     result = optionalString.orElseGet(() -> Locale.getDefault().getDisplayName());
25     System.out.println("result: " + result);
26     try
27     {
28         result = optionalString.orElseThrow(IllegalStateException::new);
29         System.out.println("result: " + result);
30     }
31     catch (Throwable t)
32     {
33         t.printStackTrace();
34     }
35
36     optionalValue = wordList.stream()
37         .filter(s -> s.contains("red"))
38         .findFirst();
39     optionalValue.ifPresent(s -> System.out.println(s + " contains red"));
40
41     Set<String> results = new HashSet<>();
42     optionalValue.ifPresent(results::add);
43     Optional<Boolean> added = optionalValue.map(results::add);
44     System.out.println(added);
45
46     System.out.println(inverse(4.0).flatMap(OptionalTest::squareRoot));
47     System.out.println(inverse(-1.0).flatMap(OptionalTest::squareRoot));
48     System.out.println(inverse(0.0).flatMap(OptionalTest::squareRoot));
49     Optional<Double> result2 = Optional.of(-4.0)
50         .flatMap(OptionalTest::inverse).flatMap(OptionalTest::squareRoot);
51     System.out.println(result2);
52 }
53
54 public static Optional<Double> inverse(Double x)
55 {
56     return x == 0 ? Optional.empty() : Optional.of(1 / x);
57 }
```

```
59     public static Optional<Double> squareRoot(Double x)
60     {
61         return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
62     }
63 }
```

java.util.Optional 8

- <U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)

yields the result of applying `mapper` to the value of this `Optional`, or an empty `Optional` if this `Optional` is empty.

1.8 Collecting Results

When you are done with a stream, you will often want to look at its elements. You can call the `iterator` method, which yields an old-fashioned iterator that you can use to visit the elements.

Alternatively, you can call the `forEach` method to apply a function to each element:

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

On a parallel stream, the `forEach` method traverses elements in arbitrary order. If you want to process them in stream order, call `forEachOrdered` instead. Of course, you might then give up some or all of the benefits of parallelism.

But more often than not, you will want to collect the result in a data structure. You can call `toArray` and get an array of the stream elements.

Since it is not possible to create a generic array at runtime, the expression `stream.toArray()` returns an `Object[]` array. If you want an array of the correct type, pass in the array constructor:

```
String[] result = stream.toArray(String[]::new);
// stream.toArray() has type Object[]
```

For collecting stream elements to another target, there is a convenient `collect` method that takes an instance of the `Collector` interface. The `Collectors` class provides a large number of factory methods for common collectors. To collect a stream into a list or set, simply call

```
List<String> result = stream.collect(Collectors.toList());
```

or

```
Set<String> result = stream.collect(Collectors.toSet());
```

If you want to control which kind of set you get, use the following call instead:

```
TreeSet<String> result = stream.collect(Collectors.toCollection(TreeSet::new));
```

Suppose you want to collect all strings in a stream by concatenating them. You can call

```
String result = stream.collect(Collectors.joining());
```

If you want a delimiter between elements, pass it to the `joining` method:

```
String result = stream.collect(Collectors.joining(", "));
```

If your stream contains objects other than strings, you need to first convert them to strings, like this:

```
String result = stream.map(Object::toString).collect(Collectors.joining(", "));
```

If you want to reduce the stream results to a sum, average, maximum, or minimum, use one of the `summarizing(Int|Long|Double)` methods. These methods take a function that maps the stream objects to a number and yield a result of type `(Int|Long|Double)SummaryStatistics`, simultaneously computing the sum, count, average, minimum, and maximum.

```
IntSummaryStatistics summary = stream.collect(
    Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

java.util.stream.BaseStream 8

- `Iterator<T> iterator()`

yields an iterator for obtaining the elements of this stream. This is a terminal operation.

The example program in Listing 1.4 shows how to collect elements from a stream.

Listing 1.4 collecting/CollectingResults.java

```
1 package collecting;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
```

```
6 import java.util.*;
7 import java.util.stream.*;
8
9 public class CollectingResults
10 {
11     public static Stream<String> noVowels() throws IOException
12     {
13         String contents = new String(Files.readAllBytes(
14             Paths.get("../gutenberg/alice30.txt")),
15             StandardCharsets.UTF_8);
16         List<String> wordList = Arrays.asList(contents.split("\\PL+"));
17         Stream<String> words = wordList.stream();
18         return words.map(s -> s.replaceAll("[aeiouAEIOU]", ""));
19     }
20
21     public static <T> void show(String label, Set<T> set)
22     {
23         System.out.print(label + ": " + set.getClass().getName());
24         System.out.println("["
25             + set.stream().limit(10).map(Object::toString)
26             .collect(Collectors.joining(", ")) + "]");
27     }
28
29     public static void main(String[] args) throws IOException
30     {
31         Iterator<Integer> iter = Stream.iterate(0, n -> n + 1).limit(10)
32             .iterator();
33         while (iter.hasNext())
34             System.out.println(iter.next());
35
36         Object[] numbers = Stream.iterate(0, n -> n + 1).limit(10).toArray();
37         System.out.println("Object array:" + numbers); // Note it's an Object[] array
38
39         try
40         {
41             Integer number = (Integer) numbers[0]; // OK
42             System.out.println("number: " + number);
43             System.out.println("The following statement throws an exception:");
44             Integer[] numbers2 = (Integer[]) numbers; // Throws exception
45         }
46         catch (ClassCastException ex)
47         {
48             System.out.println(ex);
49         }
50
51         Integer[] numbers3 = Stream.iterate(0, n -> n + 1).limit(10)
52             .toArray(Integer[]::new);
53         System.out.println("Integer array: " + numbers3); // Note it's an Integer[] array
```

(Continues)

Listing 1.4 (Continued)

```
54
55     Set<String> noVowelSet = noVowels()
56         .collect(Collectors.toSet());
57     show("noVowelSet", noVowelSet);
58
59     TreeSet<String> noVowelTreeSet = noVowels().collect(
60         Collectors.toCollection(TreeSet::new));
61     show("noVowelTreeSet", noVowelTreeSet);
62
63     String result = noVowels().limit(10).collect(
64         Collectors.joining());
65     System.out.println("Joining: " + result);
66     result = noVowels().limit(10)
67         .collect(Collectors.joining(", "));
68     System.out.println("Joining with commas: " + result);
69
70     IntSummaryStatistics summary = noVowels().collect(
71         Collectors.summarizingInt(String::length));
72     double averageWordLength = summary.getAverage();
73     double maxWordLength = summary.getMax();
74     System.out.println("Average word length: " + averageWordLength);
75     System.out.println("Max word length: " + maxWordLength);
76     System.out.println("forEach:");
77     noVowels().limit(10).forEach(System.out::println);
78 }
79 }
```

java.util.stream.Stream 8

- `void forEach(Consumer<? super T> action)`
invokes `action` on each element of the stream. This is a terminal operation.
- `Object[] toArray()`
- `<A> A[] toArray(IntFunction<A[]> generator)`
yields an array of objects, or of type `A` when passed a constructor reference `A[]::new`.
These are terminal operations.
- `<R,A> R collect(Collector<? super T,A,R> collector)`
collects the elements in this stream, using the given collector. The `Collectors` class
has factory methods for many collectors.

java.util.stream.Collectors 8

- static <T> Collector<T,?,List<T>> toList()
- static <T> Collector<T,?,Set<T>> toSet()

yields collectors that collect elements in a list or set.

- static <T,C extends Collection<T>> Collector<T,?,C> toCollection(Supplier<C> collectionFactory)
yields a collector that collects elements into an arbitrary collection. Pass a constructor reference such as TreeSet::new.

- static Collector<CharSequence,?,String> joining()
- static Collector<CharSequence,?,String> joining(CharSequence delimiter)
- static Collector<CharSequence,?,String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)

yields a collector that joins strings. The delimiter is placed between strings, and the prefix and suffix before the first and after the last string. When not specified, these are empty.

- static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)
- static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)
- static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper)

yields collectors that produce an (Int|Long|Double)SummaryStatistics object, from which you can obtain the count, sum, average, maximum, and minimum of the results of applying `mapper` to each element.

IntSummaryStatistics 8**LongSummaryStatistics 8****DoubleSummaryStatistics 8**

- long getCount()

yields the count of the summarized elements.

- (int|long|double) getSum()
- double getAverage()

yields the sum or average of the summarized elements, or zero if there are no elements.

- (int|long|double) getMax()
- (int|long|double) getMin()

yields the maximum or minimum of the summarized elements, or (Integer|Long|Double).(MAX|MIN)_VALUE if there are no elements.

1.9 Collecting into Maps

Suppose you have a `Stream<Person>` and want to collect the elements into a map so that later you can look up people by their IDs. The `Collectors.toMap` method has two function arguments that produce the map's keys and values. For example,

```
Map<Integer, String> idToName = people.collect(  
    Collectors.toMap(Person::getId, Person::getName));
```

In the common case when the values should be the actual elements, use `Function.identity()` for the second function.

```
Map<Integer, Person> idToPerson = people.collect(  
    Collectors.toMap(Person::getId, Function.identity()));
```

If there is more than one element with the same key, there is a conflict, and the collector will throw an `IllegalStateException`. You can override that behavior by supplying a third function argument that resolves the conflict and determines the value for the key, given the existing and the new value. Your function could return the existing value, the new value, or a combination of them.

Here, we construct a map that contains, for each language in the available locales, its name in your default locale (such as "German") as key, and its localized name (such as "Deutsch") as value.

```
Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());  
Map<String, String> languageNames = locales.collect(  
    Collectors.toMap(  
        Locale::getDisplayLanguage,  
        l -> l.getDisplayLanguage(l),  
        (existingValue, newValue) -> newValue));
```

We don't care that the same language might occur twice (for example, German in Germany and in Switzerland), so we just keep the first entry.

NOTE: In this chapter, we use the `Locale` class as a source of an interesting data set. See Chapter 7 for more information on locales.

Now, suppose we want to know all languages in a given country. Then we need a `Map<String, Set<String>>`. For example, the value for "Switzerland" is the set [French, German, Italian]. At first, we store a singleton set for each language. Whenever a new language is found for a given country, we form the union of the existing and the new set.

```
Map<String, Set<String>> countryLanguageSets = locales.collect(  
    Collectors.toMap(  
        ...))
```

```
Locale:::getDisplayCountry,
l -> Collections.singleton(l.getDisplayLanguage()),
(a, b) ->
{ // Union of a and b
    Set<String> union = new HashSet<>(a);
    union.addAll(b);
    return union;
}));
```

You will see a simpler way of obtaining this map in the next section.

If you want a `TreeMap`, supply the constructor as the fourth argument. You must provide a merge function. Here is one of the examples from the beginning of the section, now yielding a `TreeMap`:

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(
        Person:::getId,
        Function.identity(),
        (existingValue, newValue) -> { throw new IllegalStateException(); },
        TreeMap:::new));
```

NOTE: For each of the `toMap` methods, there is an equivalent `toConcurrentMap` method that yields a concurrent map. A single concurrent map is used in the parallel collection process. When used with a parallel stream, a shared map is more efficient than merging maps. Note that elements are no longer collected in stream order, but that doesn't usually make a difference.

The example program in Listing 1.5 gives examples of collecting stream results into maps.

Listing 1.5 collecting/CollectingIntoMaps.java

```
1 package collecting;
2
3 import java.io.*;
4 import java.util.*;
5 import java.util.function.*;
6 import java.util.stream.*;
7
8 public class CollectingIntoMaps
9 {
10     public static class Person
11     {
12         private int id;
13         private String name;
```

(Continues)

Listing 1.5 (Continued)

```
14
15     public Person(int id, String name)
16     {
17         this.id = id;
18         this.name = name;
19     }
20
21     public int getId()
22     {
23         return id;
24     }
25
26     public String getName()
27     {
28         return name;
29     }
30
31     public String toString()
32     {
33         return getClass().getName() + "[id=" + id + ",name=" + name + "]";
34     }
35 }
36
37     public static Stream<Person> people()
38     {
39         return Stream.of(new Person(1001, "Peter"), new Person(1002, "Paul"),
40                         new Person(1003, "Mary"));
41     }
42
43     public static void main(String[] args) throws IOException
44     {
45         Map<Integer, String> idToName = people().collect(
46             Collectors.toMap(Person::getId, Person::getName));
47         System.out.println("idToName: " + idToName);
48
49         Map<Integer, Person> idToPerson = people().collect(
50             Collectors.toMap(Person::getId, Function.identity()));
51         System.out.println("idToPerson: " + idToPerson.getClass().getName()
52                           + idToPerson);
53
54         idToPerson = people().collect(
55             Collectors.toMap(Person::getId, Function.identity(), (
56                 existingValue, newValue) -> {
57                     throw new IllegalStateException();
58                 }, TreeMap::new));
59         System.out.println("idToPerson: " + idToPerson.getClass().getName()
60                           + idToPerson);
61 }
```

```
62     Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
63     Map<String, String> languageNames = locales.collect(
64         Collectors.toMap(
65             Locale::getDisplayLanguage,
66             l -> l.getDisplayLanguage(),
67             (existingValue, newValue) -> newValue));
68     System.out.println("languageNames: " + languageNames);
69
70     locales = Stream.of(Locale.getAvailableLocales());
71     Map<String, Set<String>> countryLanguageSets = locales.collect(
72         Collectors.toMap(
73             Locale::getDisplayCountry,
74             l -> Collections.singleton(l.getDisplayLanguage()),
75             (a, b) -> { // union of a and b
76                 Set<String> union = new HashSet<>(a);
77                 union.addAll(b);
78                 return union;
79             });
80     System.out.println("countryLanguageSets: " + countryLanguageSets);
81 }
82 }
```

java.util.stream.Collectors 8

- static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)
- static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)
- static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)
- static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)
- static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)
- static <T,K,U,M extends ConcurrentHashMap<K,U>> Collector<T,?,M> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)

yields a collector that produces a map or concurrent map. The keyMapper and valueMapper functions are applied to each collected element, yielding a key/value entry of the resulting map. By default, an IllegalStateException is thrown when two elements give rise to the same key. You can instead supply a mergeFunction that merges values with the same key. By default, the result is a HashMap or ConcurrentHashMap. You can instead supply a mapSupplier that yields the desired map instance.

1.10 Grouping and Partitioning

In the preceding section, you saw how to collect all languages in a given country. But the process was a bit tedious. You had to generate a singleton set for each map value and then specify how to merge the existing and new values. Forming groups of values with the same characteristic is very common, and the `groupingBy` method supports it directly.

Let's look at the problem of grouping locales by country. First, form this map:

```
Map<String, List<Locale>> countryToLocales = locales.collect(  
    Collectors.groupingBy(Locale::getCountry));
```

The function `Locale::getCountry` is the *classifier function* of the grouping. You can now look up all locales for a given country code, for example

```
List<Locale> swissLocales = countryToLocales.get("CH");  
// Yields locales [it_CH, de_CH, fr_CH]
```

NOTE: A quick refresher on locales: Each locale has a language code (such as `en` for English) and a country code (such as `US` for the United States). The locale `en_US` describes English in the United States, and `en_IE` is English in Ireland. Some countries have multiple locales. For example, `ga_IE` is Gaelic in Ireland, and, as the preceding example shows, my JVM knows three locales in Switzerland.

When the classifier function is a predicate function (that is, a function returning a `boolean` value), the stream elements are partitioned into two lists: those where the function returns `true` and the complement. In this case, it is more efficient to use `partitioningBy` instead of `groupingBy`. For example, here we split all locales into those that use English and all others:

```
Map<Boolean, List<Locale>> englishAndOtherLocales = locales.collect(  
    Collectors.partitioningBy(l -> l.getLanguage().equals("en")));  
List<Locale> englishLocales = englishAndOtherLocales.get(true);
```

NOTE: If you call the `groupingByConcurrent` method, you get a concurrent map that, when used with a parallel stream, is concurrently populated. This is entirely analogous to the `toConcurrentMap` method.

java.util.stream.Collectors 8

- static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)
- static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier)

yields a collector that produces a map or concurrent map whose keys are the results of applying classifier to all collected elements, and whose values are lists of elements with the same key.

- static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)

yields a collector that produces a map whose keys are true / false, and whose values are lists of the elements that fulfill/do not fulfill the predicate.

1.11 Downstream Collectors

The `groupingBy` method yields a map whose values are lists. If you want to process those lists in some way, supply a “downstream collector.” For example, if you want sets instead of lists, you can use the `Collectors.toSet` collector that you saw in the previous section:

```
Map<String, Set<Locale>> countryToLocaleSet = locales.collect(  
    groupingBy(Locale::getCountry, toSet()));
```

NOTE: In this and the remaining examples of this section, we assume a static import of `java.util.stream.Collectors.*` to make the expressions easier to read.

Several collectors are provided for reducing grouped elements to numbers:

- `counting` produces a count of the collected elements. For example,

```
Map<String, Long> countryToLocaleCounts = locales.collect(  
    groupingBy(Locale::getCountry, counting()));
```

counts how many locales there are for each country.

- `summing(Int|Long|Double)` takes a function argument, applies the function to the downstream elements, and produces their sum. For example,

```
Map<String, Integer> stateToCityPopulation = cities.collect(  
    groupingBy(City::getState, summingInt(City::getPopulation)));
```

computes the sum of populations per state in a stream of cities.

- `maxBy` and `minBy` take a comparator and produce maximum and minimum of the downstream elements. For example,

```
Map<String, Optional<City>> stateToLargestCity = cities.collect(  
    groupingBy(City::getState,  
        maxBy(Comparator.comparing(City::getPopulation))));
```

produces the largest city per state.

The `mapping` method yields a collector that applies a function to downstream results and passes the function values to yet another collector. For example,

```
Map<String, Optional<String>> stateToLongestCityName = cities.collect(  
    groupingBy(City::getState,  
        mapping(City::getName,  
            maxBy(Comparator.comparing(String::length))));
```

Here, we group cities by state. Within each state, we produce the names of the cities and reduce by maximum length.

The `mapping` method also yields a nicer solution to a problem from the preceding section—gathering a set of all languages in a country.

```
Map<String, Set<String>> countryToLanguages = locales.collect(  
    groupingBy(Locale::getDisplayCountry,  
        mapping(Locale::getDisplayLanguage,  
            toSet())));
```

In the previous section, we used `toMap` instead of `groupingBy`. In this form, you don't need to worry about combining the individual sets.

If the grouping or mapping function has return type `int`, `long`, or `double`, you can collect elements into a summary statistics object, as discussed in Section 1.8, "Collecting Results," on p. 19. For example,

```
Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities.collect(  
    groupingBy(City::getState,  
        summarizingInt(City::getPopulation)));
```

Then you can get the sum, count, average, minimum, and maximum of the function values from the summary statistics objects of each group.

NOTE: There are also three versions of a reducing method that apply general reductions, as described in Section 1.12, "Reduction Operations," on p. 33.

Composing collectors is a powerful approach, but it can also lead to very convoluted expressions. Their best use is with `groupingBy` or `partitioningBy` to process the

“downstream” map values. Otherwise, simply apply methods such as `map`, `reduce`, `count`, `max`, or `min` directly on streams.

The example program in Listing 1.6 demonstrates downstream collectors.

Listing 1.6 collecting/DownstreamCollectors.java

```
1 package collecting;
2
3 import static java.util.stream.Collectors.*;
4
5 import java.io.*;
6 import java.nio.file.*;
7 import java.util.*;
8 import java.util.stream.*;
9
10 public class DownstreamCollectors
11 {
12
13     public static class City
14     {
15         private String name;
16         private String state;
17         private int population;
18
19         public City(String name, String state, int population)
20         {
21             this.name = name;
22             this.state = state;
23             this.population = population;
24         }
25
26         public String getName()
27         {
28             return name;
29         }
30
31         public String getState()
32         {
33             return state;
34         }
35
36         public int getPopulation()
37         {
38             return population;
39         }
40     }
41 }
```

(Continues)

Listing 1.6 (Continued)

```
42     public static Stream<City> readCities(String filename) throws IOException
43     {
44         return Files.lines(Paths.get(filename)).map(l -> l.split(", "))
45             .map(a -> new City(a[0], a[1], Integer.parseInt(a[2])));
46     }
47
48     public static void main(String[] args) throws IOException
49     {
50         Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
51         locales = Stream.of(Locale.getAvailableLocales());
52         Map<String, Set<Locale>> countryToLocaleSet = locales.collect(groupingBy(
53             Locale::getCountry, toSet()));
54         System.out.println("countryToLocaleSet: " + countryToLocaleSet);
55
56         locales = Stream.of(Locale.getAvailableLocales());
57         Map<String, Long> countryToLocaleCounts = locales.collect(groupingBy(
58             Locale::getCountry, counting()));
59         System.out.println("countryToLocaleCounts: " + countryToLocaleCounts);
60
61         Stream<City> cities = readCities("cities.txt");
62         Map<String, Integer> stateToCityPopulation = cities.collect(groupingBy(
63             City::getState, summingInt(City::getPopulation)));
64         System.out.println("stateToCityPopulation: " + stateToCityPopulation);
65
66         cities = readCities("cities.txt");
67         Map<String, Optional<String>> stateToLongestCityName = cities
68             .collect(groupingBy(
69                 City::getState,
70                 mapping(City::getName,
71                     maxBy(Comparator.comparing(String::length)))));
72
73         System.out.println("stateToLongestCityName: " + stateToLongestCityName);
74
75         locales = Stream.of(Locale.getAvailableLocales());
76         Map<String, Set<String>> countryToLanguages = locales.collect(groupingBy(
77             Locale::getDisplayCountry,
78             mapping(Locale::getDisplayLanguage, toSet())));
79         System.out.println("countryToLanguages: " + countryToLanguages);
80
81         cities = readCities("cities.txt");
82         Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities
83             .collect(groupingBy(City::getState,
84                 summarizingInt(City::getPopulation)));
85         System.out.println(stateToCityPopulationSummary.get("NY"));
86
87         cities = readCities("cities.txt");
```

```
88     Map<String, String> stateToCityNames = cities.collect(groupingBy(
89         City::getState,
90         reducing("", City::getName, (s, t) -> s.length() == 0 ? t : s
91             + ", " + t)));
92
93     cities = readCities("cities.txt");
94     stateToCityNames = cities.collect(groupingBy(City::getState,
95         mapping(City::getName, joining(", "))));
96     System.out.println("stateToCityNames: " + stateToCityNames);
97 }
98 }
```

java.util.stream.Collectors 8

- static <T> Collector<T,?,Long> counting()
yields a collector that counts the collected elements.
- static <T> Collector<T,?,Integer> summingInt(ToIntFunction<? super T> mapper)
- static <T> Collector<T,?,Long> summingLong(ToLongFunction<? super T> mapper)
- static <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper)
yields a collector that computes the sum of the values of applying `mapper` to the collected elements.
- static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator)
- static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator)
yields a collector that computes the maximum or minimum of the collected elements, using the ordering specified by `comparator`.
- static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,> mapper, Collector<? super U,A,R> downstream)
yields a collector that produces a map whose keys are `mapper` applied to the collected elements, and whose values are the result of collecting the elements with the same key using the `downstream` collector.

1.12 Reduction Operations

The `reduce` method is a general mechanism for computing a value from a stream. The simplest form takes a binary function and keeps applying it, starting with the first two elements. It's easy to explain this if the function is the sum:

```
List<Integer> values = . . .;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

In this case, the `reduce` method computes $v_0 + v_1 + v_2 + \dots$, where the v_i are the stream elements. The method returns an `Optional` because there is no valid result if the stream is empty.

NOTE: In this case, you can write `reduce(Integer::sum)` instead of `reduce((x, y) -> x + y)`.

In general, if the `reduce` method has a reduction operation op , the reduction yields $v_0 op v_1 op v_2 op \dots$, where we write $v_i op v_{i+1}$ for the function call $op(v_i, v_{i+1})$. The operation should be *associative*: It shouldn't matter in which order you combine the elements. In math notation, $(x op y) op z$ must be equal to $x op (y op z)$. This allows efficient reduction with parallel streams.

There are many associative operations that might be useful in practice, such as sum, product, string concatenation, maximum and minimum, set union and intersection. An example of an operation that is not associative is subtraction. For example, $(6 - 3) - 2 \neq 6 - (3 - 2)$.

Often, there is an *identity value* e such that $e op x = x$, and you can use that element as the start of the computation. For example, 0 is the identity value for addition. Then call the second form of `reduce`:

```
List<Integer> values = . . .;
Integer sum = values.stream().reduce(0, (x, y) -> x + y)
    // Computes 0 + v0 + v1 + v2 + . . .
```

The identity value is returned if the stream is empty, and you no longer need to deal with the `Optional` class.

Now suppose you have a stream of objects and want to form the sum of some property, such as all lengths in a stream of strings. You can't use the simple form of `reduce`. It requires a function $(T, T) \rightarrow T$, with the same types for the arguments and the result. But in this situation, you have two types: The stream elements have type `String`, and the accumulated result is an integer. There is a form of `reduce` that can deal with this situation.

First, you supply an “accumulator” function $(total, word) \rightarrow total + word.length()$. That function is called repeatedly, forming the cumulative total. But when the computation is parallelized, there will be multiple computations of this kind, and you need to combine their results. You supply a second function for that purpose. The complete call is

```
int result = words.reduce(0,
    (total, word) -> total + word.length(),
    (total1, total2) -> total1 + total2);
```

NOTE: In practice, you probably won't use the `reduce` method a lot. It is usually easier to map to a stream of numbers and use one of its methods to compute sum, max, or min. (We discuss streams of numbers in Section 1.13, "Primitive Type Streams," on p. 36.) In this particular example, you could have called `words.mapToInt(String::length).sum()`, which is both simpler and more efficient since it doesn't involve boxing.

NOTE: There are times when `reduce` is not general enough. For example, suppose you want to collect the results in a `BitSet`. If the collection is parallelized, you can't put the elements directly into a single `BitSet` because a `BitSet` object is not threadsafe. For that reason, you can't use `reduce`. Each segment needs to start out with its own empty set, and `reduce` only lets you supply one identity value. Instead, use `collect`. It takes three arguments:

1. A *supplier* that makes new instances of the target type, for example a constructor for a hash set.
2. An *accumulator* that adds an element to an instance, such as an `add` method.
3. A *combiner* that merges two instances into one, such as `addAll`.

Here is how the `collect` method works for a bit set:

```
BitSet result = stream.collect(BitSet::new, BitSet::set, BitSet::or);
```

java.util.Stream 8

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
- `T reduce(T identity, BinaryOperator<T> accumulator)`
- `<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`
forms a cumulative total of the stream elements with the given accumulator function. If `identity` is provided, then it is the first value to be accumulated. If `combiner` is provided, it can be used to combine totals of segments that are accumulated separately.
- `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`
collects elements in a result of type `R`. On each segment, `supplier` is called to provide an initial result, `accumulator` is called to mutably add elements to it, and `combiner` is called to combine two results.

1.13 Primitive Type Streams

So far, we have collected integers in a `Stream<Integer>`, even though it is clearly inefficient to wrap each integer into a wrapper object. The same is true for the other primitive types: `double`, `float`, `long`, `short`, `char`, `byte`, and `boolean`. The stream library has specialized types `IntStream`, `LongStream`, and `DoubleStream` that store primitive values directly, without using wrappers. If you want to store `short`, `char`, `byte`, and `boolean`, use an `IntStream`, and for `float`, use a `DoubleStream`.

To create an `IntStream`, call the `IntStream.of` and `Arrays.stream` methods:

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to); // values is an int[] array
```

As with object streams, you can also use the static `generate` and `iterate` methods. In addition, `IntStream` and `LongStream` have static methods `range` and `rangeClosed` that generate integer ranges with step size one:

```
IntStream zeroToNinetyNine = IntStream.range(0, 100); // Upper bound is excluded
IntStream zeroToHundred = IntStream.rangeClosed(0, 100); // Upper bound is included
```

The `CharSequence` interface has methods `codePoints` and `chars` that yield an `IntStream` of the Unicode codes of the characters or of the code units in the UTF-16 encoding. (See Chapter 2 for the sordid details.)

```
String sentence = "\uD835\uDD46 is the set of octonions.";
// \uD835\uDD46 is the UTF-16 encoding of the letter ☈, unicode U+1D546

IntStream codes = sentence.codePoints();
// The stream with hex values 1D546 20 69 73 20 . . .
```

When you have a stream of objects, you can transform it to a primitive type stream with the `mapToInt`, `mapToLong`, or `mapToDouble` methods. For example, if you have a stream of strings and want to process their lengths as integers, you might as well do it in an `IntStream`:

```
Stream<String> words = . . .;
IntStream lengths = words.mapToInt(String::length);
```

To convert a primitive type stream to an object stream, use the `boxed` method:

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

Generally, the methods on primitive type streams are analogous to those on object streams. Here are the most notable differences:

- The `toArray` methods return primitive type arrays.

- Methods that yield an optional result return an `OptionalInt`, `OptionalLong`, or `OptionalDouble`. These classes are analogous to the `Optional` class but have methods `getAsInt`, `getAsLong`, and `getAsDouble` instead of the `get` method.
- There are methods `sum`, `average`, `max`, and `min` that return the sum, average, maximum, and minimum. These methods are not defined for object streams.
- The `summaryStatistics` method yields an object of type `IntSummaryStatistics`, `LongSummaryStatistics`, or `DoubleSummaryStatistics` that can simultaneously report the sum, average, maximum, and minimum of the stream.

NOTE: The `Random` class has methods `ints`, `longs`, and `doubles` that return primitive type streams of random numbers.

The program in Listing 1.7 gives examples for the API of primitive type streams.

Listing 1.7 streams/PrimitiveTypeStreams.java

```
1 package streams;
2
3 import java.io.IOException;
4 import java.nio.charset.StandardCharsets;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8 import java.util.stream.Collectors;
9 import java.util.stream.IntStream;
10 import java.util.stream.Stream;
11
12 public class PrimitiveTypeStreams
13 {
14     public static void show(String title, IntStream stream)
15     {
16         final int SIZE = 10;
17         int[] firstElements = stream.limit(SIZE + 1).toArray();
18         System.out.print(title + ": ");
19         for (int i = 0; i < firstElements.length; i++)
20         {
21             if (i > 0) System.out.print(", ");
22             if (i < SIZE) System.out.print(firstElements[i]);
23             else System.out.print("...");
24         }
25         System.out.println();
26     }
}
```

(Continues)

Listing 1.7 (Continued)

```
27
28     public static void main(String[] args) throws IOException
29     {
30         IntStream is1 = IntStream.generate(() -> (int) (Math.random() * 100));
31         show("is1", is1);
32         IntStream is2 = IntStream.range(5, 10);
33         show("is2", is2);
34         IntStream is3 = IntStream.rangeClosed(5, 10);
35         show("is3", is3);
36
37         Path path = Paths.get("../gutenberg/alice30.txt");
38         String contents = new String(Files.readAllBytes(path), StandardCharsets.UTF_8);
39
40         Stream<String> words = Stream.of(contents.split("\\PL+"));
41         IntStream is4 = words.mapToInt(String::length);
42         show("is4", is4);
43         String sentence = "\uD835\uDD46 is the set of octonions.";
44         System.out.println(sentence);
45         IntStream codes = sentence.codePoints();
46         System.out.println(codes.mapToObj(c -> String.format("%X ", c)).collect(
47             Collectors.joining()));
48
49         Stream<Integer> integers = IntStream.range(0, 100).boxed();
50         IntStream is5 = integers.mapToInt(Integer::intValue);
51         show("is5", is5);
52     }
53 }
```

java.util.stream.IntStream 8

- `static IntStream range(int startInclusive, int endExclusive)`
- `static IntStream rangeClosed(int startInclusive, int endInclusive)`
yields an `IntStream` with the integers in the given range.
- `static IntStream of(int... values)`
yields an `IntStream` with the given elements.
- `int[] toArray()`
yields an array with the elements of this stream.

(Continues)

java.util.stream.IntStream 8 (Continued)

- int sum()
- OptionalDouble average()
- OptionalInt max()
- OptionalInt min()
- IntSummaryStatistics summaryStatistics()

yields the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these results can be obtained.

- Stream<Integer> boxed()
yields a stream of wrapper objects for the elements in this stream.

java.util.stream.LongStream 8

- static LongStream range(long startInclusive, long endExclusive)
- static LongStream rangeClosed(long startInclusive, long endInclusive)

yields a LongStream with the integers in the given range.

- static LongStream of(long... values)
yields a LongStream with the given elements.
- long[] toArray()
yields an array with the elements of this stream.

- long sum()
- OptionalDouble average()
- OptionalLong max()
- OptionalLong min()
- LongSummaryStatistics summaryStatistics()

yields the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these results can be obtained.

- Stream<Long> boxed()
yields a stream of wrapper objects for the elements in this stream.

java.util.stream.DoubleStream 8

- static DoubleStream of(double... values)
yields a DoubleStream with the given elements.

(Continues)

java.util.stream.DoubleStream 8 (Continued)

- `double[] toArray()`
yields an array with the elements of this stream.
- `double sum()`
- `OptionalDouble average()`
- `OptionalDouble max()`
- `OptionalDouble min()`
- `DoubleSummaryStatistics summaryStatistics()`
yields the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these results can be obtained.
- `Stream<Double> boxed()`
yields a stream of wrapper objects for the elements in this stream.

java.lang(CharSequence 1.0

- `IntStream codePoints() 8`
yields a stream of all Unicode code points of this string.

java.util.Random 1.0

- `IntStream ints()`
- `IntStream ints(int randomNumberOrigin, int randomNumberBound) 8`
- `IntStream ints(long streamSize) 8`
- `IntStream ints(long streamSize, int randomNumberOrigin, int randomNumberBound) 8`
- `LongStream longs()` 8
- `LongStream longs(long randomNumberOrigin, long randomNumberBound) 8`
- `LongStream longs(long streamSize) 8`
- `LongStream longs(long streamSize, long randomNumberOrigin, long randomNumberBound) 8`
- `DoubleStream doubles()` 8
- `DoubleStream doubles(double randomNumberOrigin, double randomNumberBound) 8`
- `DoubleStream doubles(long streamSize) 8`
- `DoubleStream doubles(long streamSize, double randomNumberOrigin, double randomNumberBound) 8`
yields streams of random numbers. If `streamSize` is provided, the stream is finite with the given number of elements. When bounds are provided, the elements are between `randomNumberOrigin` (inclusive) and `randomNumberBound` (exclusive).

java.util.Optional(Int|Long|Double) 8

- static Optional(Int|Long|Double) of((int|long|double) value)
yields an optional object with the supplied primitive type value.
- (int|long|double) getAs(Int|Long|Double)()
yields the value of this optional object, or throws a NoSuchElementException if it is empty.
- (int|long|double) orElse((int|long|double) other)
- (int|long|double) orElseGet((Int|Long|Double)Supplier other)
yields the value of this optional object, or the alternative value if this object is empty.
- void ifPresent((Int|Long|Double)Consumer consumer)
If this optional object is not empty, passes its value to consumer.

java.util.(Int|Long|Double)SummaryStatistics 8

- long getCount()
- (int|long|double) getSum()
- double getAverage()
- (int|long|double) getMax()
- (int|long|double) getMin()

yields the count, sum, average, maximum, and minimum of the collected elements.

1.14 Parallel Streams

Streams make it easy to parallelize bulk operations. The process is mostly automatic, but you need to follow a few rules. First of all, you must have a parallel stream. You can get a parallel stream from any collection with the `Collection.parallelStream()` method:

```
Stream<String> parallelWords = words.parallelStream();
```

Moreover, the `parallel` method converts any sequential stream into a parallel one.

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

As long as the stream is in parallel mode when the terminal method executes, all intermediate stream operations will be parallelized.

When stream operations run in parallel, the intent is that the same result is returned as if they had run serially. It is important that the operations can be executed in an arbitrary order.

Here is an example of something you cannot do. Suppose you want to count all short words in a stream of strings:

```
int[] shortWords = new int[12];
words.parallelStream().forEach(
    s -> { if (s.length() < 12) shortWords[s.length()]++; });
// Error-race condition!
System.out.println(Arrays.toString(shortWords));
```

This is very, very bad code. The function passed to `forEach` runs concurrently in multiple threads, each updating a shared array. As we explained in Chapter 14 of Volume 1, that's a classic *race condition*. If you run this program multiple times, you are quite likely to get a different sequence of counts in each run—each of them wrong.

It is your responsibility to ensure that any functions you pass to parallel stream operations are safe to execute in parallel. The best way to do that is to stay away from mutable state. In this example, you can safely parallelize the computation if you group strings by length and count them.

```
Map<Integer, Long> shortWordCounts =
    words.parallelStream()
        .filter(s -> s.length() < 10)
        .collect(groupingBy(
            String::length,
            counting()));
```



CAUTION: The functions that you pass to parallel stream operations should not block. Parallel streams use a fork-join pool for operating on segments of the stream. If multiple stream operations block, the pool may not be able to do any work.

By default, streams that arise from ordered collections (arrays and lists), from ranges, generators, and iterators, or from calling `Stream.sorted`, are *ordered*. Results are accumulated in the order of the original elements, and are entirely predictable. If you run the same operations twice, you will get exactly the same results.

Ordering does not preclude efficient parallelization. For example, when computing `stream.map(fun)`, the stream can be partitioned into n segments, each of which is concurrently processed. Then the results are reassembled in order.

Some operations can be more effectively parallelized when the ordering requirement is dropped. By calling the `unordered` method on a stream, you indicate that you are not interested in ordering. One operation that can benefit from this is `Stream.distinct`. On an ordered stream, `distinct` retains the first of all equal elements. That impedes parallelization—the thread processing a segment can't know which

elements to discard until the preceding segment has been processed. If it is acceptable to retain *any* of the unique elements, all segments can be processed concurrently (using a shared set to track duplicates).

You can also speed up the `limit` method by dropping ordering. If you just want any `n` elements from a stream and you don't care which ones you get, call

```
Stream<String> sample = words.parallelStream().unordered().limit(n);
```

As discussed in Section 1.9, “Collecting into Maps,” on p. 24, merging maps is expensive. For that reason, the `Collectors.groupingByConcurrent` method uses a shared concurrent map. To benefit from parallelism, the order of the map values will not be the same as the stream order.

```
Map<Integer, List<String>> result = words.parallelStream().collect(  
    Collectors.groupingByConcurrent(String::length));  
    // Values aren't collected in stream order
```

Of course, you won't care if you use a downstream collector that is independent of the ordering, such as

```
Map<Integer, Long> wordCounts =  
    words.parallelStream()  
    .collect(  
        groupingByConcurrent(  
            String::length,  
            counting()));
```



CAUTION: It is very important that you don't modify the collection that is backing a stream while carrying out a stream operation (even if the modification is threadsafe). Remember that streams don't collect their data—that data is always in a separate collection. If you were to modify that collection, the outcome of the stream operations would be undefined. The JDK documentation refers to this requirement as *noninterference*. It applies both to sequential and parallel streams.

To be exact, since intermediate stream operations are lazy, it is possible to mutate the collection up to the point when the terminal operation executes. For example, the following, while certainly not recommended, will work:

```
List<String> wordList = . . .;  
Stream<String> words = wordList.stream();  
wordList.add("END");  
long n = words.distinct().count();
```

But this code is wrong:

```
Stream<String> words = wordList.stream();  
words.forEach(s -> if (s.length() < 12) wordList.remove(s));  
    // Error-interference
```

For parallel streams to work well, a number of conditions need to be fulfilled:

- The data should be in memory. It would be inefficient to have to wait for the data to arrive.
- The stream should be efficiently splittable into subregions. A stream backed by an array or a balanced binary tree works well, but the result of `Stream.iterate` does not.
- The stream operations should do a substantial amount of work. If the total work load is not large, it does not make sense to pay for the cost of setting up the parallel computation.
- The stream operations should not block.

In other words, don't turn all your streams into parallel streams. Use parallel streams only when you do a substantial amount of sustained computational work on data that is already in memory.

The example program in Listing 1.8 demonstrates how to work with parallel streams.

Listing 1.8 parallel/ParallelStreams.java

```
1 package parallel;
2
3 import static java.util.stream.Collectors.*;
4
5 import java.io.*;
6 import java.nio.charset.*;
7 import java.nio.file.*;
8 import java.util.*;
9 import java.util.stream.*;
10
11 public class ParallelStreams
12 {
13     public static void main(String[] args) throws IOException
14     {
15         String contents = new String(Files.readAllBytes(
16             Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
17         List<String> wordList = Arrays.asList(contents.split("\\PL+"));
18
19         // Very bad code ahead
20         int[] shortWords = new int[10];
21         wordList.parallelStream().forEach(s ->
22             {
23                 if (s.length() < 10) shortWords[s.length()]++;
24             });
25     }
26 }
```

```
25 System.out.println(Arrays.toString(shortWords));
26
27 // Try again--the result will likely be different (and also wrong)
28 Arrays.fill(shortWords, 0);
29 wordList.parallelStream().forEach(s ->
30 {
31     if (s.length() < 10) shortWords[s.length()]++;
32 });
33 System.out.println(Arrays.toString(shortWords));
34
35 // Remedy: Group and count
36 Map<Integer, Long> shortWordCounts = wordList.parallelStream()
37     .filter(s -> s.length() < 10)
38     .collect(groupingBy(String::length, counting()));
39
40 System.out.println(shortWordCounts);
41
42 // Downstream order not deterministic
43 Map<Integer, List<String>> result = wordList.parallelStream().collect(
44     Collectors.groupingByConcurrent(String::length));
45
46 System.out.println(result.get(14));
47
48 result = wordList.parallelStream().collect(
49     Collectors.groupingByConcurrent(String::length));
50
51 System.out.println(result.get(14));
52
53 Map<Integer, Long> wordCounts = wordList.parallelStream().collect(
54     groupingByConcurrent(String::length, counting()));
55
56 System.out.println(wordCounts);
57 }
58 }
```

java.util.stream.BaseStream<T,S extends BaseStream<T,S>> 8

- `S parallel()`
yields a parallel stream with the same elements as this stream.
- `S unordered()`
yields an unordered stream with the same elements as this stream.

java.util.Collection<E> 1.2

- `Stream<E> parallelStream()` 8

yields a parallel stream with the elements of this collection.

In this chapter, you have learned how to put the stream library of Java 8 to use. The next chapter covers another important topic: processing input and output.

CHAPTER

2

Input and Output

In this chapter

- 2.1 Input/Output Streams, page 48
- 2.2 Text Input and Output, page 60
- 2.3 Reading and Writing Binary Data, page 69
- 2.4 Object Input/Output Streams and Serialization, page 80
- 2.5 Working with Files, page 100
- 2.6 Memory-Mapped Files, page 116
- 2.7 Regular Expressions, page 128

In this chapter, we will 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` form 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 for processing Unicode characters that inherit from the abstract `Reader` and `Writer` classes. 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. These methods call the abstract `read` method, so subclasses need to override only one method.

Similarly, the `OutputStream` class defines the abstract method

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

which writes one byte to an output location.

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:

```
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
    byte[] 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 stream classes derived from the basic `InputStream` and `OutputStream` classes.

java.io.InputStream 1.0

- `abstract int read()`
reads a byte of data and returns the byte read; returns -1 at the end of the input stream.
- `int read(byte[] b)`
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.

(Continues)

java.io.InputStream 1.0 (Continued)

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

reads into an array of bytes and returns the actual number of bytes read, or -1 at the end of the input stream.

Parameters: `b` The array into which the data is read
 `off` The offset into `b` where the first bytes should be placed
 `len` The maximum number of bytes to read

- `long skip(long n)`

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).

- `int available()`

returns the number of bytes available, without blocking (recall that blocking means that the current thread loses its turn).

- `void close()`

closes the input stream.

- `void mark(int readlimit)`

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.

- `void reset()`

returns to the last marker. Subsequent calls to `read` reread the bytes. If there is no current marker, the input stream is not reset.

- `boolean markSupported()`

returns `true` if the input stream supports marking.

java.io.OutputStream 1.0

- `abstract void write(int n)`

writes a byte of data.

(Continues)

java.io.OutputStream 1.0 (Continued)

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

writes all bytes or a range of bytes in the array b.

Parameters: b The array from which to write the data

 off The offset into b to the first byte that will be written

 len The number of bytes to write

- void close()

flushes and closes the output stream.

- void flush()

flushes the output stream—that is, sends any buffered data to its destination.

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`.

```
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.)

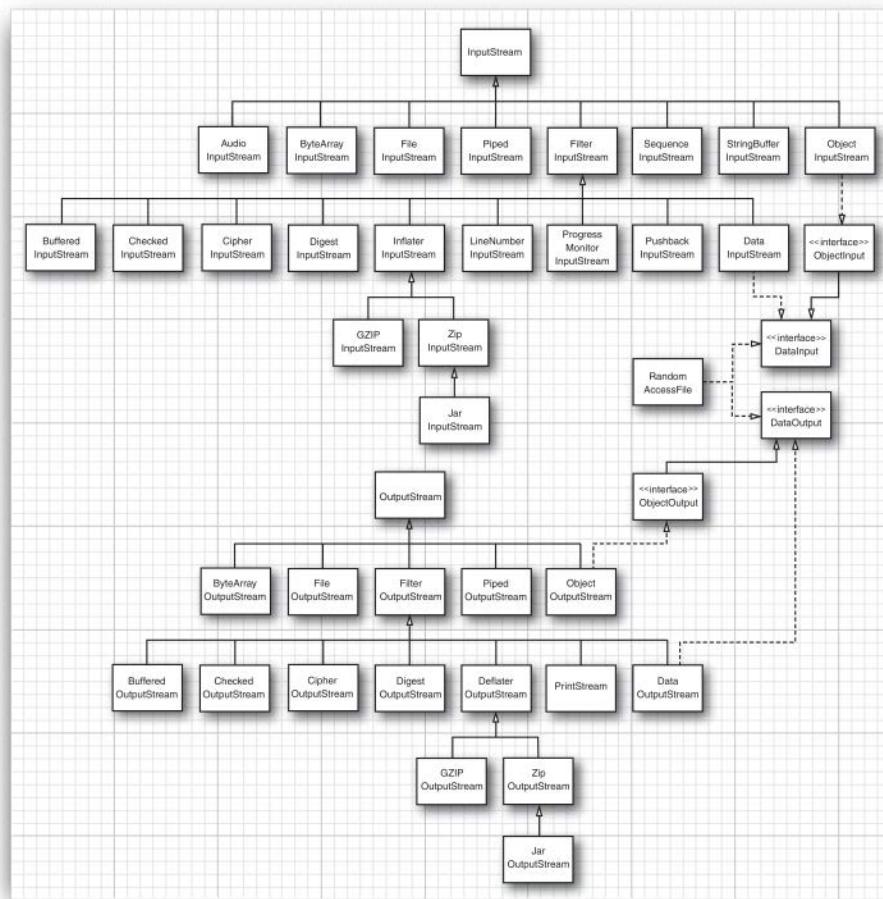


Figure 2.1 Input and output stream hierarchy

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

```
void close() throws IOException
```

and

```
void flush()
```

respectively. The classes `InputStream`, `OutputStream`, `Reader`, and `Writer` all implement the `Closeable` interface.

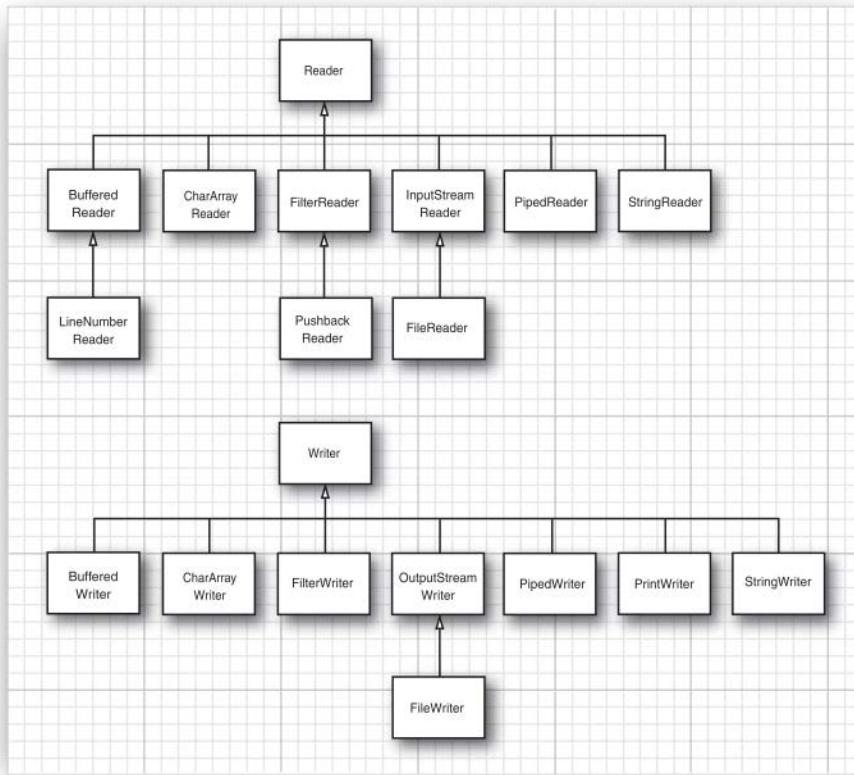


Figure 2.2 Reader and writer hierarchy

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 2.6.2, “The Buffer Data Structure,” on p. 124 for details.)

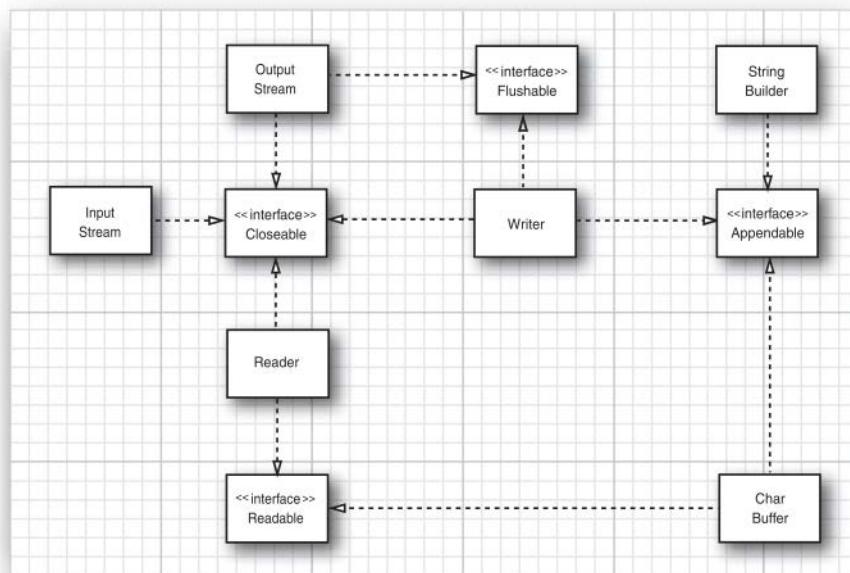


Figure 2.3 The `Closeable`, `Flushable`, `Readable`, and `Appendable` interfaces

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.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,

```
FileInputStream 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`.

```
FileInputStream fin = new FileInputStream("employee.dat");  
DataInputStream din = new DataInputStream(fin);  
double x = din.readDouble();
```

If you look at Figure 2.1 again, you can see the classes `FilterInputStream` and `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 `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 *and* the data input methods for a file, you need to use the following rather monstrous sequence of constructors:

```
DataInputStream din = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("employee.dat")));
```

Notice that we put the `DataInputStream` *last* in the chain of constructors because we want to use the `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 `PushbackInputStream` for this purpose.

```
PushbackInputStream pbin = new PushbackInputStream(  
    new BufferedInputStream(  
        new FileInputStream("employee.dat")));
```

Now you can speculatively read the next byte

```
int b = pbin.read();
```

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

```
if (b != '<') pbin.unread(b);
```

However, reading and unreading are the *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.

```
DataInputStream din = new DataInputStream(  
    pbin = new PushbackInputStream(  
        new BufferedInputStream(  
            new FileInputStream("employee.dat"))));
```

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 truly 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):

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

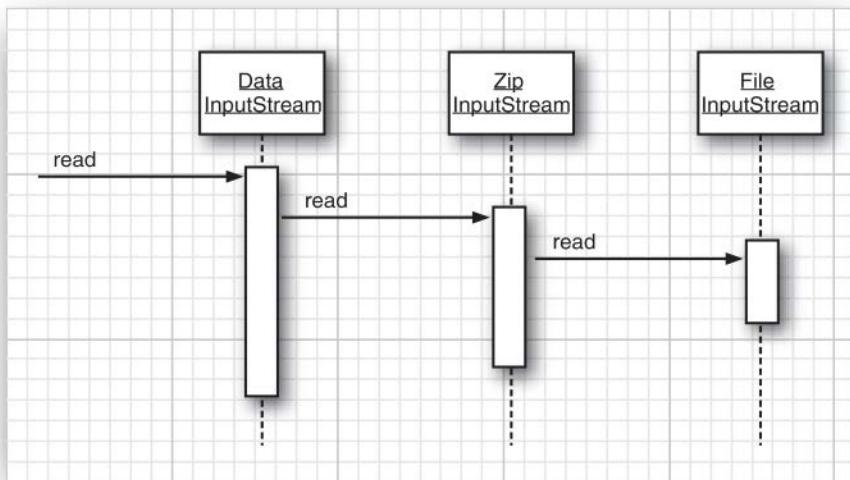


Figure 2.4 A sequence of filtered input streams

(See Section 2.3.3, “ZIP Archives,” on p. 77 for more on Java’s handling of ZIP files.)

java.io.FileInputStream 1.0

- `FileInputStream(String name)`
- `FileInputStream(File file)`

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`.

Parameters: b The byte to be read again

2.2 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 2.3, "Reading and Writing Binary Data," on p. 69.

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 are encoded in 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:

```
Reader 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:

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

See Section 2.2.4, "Character Encodings," on p. 67 for more information on character encodings.

2.2.1 How to Write Text Output

For text output, use a `PrintWriter`. That class has methods to print strings and numbers in text format. There is a convenience constructor for printing to a file. The statement

```
PrintWriter out = new PrintWriter("employee.txt", "UTF-8");
```

is equivalent to

```
PrintWriter out = new PrintWriter(  
    new FileOutputStream("employee.txt"), "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:

```
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:

```
PrintWriter out = new PrintWriter(
    new OutputStreamWriter(
        new FileOutputStream("employee.txt"), "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.

2.2.2 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:

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

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

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

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

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

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:

```
InputStream inputStream = ...;
try (BufferedReader in = new BufferedReader(new InputStreamReader(inputStream,
                                                StandardCharsets.UTF_8)))
{
    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.

2.2.3 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 `\n`. This work is done in the following `writeData` method that we add to our `Employee` class:

```
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.

```
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

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

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

```
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:

```
int n = in.nextInt();
in.nextLine(); // consume newline
Employee[] 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`

```
1 package textFile;
2
3 import java.io.*;
4 import java.time.*;
5 import java.util.*;
6
7 /**
8  * @version 1.14 2016-07-11
9  * @author Cay Horstmann
10 */
11 public class TextFileTest
12 {
13     public static void main(String[] args) throws IOException
14     {
15         Employee[] staff = new Employee[3];
16
17         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
18         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
19         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
20
21         // save all employee records to the file employee.dat
22         try (PrintWriter out = new PrintWriter("employee.dat", "UTF-8"))
23         {
24             writeData(staff, out);
25         }
26
27         // retrieve all records into a new array
28         try (Scanner in = new Scanner(
29             new FileInputStream("employee.dat"), "UTF-8"))
30         {
31             Employee[] newStaff = readData(in);
32
33             // print the newly read employee records
34             for (Employee e : newStaff)
35                 System.out.println(e);
36         }
37     }
38
39 /**
40 * Writes all employees in an array to a print writer
41 * @param employees an array of employees
```

(Continues)

Listing 2.1 (*Continued*)

```
42     * @param out a print writer
43     */
44     private static void writeData(Employee[] employees, PrintWriter out) throws IOException
45     {
46         // write number of employees
47         out.println(employees.length);
48
49         for (Employee e : employees)
50             writeEmployee(out, e);
51     }
52
53     /**
54      * Reads an array of employees from a scanner
55      * @param in the scanner
56      * @return the array of employees
57      */
58     private static Employee[] readData(Scanner in)
59     {
60         // retrieve the array size
61         int n = in.nextInt();
62         in.nextLine(); // consume newline
63
64         Employee[] employees = new Employee[n];
65         for (int i = 0; i < n; i++)
66         {
67             employees[i] = readEmployee(in);
68         }
69         return employees;
70     }
71
72     /**
73      * Writes employee data to a print writer
74      * @param out the print writer
75      */
76     public static void writeEmployee(PrintWriter out, Employee e)
77     {
78         out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
79     }
80
81     /**
82      * Reads employee data from a buffered reader
83      * @param in the scanner
84      */
85     public static Employee readEmployee(Scanner in)
86     {
```

```

87     String line = in.nextLine();
88     String[] tokens = line.split("\\|");
89     String name = tokens[0];
90     double salary = Double.parseDouble(tokens[1]);
91     LocalDate hireDate = LocalDate.parse(tokens[2]);
92     int year = hireDate.getYear();
93     int month = hireDate.getMonthValue();
94     int day = hireDate.getDayOfMonth();
95     return new Employee(name, salary, year, month, day);
96   }
97 }
```

2.2.4 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.

Table 2.1 UTF-8 Encoding

Character Range	Encoding
0...7F	0a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
80...7FF	110a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
800...FFFF	1110a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
10000...10FFFF	11110a ₂₀ a ₁₉ a ₁₈ 10a ₁₇ a ₁₆ a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀

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 big-endian format, the more significant byte comes first: 0x21 followed by 0x22. In little-endian format, it is the other way around: 0x22 0x21. To indicate which of the two is used, a file can start with the “byte order mark,” the 16-bit quantity 0xFEFF. A reader can use this value to determine the byte order and discard it.

Table 2.2 UTF-16 Encoding

Character Range	Encoding
0...FFFF	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8\ a_7a_6a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$110110b_{19}b_{18}\ b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}\ 110111a_9a_8\ a_7a_6a_5a_4a_3a_2a_1a_0$ where $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$



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

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

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

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



TIP: Some methods allow you to specify a character encoding with a `Charset` object or a string. Choose the `StandardCharsets` constants, so you don't have to worry about the correct spelling. For example, `new String(bytes, "UTF 8")` is not acceptable and will cause a runtime error.



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.

2.3 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.

2.3.1 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  
writeByte  
writeInt  
writeShort  
writeLong  
writeFloat
```

```
writeDouble  
writeChar  
writeBoolean  
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 the space needed will be the same 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`, say the decimal number 1234, or 4D2 in hexadecimal ($1234 = 4 \times 256 + 13 \times 16 + 2$). 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 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  
readShort  
readLong  
readFloat
```

```
readDouble  
readChar  
readBoolean  
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`:

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

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

```
DataOutputStream 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.

Parameters: `b` The buffer into which the data are read

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

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

Parameters: `b` The buffer into which the data are read

`off` The start offset of the data

`len` The maximum number of bytes to read

- `String readUTF()`

reads a string of characters in the “modified UTF-8” format.

- `int skipBytes(int n)`

skips `n` bytes, blocking until all bytes are skipped.

Parameters: `n` The number of bytes to be skipped

java.io.DataOutput 1.0

- void writeBoolean(boolean b)
 - void writeByte(int b)
 - void writeChar(int c)
 - void writeDouble(double d)
 - void writeFloat(float f)
 - void writeInt(int i)
 - void writeLong(long l)
 - void writeShort(int s)
- writes a value of the given type.
- void writeChars(String s)
- writes all characters in the string.
- void writeUTF(String s)
- writes a string of characters in the “modified UTF-8” format.

2.3.2 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.

```
RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
RandomAccessFile 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.

```
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
Employee 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:

```
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.

```
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.

```
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.

```
public static String readFixedString(int size, DataInput in)
    throws IOException
{
    StringBuilder b = new StringBuilder(size);
    int i = 0;
    boolean more = true;
```

```
    while (more && i < size)
    {
        char ch = in.readChar();
        i++;
        if (ch == 0) more = false;
        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.

```
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.

```
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 contains 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 at the last record first.

Listing 2.2 randomAccess/RandomAccessTest.java

```
1 package randomAccess;
2
3 import java.io.*;
4 import java.util.*;
5 import java.time.*;
```

```
6
7 /**
8  * @version 1.13 2016-07-11
9  * @author Cay Horstmann
10 */
11 public class RandomAccessTest
12 {
13     public static void main(String[] args) throws IOException
14     {
15         Employee[] staff = new Employee[3];
16
17         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
18         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
19         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
20
21         try (DataOutputStream out = new DataOutputStream(new FileOutputStream("employee.dat")))
22         {
23             // save all employee records to the file employee.dat
24             for (Employee e : staff)
25                 writeData(out, e);
26         }
27
28         try (RandomAccessFile in = new RandomAccessFile("employee.dat", "r"))
29         {
30             // retrieve all records into a new array
31
32             // compute the array size
33             int n = (int)(in.length() / Employee.RECORD_SIZE);
34             Employee[] newStaff = new Employee[n];
35
36             // read employees in reverse order
37             for (int i = n - 1; i >= 0; i--)
38             {
39                 newStaff[i] = new Employee();
40                 in.seek(i * Employee.RECORD_SIZE);
41                 newStaff[i] = readData(in);
42             }
43
44             // print the newly read employee records
45             for (Employee e : newStaff)
46                 System.out.println(e);
47         }
48     }
49
50 /**
51  * Writes employee data to a data output
52  * @param out the data output
53  * @param e the employee
54 */
```

(Continues)

Listing 2.2 (Continued)

```

55  public static void writeData(DataOutput out, Employee e) throws IOException
56  {
57      DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
58      out.writeDouble(e.getSalary());
59
60      LocalDate hireDay = e.getHireDay();
61      out.writeInt(hireDay.getYear());
62      out.writeInt(hireDay.getMonthValue());
63      out.writeInt(hireDay.getDayOfMonth());
64  }
65
66 /**
67 * Reads employee data from a data input
68 * @param in the data input
69 * @return the employee
70 */
71 public static Employee readData(DataInput in) throws IOException
72 {
73     String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
74     double salary = in.readDouble();
75     int y = in.readInt();
76     int m = in.readInt();
77     int d = in.readInt();
78     return new Employee(name, salary, y, m - 1, d);
79 }
80 }
```

java.io.RandomAccessFile 1.0

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

Parameters: file The file to be opened
 mode "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.
- void seek(long pos)
 sets the file pointer to pos bytes from the beginning of the file.
- long length()
 returns the length of the file in bytes.

2.3.3 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. Pass the entry to the `getInputStream` method of the `ZipInputStream` to obtain an input stream for reading the entry. Then call `closeEntry` to read the next entry. Here is a typical code sequence to read through a ZIP file:

```
ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
    InputStream in = zin.getInputStream(entry);
    read the contents of in
    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 you are done, call `closeEntry`. Repeat for all the files you want to store. Here is a code skeleton:

```
FileOutputStream fout = new FileOutputStream("test.zip");
ZipOutputStream zout = new ZipOutputStream(fout);
for all files
{
    ZipEntry 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 13) 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. In fact, whenever the class loader of an applet reads a JAR file, it reads and decompresses data from the network.

NOTE: Section 2.5.8, “ZIP File Systems,” on p. 115 shows how to access a ZIP archive without a special API, using the `FileSystem` class of Java SE 7.

`java.util.zip.ZipInputStream 1.1`

- `ZipInputStream(InputStream in)`
creates a `ZipInputStream` that allows you to inflate data from the given `InputStream`.
- `ZipEntry getNextEntry()`
returns a `ZipEntry` object for the next entry, or `null` if there are no more entries.
- `void closeEntry()`
closes the current open entry in the ZIP file. You can then read the next entry by using `getNextEntry()`.

`java.util.zip.ZipOutputStream 1.1`

- `ZipOutputStream(OutputStream out)`
creates a `ZipOutputStream` that you can use to write compressed data to the specified `OutputStream`.
- `void putNextEntry(ZipEntry ze)`
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.
- `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. The default value is `Deflater.DEFAULT_COMPRESSION`. Throws an `IllegalArgumentException` if the level is not valid.

Parameters: `level` A compression level, from 0 (`NO_COMPRESSION`) to 9 (`BEST_COMPRESSION`)

(Continues)

java.util.zip.ZipOutputStream 1.1 (Continued)

- `void setMethod(int method)`

sets the default compression method for this `ZipOutputStream` for any entries that do not specify a method.

Parameters: `method` The compression method, either `DEFLATED` or `STORED`

java.util.zip.ZipEntry 1.1

- `ZipEntry(String name)`

constructs a zip entry with a given name.

Parameters: `name` The name of the entry

- `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)`

Parameters: `method` The compression method for the entry; must be either `DEFLATED` or `STORED`

- `void setSize(long size)`

sets the size of this entry. Only required if the compression method is `STORED`.

Parameters: `size` The uncompressed size of this entry

- `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`.

Parameters: `crc` The checksum of this entry

java.util.zip.ZipFile 1.1

- `ZipFile(String name)`
- `ZipFile(File file)`

creates a `ZipFile` for reading from the given string or `File` object.
- `Enumeration entries()`

returns an `Enumeration` object that enumerates the `ZipEntry` objects that describe the entries of the `ZipFile`.
- `ZipEntry getEntry(String name)`

returns the entry corresponding to the given name, or `null` if there is no such entry.
Parameters: name The entry name
- `InputStream getInputStream(ZipEntry ze)`

returns an `InputStream` for the given entry.
Parameters: ze A `ZipEntry` in the ZIP file
- `String getName()`

returns the path of this ZIP file.

2.4 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.)

2.4.1 Saving and Loading Serializable Objects

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

```
ObjectOutputStream 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:

```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
Manager 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:

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

Then, retrieve the objects in the same order in which they were written, using the `readObject` method:

```
Employee e1 = (Employee) in.readObject();
Employee 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 that we need to consider: What happens when one object is shared by several objects as part of its 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 the `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:

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

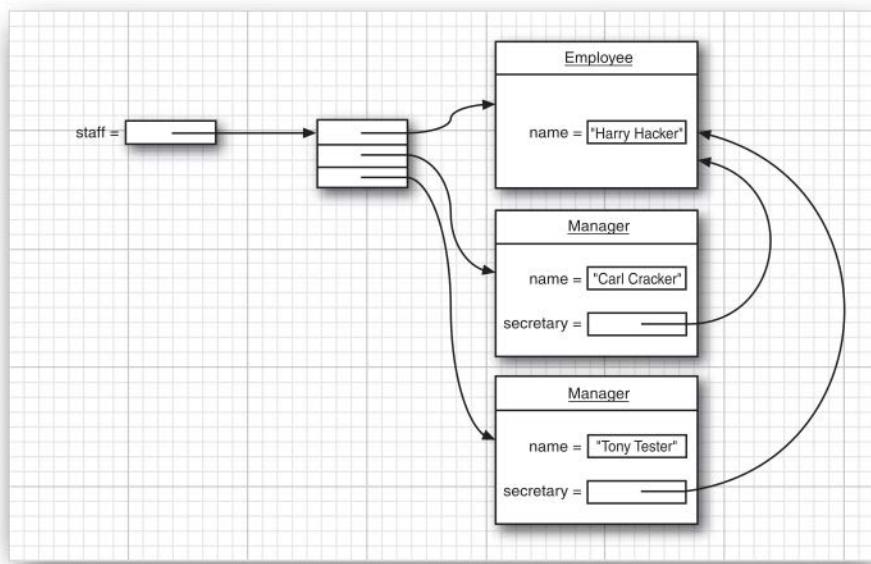


Figure 2.5 Two managers can share a mutual employee.

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).

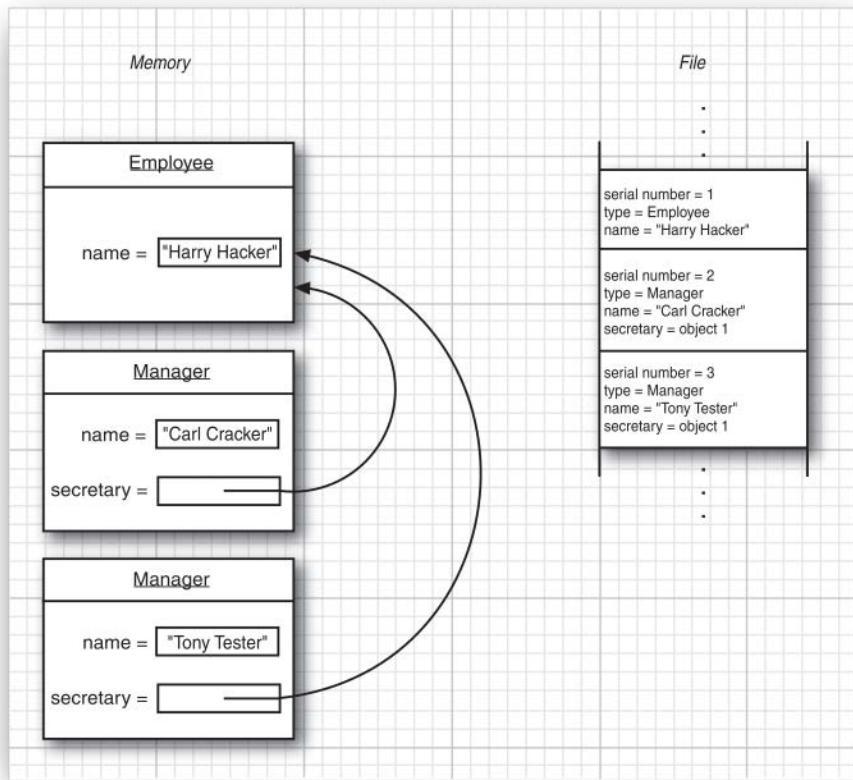


Figure 2.6 An example of object serialization

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 back the objects, 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 communicating 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.

Listing 2.3 `objectStream/ObjectStreamTest.java`

```
1 package objectStream;
2
3 import java.io.*;
4
5 /**
6 * @version 1.10 17 Aug 1998
7 * @author Cay Horstmann
8 */
9 class ObjectStreamTest
10 {
11     public static void main(String[] args) throws IOException, ClassNotFoundException
12     {
13         Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
14         Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
15         carl.setSecretary(harry);
16         Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
17         tony.setSecretary(harry);
18
19         Employee[] staff = new Employee[3];
20
21         staff[0] = carl;
22         staff[1] = harry;
23         staff[2] = tony;
24
25         // save all employee records to the file employee.dat
26         try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("employee.dat")))
27         {
28             out.writeObject(staff);
29         }
30     }
```

```
31     try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("employee.dat")))
32     {
33         // retrieve all records into a new array
34
35         Employee[] newStaff = (Employee[]) in.readObject();
36
37         // raise secretary's salary
38         newStaff[1].raiseSalary(10);
39
40         // print the newly read employee records
41         for (Employee e : newStaff)
42             System.out.println(e);
43     }
44 }
45 }
```

java.io.ObjectOutputStream 1.1

- `ObjectOutputStream(OutputStream out)`
creates an `ObjectOutputStream` so that 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 1.1

- `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 to allow multiple object references to be recovered.

2.4.2 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 length	characters
----	--------------------	------------

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” to 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 2.4.5, "Versioning," on p. 95.

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`:

```
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:

B	byte
C	char

D	double
F	float
I	int
J	long
L	object
S	short
Z	boolean
[array

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	Fingerprint and flags
E6 D2 86 7D AE AC 18 1B 02	Number of instance fields
00 03	Instance field type and name
D 00 06 salary	Instance field type and name
L 00 07 hireDay	Instance field class name: Date
74 00 10 Ljava/util/Date;	Instance field type and name
L 00 04 name	Instance field class name: String
74 00 12 Ljava/lang/String;	End marker
78	No superclass
70	

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:

40 E8 6A 00 00 00 00 00	salary field value: double
73	hireDay field value: new object
71 00 7E 00 08	Existing class java.util.Date
77 08 00 00 91 1B 4E B1 80 78	External storage (details later)
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:

75	class descriptor	4-byte number of entries
		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:

75	Array
72 00 0B [LEmployee;	New class, string length, class name <code>Employee[]</code>
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`

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

70

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	Array staff (serial #1)
72 00 0B [LEmployee;	New class, string length, class name
FC BF 36 11 C5 91 11 C7 02	Employee[] (serial #0)
00 00	Fingerprint and flags
78	Number of instance fields
70	End marker
00 00 00 03	No superclass
73	Number of array entries
72 00 07 Manager	staff[0]— new object (serial #7)
36 06 AE 13 63 8F 59 B7 02	New class, string length, class name
00 01	(serial #2)
L 00 09 secretary	Fingerprint and flags
74 00 0A LEmployee;	Number of data fields
78	Instance field type and name
72 00 08 Employee	Instance field class name: String (serial #3)
E6 D2 86 7D AE AC 18 1B 02	End marker
00 03	Superclass: new class, string length, class name (serial #4)
D 00 06 salary	Fingerprint and flags
L 00 07 hireDay	Number of instance fields
74 00 10 Ljava/util/Date;	Instance field type and name
L 00 04 name	Instance field class name: String (serial #5)
74 00 12 Ljava/lang/String;	Instance field type and name
78	Instance field class name: String (serial #6)
70	End marker
40 F3 88 00 00 00 00 00	No superclass
73	salary field value: double
72 00 0E java.util.Date	hireDay field value: new object (serial #9)
	New class, string length, class name (serial #8)

68 6A 81 01 4B 59 74 19 03	Fingerprint and flags
00 00	No instance variables
78	End marker
70	No superclass
77 08	External storage, number of bytes
00 00 00 83 E9 39 E0 00	Date
78	End marker
74 00 0C Carl Cracker	name field value: String (serial #10)
73	secretary field value: new object (serial #11)
71 00 7E 00 04	existing class (use serial #4)
40 E8 6A 00 00 00 00 00	salary field value: double
73	hireDay field value: new object (serial #12)
71 00 7E 00 08	Existing class (use serial #8)
77 08	External storage, number of bytes
00 00 00 91 1B 4E B1 80	Date
78	End marker
74 00 0C Harry Hacker	name field value: String (serial #13)
71 00 7E 00 0B	staff[1]: existing object (use serial #11)
73	staff[2]: new object (serial #14)
71 00 7E 00 02	Existing class (use serial #2)
40 E3 88 00 00 00 00 00	salary field value: double
73	hireDay field value: new object (serial #15)
71 00 7E 00 08	Existing class (use serial #8)
77 08	External storage, number of bytes
00 00 00 94 6D 3E EC 00 00	Date
78	End marker
74 00 0B Tony Tester	name field value: String (serial #16)
71 00 7E 00 0B	secretary field value: existing object (use serial #11)

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 that 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.

2.4.3 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

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

Then, the data fields are no longer automatically serialized, and 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`.

```
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.

```
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:

```
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:

```
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:

```
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.

2.4.4 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

```
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:

```
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:

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

Now the test

```
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:

```
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.

2.4.5 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;
```

If you start the `serialver` program with the `-show` option, the program brings up a graphical dialog box (see Figure 2.7).

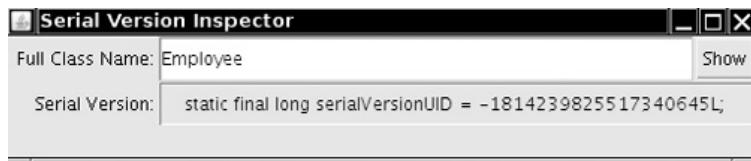


Figure 2.7 The graphical version of the `serialver` program

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

```
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 2.8 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 2.9 shows the opposite scenario: A program using 1.0 objects reads a 2.0 object. The additional `department` field is ignored.

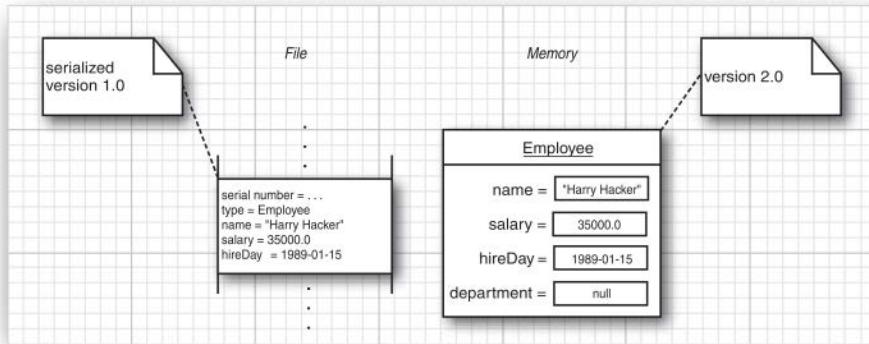


Figure 2.8 Reading an object with fewer data fields

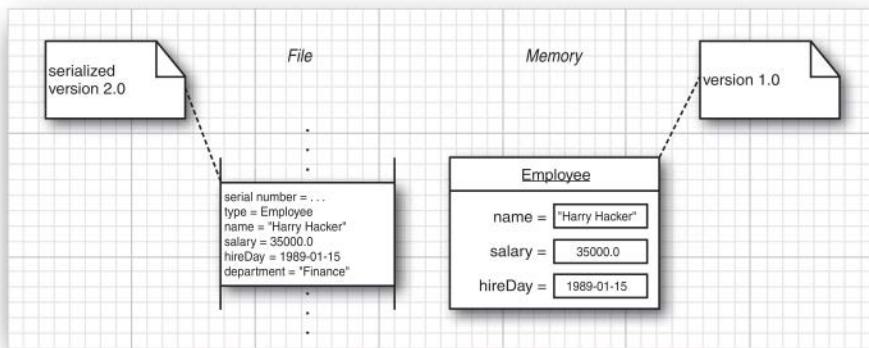


Figure 2.9 Reading an object with more data fields

Is this process safe? It depends. Dropping a data field seems harmless—the recipient still has all the data that it knew 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.

2.4.6 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 `Serializable` 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`

```
1 package serialClone;
2
3 /**
4  * @version 1.21 13 Jul 2016
5  * @author Cay Horstmann
6  */
7
8 import java.io.*;
9 import java.util.*;
10 import java.time.*;
11
12 public class SerialCloneTest
13 {
14     public static void main(String[] args) throws CloneNotSupportedException
15     {
16         Employee harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
17         // clone harry
18         Employee harry2 = (Employee) harry.clone();
19
20         // mutate harry
21         harry.raiseSalary(10);
```

```
22      // now harry and the clone are different
23      System.out.println(harry);
24      System.out.println(harry2);
25  }
26 }
27 */
28 /**
29 * A class whose clone method uses serialization.
30 */
31 class SerialCloneable implements Cloneable, Serializable
32 {
33     public Object clone() throws CloneNotSupportedException
34     {
35         try {
36             // save the object to a byte array
37             ByteArrayOutputStream bout = new ByteArrayOutputStream();
38             try (ObjectOutputStream out = new ObjectOutputStream(bout))
39             {
40                 out.writeObject(this);
41             }
42         }
43
44         // read a clone of the object from the byte array
45         try (InputStream bin = new ByteArrayInputStream(bout.toByteArray()))
46         {
47             ObjectInputStream in = new ObjectInputStream(bin);
48             return in.readObject();
49         }
50     }
51     catch (IOException | ClassNotFoundException e)
52     {
53         CloneNotSupportedException e2 = new CloneNotSupportedException();
54         e2.initCause(e);
55         throw e2;
56     }
57 }
58 */
59 /**
60 * The familiar Employee class, redefined to extend the
61 * SerialCloneable class.
62 */
63 class Employee extends SerialCloneable
64 {
65     private String name;
66     private double salary;
67     private LocalDate hireDay;
68 }
```

(Continues)

Listing 2.4 (Continued)

```
70  public Employee(String n, double s, int year, int month, int day)
71  {
72      name = n;
73      salary = s;
74      hireDay = LocalDate.of(year, month, day);
75  }
76
77  public String getName()
78  {
79      return name;
80  }
81
82  public double getSalary()
83  {
84      return salary;
85  }
86
87  public LocalDate getHireDay()
88  {
89      return hireDay;
90  }
91
92 /**
93 * Raises the salary of this employee.
94 * @byPercent the percentage of the raise
95 */
96 public void raiseSalary(double byPercent)
97 {
98     double raise = salary * byPercent / 100;
99     salary += raise;
100}
101
102 public String toString()
103 {
104     return getClass().getName()
105         + "[name=" + name
106         + ",salary=" + salary
107         + ",hireDay=" + hireDay
108         + "]";
109 }
110 }
```

2.5 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 the 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 SE 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.

2.5.1 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.

```
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:

```
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

```
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:

```
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:

```
Scanner 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 7

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

java.nio.file.Path 7

- 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.

java.io.File 1.0

- Path toPath() 7
makes a Path from this file.

2.5.2 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:

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

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

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

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

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

Conversely, if you want to write a string, call

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

To append to a given file, use

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

You can also write a collection of lines with

```
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:

```
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`.

java.nio.file.Files 7

- `static byte[] readAllBytes(Path path)`
- `static List<String> readAllLines(Path path, Charset charset)`
reads the contents of a file.
- `static Path write(Path path, byte[] contents, OpenOption... options)`
- `static Path write(Path path, Iterable<? extends CharSequence> contents, OpenOption options)`
writes the given contents to a file and returns path.

(Continues)

java.nio.file.Files 7 (Continued)

- static InputStream newInputStream(Path path, OpenOption... options)
 - static OutputStream newOutputStream(Path path, OpenOption... options)
 - static BufferedReader newBufferedReader(Path path, Charset charset)
 - static BufferedWriter newBufferedWriter(Path path, Charset charset, OpenOption... options)
- opens a file for reading or writing.

2.5.3 Creating Files and Directories

To create a new directory, call

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

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

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

You can create an empty file with

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

The call throws an exception if the file already exists. The 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.

```
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.nio.file.Files 7

- 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.

2.5.4 Copying, Moving, and Deleting Files

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

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

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

```
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:

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

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

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

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:

```
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

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

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

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

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

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

Table 2.3 Standard Options for File Operations

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

java.nio.file.Files 7

- static Path copy(Path from, Path to, CopyOption... options)
- static Path move(Path from, Path to, CopyOption... options)
copies or moves from to the given target location and returns to.
- static long copy(InputStream from, Path to, CopyOption... options)
- static long copy(Path from, OutputStream to, CopyOption... options)
copies from an input stream to a file, or from a file to an output stream, returning the number of bytes copied.
- static void delete(Path path)
- static boolean deleteIfExists(Path path)
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.

2.5.5 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.

```
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

```
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:

```
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 7

- 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.

java.nio.file.attribute.BasicFileAttributes 7

- FileTime creationTime()
- FileTime lastAccessTime()
- FileTime lastModifiedTime()
- boolean isRegularFile()
- boolean isDirectory()
- boolean isSymbolicLink()
- long size()
- Object fileKey()

gets the requested attribute.

2.5.6 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:

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

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

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

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

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

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

You can limit the depth of the tree that you want to visit by calling `Files.walk(pathToRoot, depth)`. Both `walk` methods have a varargs parameter of type

`FileVisitOption...`, but there is only one option you can supply: `FOLLOW_LINKS` to follow symbolic links.

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

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

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

Unfortunately, you cannot easily use the `Files.walk` method to delete a tree of directories since you need to first delete the children before deleting the parent. The next section shows you how to overcome that problem.

2.5.7 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

Pattern	Description	Example
*	Matches zero or more characters of a path component	*.java matches all Java files in the current directory
**	Matches zero or more characters, crossing directory boundaries	**.java matches all Java files in any subdirectory
?	Matches one character	????.java matches all four-character Java files (not counting the extension)
[. . .]	Matches a set of characters. You can use hyphens [0-9] and negation [!0-9].	Test[0-9A-F].java matches Testx.java, where x is one hexadecimal digit
{. . .}	Matches alternatives, separated by commas	*.{java,class} matches all Java and class files
\	Escapes any of the above as well as \	*** matches all files with a * in their name



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(Path 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 but the `visitFileFailed` method do nothing and continue. The `visitFileFailed` method throws the exception that caused the failure, thereby terminating the visit.

For example, here is how you can 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 that it's not allowed to access.

Also note that the attributes of the path are passed as a parameter to the `preVisitDirectory` and `visitFile` method. 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;
    }
});
```

java.nio.file.Files 7

- static `DirectoryStream<Path> newDirectoryStream(Path path)`
- static `DirectoryStream<Path> newDirectoryStream(Path path, String glob)`
gets an iterator over the files and directories in a given directory. The second method only accepts those entries matching the given glob pattern.
- static `Path walkFileTree(Path start, FileVisitor<? super Path> visitor)`
walks all descendants of the given path, applying the visitor to all descendants.

java.nio.file.SimpleFileVisitor<T> 7

- static FileVisitResult visitFile(T path, BasicFileAttributes attrs)
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.
- static FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
- static FileVisitResult postVisitDirectory(T dir, BasicFileAttributes attrs)
are called before and after visiting a directory. The default implementation does nothing and continues.
- static FileVisitResult visitFileFailed(T path, IOException exc)
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.

2.5.8 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 `zipname` is the name of a ZIP file, then the call

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

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

```
Files.copy(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:

```
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 2.3.3, "ZIP Archives," on p. 77 which required a set of new classes just to deal with ZIP archives.

java.nio.file.FileSystem 7

- static FileSystem newFileSystem(Path path, ClassLoader loader)

iterates over the installed file system providers and, provided that loader is not null, the file systems that the given class loader can load. Returns the file system created by the first file system provider that accepts the given path. By default, there is a provider for ZIP file systems that accepts files whose names end in .zip or .jar.

java.nio.file.FileSystem 7

- static Path getPath(String first, String... more)

makes a path by joining the given strings.

2.6 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.

2.6.1 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

Method	Time
Plain input stream	110 seconds
Buffered input stream	9.9 seconds
Random access file	162 seconds
Memory-mapped file	7.2 seconds

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.

```
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

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

Alternatively, you can use random access:

```
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

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

Finally, there are methods

```
getInt
getLong
getShort
getChar
getFloat
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

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

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

```
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

```
putInt
putLong
putShort
putChar
putFloat
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:

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

NOTE: For a nice explanation of the CRC algorithm, see www.relisoft.com/Science/CrcMath.html.

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 memoryMap.MemoryMapTest filename
```

Listing 2.5 *memoryMap/MemoryMapTest.java*

```
1 package memoryMap;
2
3 import java.io.*;
4 import java.nio.*;
5 import java.nio.channels.*;
6 import java.nio.file.*;
7 import java.util.zip.*;
8
9 /**
10  * This program computes the CRC checksum of a file in four ways. <br>
11  * Usage: java memoryMap.MemoryMapTest filename
12  * @version 1.01 2012-05-30
13  * @author Cay Horstmann
14  */
15 public class MemoryMapTest
16 {
17     public static long checksumInputStream(Path filename) throws IOException
18     {
19         try (InputStream in = Files.newInputStream(filename))
20         {
21             CRC32 crc = new CRC32();
22
23             int c;
24             while ((c = in.read()) != -1)
25                 crc.update(c);
26             return crc.getValue();
27         }
28     }
29
30     public static long checksumBufferedInputStream(Path filename) throws IOException
31     {
32         try (InputStream in = new BufferedInputStream(Files.newInputStream(filename)))
33         {
```

(Continues)

Listing 2.5 (Continued)

```
34     CRC32 crc = new CRC32();
35
36     int c;
37     while ((c = in.read()) != -1)
38         crc.update(c);
39     return crc.getValue();
40 }
41 }
42
43 public static long checksumRandomAccessFile(Path filename) throws IOException
44 {
45     try (RandomAccessFile file = new RandomAccessFile(filename.toFile(), "r"))
46     {
47         long length = file.length();
48         CRC32 crc = new CRC32();
49
50         for (long p = 0; p < length; p++)
51         {
52             file.seek(p);
53             int c = file.readByte();
54             crc.update(c);
55         }
56         return crc.getValue();
57     }
58 }
59
60 public static long checksumMappedFile(Path filename) throws IOException
61 {
62     try (FileChannel channel = FileChannel.open(filename))
63     {
64         CRC32 crc = new CRC32();
65         int length = (int) channel.size();
66         MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, length);
67
68         for (int p = 0; p < length; p++)
69         {
70             int c = buffer.get(p);
71             crc.update(c);
72         }
73         return crc.getValue();
74     }
75 }
76
77 public static void main(String[] args) throws IOException
78 {
79     System.out.println("Input Stream:");
80     long start = System.currentTimeMillis();
```

```
81     Path filename = Paths.get(args[0]);
82     long crcValue = checksumInputStream(filename);
83     long end = System.currentTimeMillis();
84     System.out.println(Long.toHexString(crcValue));
85     System.out.println((end - start) + " milliseconds");
86
87     System.out.println("Buffered Input Stream:");
88     start = System.currentTimeMillis();
89     crcValue = checksumBufferedInputStream(filename);
90     end = System.currentTimeMillis();
91     System.out.println(Long.toHexString(crcValue));
92     System.out.println((end - start) + " milliseconds");
93
94     System.out.println("Random Access File:");
95     start = System.currentTimeMillis();
96     crcValue = checksumRandomAccessFile(filename);
97     end = System.currentTimeMillis();
98     System.out.println(Long.toHexString(crcValue));
99     System.out.println((end - start) + " milliseconds");
100
101    System.out.println("Mapped File:");
102    start = System.currentTimeMillis();
103    crcValue = checksumMappedFile(filename);
104    end = System.currentTimeMillis();
105    System.out.println(Long.toHexString(crcValue));
106    System.out.println((end - start) + " milliseconds");
107  }
108 }
```

java.io.FileInputStream 1.0

- `FileChannel getChannel() 1.4`
returns a channel for accessing this input stream.

java.io.FileOutputStream 1.0

- `FileChannel getChannel() 1.4`
returns a channel for accessing this output stream.

java.io.RandomAccessFile 1.0

- `FileChannel getChannel() 1.4`
returns a channel for accessing this file.

java.nio.channels.FileChannel 1.4

- static FileChannel open(Path path, OpenOption... options) [7](#)

opens a file channel for the given path. By default, the channel is opened for reading.

Parameters: path The path to the file on which to open the channel

options Values WRITE, APPEND, TRUNCATE_EXISTING, CREATE in the StandardOpenOption enumeration

- MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)

maps a region of the file to memory.

Parameters: mode One of the constants READ_ONLY, READ_WRITE, or PRIVATE in the FileChannel.MapMode class

position The start of the mapped region

size The size of the mapped region

java.nio.Buffer 1.4

- boolean hasRemaining()

returns true if the current buffer position has not yet reached the buffer's limit position.

- int limit()

returns the limit position of the buffer—that is, the first position at which no more values are available.

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.

(Continues)

java.nio.ByteBuffer 1.4 (Continued)

- `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.

Parameters:

<code>destination</code>	The byte array to be filled
<code>offset</code>	The offset of the region to be filled
<code>length</code>	The length of the region to be filled

- `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.

Parameters:

<code>source</code>	The byte array to be written
<code>offset</code>	The offset of the region to be written
<code>length</code>	The length of the region to be written

- `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.
- `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.

2.6.2 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 2.10, 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

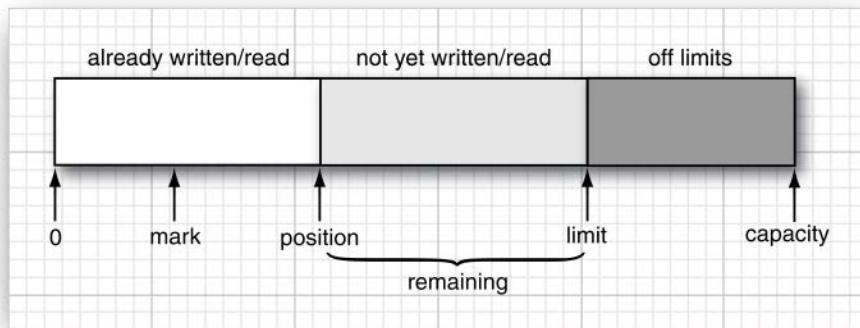


Figure 2.10 A buffer

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 $\text{limit} - \text{position}$) 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.

java.nio.Buffer 1.4

- **Buffer clear()**
prepares this buffer for writing by setting the position to 0 and the limit to the capacity; returns this.
- **Buffer flip()**
prepares this buffer for reading after writing, by setting the limit to the position and the position to 0; returns this.
- **Buffer rewind()**
prepares this buffer for rereading the same values by setting the position to 0 and leaving the limit unchanged; returns this.
- **Buffer mark()**
sets the mark of this buffer to the position; returns this.
- **Buffer reset()**
sets the position of this buffer to the mark, thus allowing the marked portion to be read or written again; returns this.
- **int remaining()**
returns the remaining number of readable or writable values—that is, the difference between the limit and position.
- **int position()**
- **void position(int newValue)**
gets and sets the position of this buffer.
- **int capacity()**
returns the capacity of this buffer.

2.6.3 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.

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

or

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

The first call blocks until the lock becomes available. The second call returns immediately, either with the lock or with `null` if the lock is not available. The file remains locked until the channel is closed or the `release` method is invoked on the lock.

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

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

or

```
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:

```
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.

Parameters: `position` The start of the region to be locked

`size` The size of the region to be locked

`shared` `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 ``.

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.

Here is 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.
- 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.

Table 2.6 Regular Expression Syntax

Expression	Description	Example
Characters		
<i>c</i> , not one of . * + ? { () [\ ^ \$	The character <i>c</i>	J
.	Any character except line terminators, or any character if the DOTALL flag is set	
\x{ <i>p</i> }	The Unicode code point with hex code <i>p</i>	\x{1D546}
\uhhhh, \xhh, \o <i>o</i> , \oo <i>o</i> , \ooo <i>o</i>	The UTF-16 code unit with the given hex or octal value	\uFEFF
\a, \e, \f, \n, \r, \t	Alert (\x{7}), escape (\x{1B}), form feed (\x{B}), newline (\x{A}), carriage return (\x{D}), tab (\x{9})	\n
\cc, where <i>c</i> is in [A-Z] or one of @ [\] ^ _ ?	The control character corresponding to the character <i>c</i>	\cH is a backspace (\x{8})
\c, where <i>c</i> is not in [A-Za-z0-9]	The character <i>c</i>	\`
\Q ... \E	Everything between the start and the end of the quotation	\Q(...)\\E matches the string (...)
Character Classes		
[C ₁ C ₂ ...], where C _i are characters, ranges <i>c-d</i> , or character classes	Any of the characters represented by C ₁ , C ₂ , ...	[0-9+-]
[^ ...]	Complement of a character class	[^\d\s]
[... && ...]	Intersection of character classes	[\p{L}&&[^A-Za-z]]
\p{ ... }, \P{ ... }	A predefined character class (see Table 2.7); its complement	\p{L} matches a Unicode letter, and so does \p{L—you can omit braces around a single letter}

(Continues)

Table 2.6 (Continued)

Expression	Description	Example
\d, \D	Digits ([0-9], or \p{Digit} when the UNICODE_CHARACTER_CLASS flag is set); the complement	\d+ is a sequence of digits
\w, \W	Word characters ([a-zA-Z0-9_], or Unicode word characters when the UNICODE_CHARACTER_CLASS flag is set); the complement	
\s, \S	Spaces ([\n\r\t\f\x{B}], or \p{IsWhite_Space} when the UNICODE_CHARACTER_CLASS flag is set); the complement	\s*, \s* is a comma surrounded by optional white space
\h, \v, \H, \V	Horizontal whitespace, vertical whitespace, their complements	
Sequences and Alternatives		
XY	Any string from X, followed by any string from Y	[1-9] [0-9]* is a positive number without leading zero
X Y	Any string from X or Y	http ftp
Grouping		
(X)	Captures the match of X	'([^\']*')' captures the quoted text
\n	The <i>n</i> th group	([""])*\1 matches 'Fred' or "Fred" but not "Fred"
(?<name>X)	Captures the match of X with the given name	'(?<id>[A-Za-z0-9]+)' captures the match with name id
\k<name>	The group with the given name	\k<id> matches the group with name id
(?:X)	Use parentheses without capturing X	In (?>http ftp)::(.*) , the match after :: is \1

(Continues)

Table 2.6 (Continued)

Expression	Description	Example
(? <i>f</i> ₁ <i>f</i> ₂ . . . :X), (? <i>f</i> ₁ . . . - <i>f</i> _k . . . :X), with <i>f</i> _i in [dimsuUx]	Matches, but does not capture, X with the given flags on or off (after -)	(?i:jpe?g) is a case-insensitive match
Other (? . . .)	See the Pattern API documentation	
Quantifiers		
X?	Optional X	\+? is an optional + sign
X*, X+	0 or more X, 1 or more X	[1-9][0-9]+ is an integer ≥ 10
X{n}, X{n,}, X{m,n}	<i>n</i> times X, at least <i>n</i> times X, between <i>m</i> and <i>n</i> times X	[0-7]{1,3} are one to three octal digits
Q?, where Q is a quantified expression	Reluctant quantifier, attempting the shortest match before trying longer matches	.*(<.+?>).* captures the shortest sequence enclosed in angle brackets
Q+, where Q is a quantified expression	Possessive quantifier, taking the longest match without backtracking	'[^\']*'+' matches strings enclosed in single quotes and fails quickly on strings without a closing quote
Boundary Matches		
^, \$	Beginning, end of input (or beginning, end of line in multiline mode)	^Java\$ matches the input or line Java
\A, \Z, \z	Beginning of input, end of input, absolute end of input (unchanged in multiline mode)	
\b, \B	Word boundary, nonword boundary	\bJava\b matches the word Java
\R	A Unicode line break	
\G	The end of the previous match	

Table 2.7 Predefined Character Class Names Used with \p

Character Class Name	Explanation
<i>posixClass</i>	<i>posixClass</i> is one of Lower, Upper, Alpha, Digit, Alnum, Punct, Graph, Print, Cntrl, XDigit, Space, Blank, ASCII, interpreted as POSIX or Unicode class, depending on the <code>UNICODE_CHARACTER_CLASS</code> flag
<code>IsScript</code> , <code>sc=Script</code> , <code>script=Script</code>	A script accepted by <code>Character.UnicodeScript.forName</code>
<code>InBlock</code> , <code>b1k=Block</code> , <code>block=Block</code>	A block accepted by <code>Character.UnicodeBlock.forName</code>
<code>Category</code> , <code>InCategory</code> , <code>gc=Category</code> , <code>general_category=Category</code>	A one- or two-letter name for a Unicode general category
<code>IsProperty</code>	<i>Property</i> is one of Alphabetic, Ideographic, Letter, Lowercase, Uppercase, Titlecase, Punctuation, Control, White_Space, Digit, Hex_Digit, Join_Control, Noncharacter_Code_Point, Assigned
<code>javaMethod</code>	Invokes the method <code>Character.isMethod</code> (must not be deprecated)

- 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).

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:

```
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

```
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

```
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

```
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]))[ap]m
```

and the input

```
11:59am
```

the matcher reports the following groups

Group Index	Start	End	String
0	0	7	11:59am
1	0	5	11:59
2	0	2	11
3	3	5	59

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

```
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 1.02 2012-06-02
11 * @author Cay Horstmann
12 */
13 public class RegexTest
14 {
15     public static void main(String[] args) throws PatternSyntaxException
16     {
17         Scanner in = new Scanner(System.in);
18         System.out.println("Enter pattern: ");
19         String patternString = in.nextLine();
20
21         Pattern pattern = Pattern.compile(patternString);
22
23         while (true)
24         {
25             System.out.println("Enter string to match: ");
26             String input = in.nextLine();
27             if (input == null || input.equals("")) return;
28             Matcher matcher = pattern.matcher(input);
29             if (matcher.matches())
30             {
31                 System.out.println("Match");
32                 int g = matcher.groupCount();
33                 if (g > 0)
34                 {
35                     for (int i = 0; i < input.length(); i++)
36                     {
37                         // Print any empty groups
38                         for (int j = 1; j <= g; j++)
39                             if (i == matcher.start(j) && i == matcher.end(j))
40                                 System.out.print("(");
41
42                         System.out.print(input.substring(i));
43
44                         if (j < g)
45                             System.out.print(",");
46
47                     }
48                     System.out.println(")");
49                 }
50             }
51         }
52     }
53 }
```

```
41         // Print ( for non-empty groups starting here
42         for (int j = 1; j <= g; j++)
43             if (i == matcher.start(j) && i != matcher.end(j))
44                 System.out.print('(');
45             System.out.print(input.charAt(i));
46         // Print ) for non-empty groups ending here
47         for (int j = 1; j <= g; j++)
48             if (i + 1 != matcher.start(j) && i + 1 == matcher.end(j))
49                 System.out.print(')');
50     }
51     System.out.println();
52 }
53 }
54 else
55     System.out.println("No match");
56 }
57 }
58 }
```

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();
    ...
}
```

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`

```
1 package match;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.regex.*;
7
```

(Continues)

Listing 2.7 (*Continued*)

```
8  /**
9  * This program displays all URLs in a web page by matching a regular expression that describes
10 * the <a href=...> HTML tag. Start the program as <br>
11 * java match.HrefMatch URL
12 * @version 1.02 2016-07-14
13 * @author Cay Horstmann
14 */
15 public class HrefMatch
16 {
17     public static void main(String[] args)
18     {
19         try
20         {
21             // get URL string from command line or use default
22             String urlString;
23             if (args.length > 0) urlString = args[0];
24             else urlString = "http://java.sun.com";
25
26             // open reader for URL
27             InputStreamReader in = new InputStreamReader(new URL(urlString).openStream(),
28                     StandardCharsets.UTF_8);
29
30             // read contents into string builder
31             StringBuilder input = new StringBuilder();
32             int ch;
33             while ((ch = in.read()) != -1)
34                 input.append((char) ch);
35
36             // search for all occurrences of pattern
37             String patternString = "<a\\s+href\\s*=\\s*(\"[^"]*\"|[^\s>]*\")\\s*>";
38             Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
39             Matcher matcher = pattern.matcher(input);
40
41             while (matcher.find())
42             {
43                 String match = matcher.group();
44                 System.out.println(match);
45             }
46         }
47         catch (IOException | PatternSyntaxException e)
48         {
49             e.printStackTrace();
50         }
51     }
52 }
```

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:

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

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))`.

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

Finally, the `Pattern` class has a `split` method that splits an input into an array of strings, using the regular expression matches as boundaries. For example, the following instructions split the input into tokens, where the delimiters are punctuation marks surrounded by optional whitespace.

```
Pattern pattern = Pattern.compile("\\s*\\p{Punct}\\s*");
String[] tokens = pattern.split(input);
```

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

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

If you don't care about precompiling the pattern or lazy fetching, you can just use the `String.split` method:

```
String[] tokens = input.split("\\s*\\s*");
```

java.util.regex.Pattern 1.4

- static `Pattern compile(String expression)`
- static `Pattern compile(String expression, int flags)`

compiles the regular expression string into a pattern object for fast processing of matches.

Parameters: expression The regular expression

flags One or more of the flags CASE_INSENSITIVE, UNICODE_CASE, MULTILINE, UNIX_LINES, DOTALL, and CANON_EQ

(Continues)

java.util.regex.Pattern 1.4 (Continued)

- `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) 8`
splits the input string into tokens, where the pattern specifies the form of the delimiters. Returns an array or stream of tokens. The delimiters are not part of the tokens.

Parameters: `input` The string to be split into tokens
 `limit` 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.

java.util.regex.Matcher 1.4

- `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.

Parameters: `start` The index at which to start searching

- `int start()`
- `int end()`
returns the start or past-the-end position of the current match.
- `String group()`
returns the current match.

(Continues)

java.util.regex.Matcher 1.4 (Continued)

- `int groupCount()`
returns the number of groups in the input pattern.
- `int start(int groupIndex)`
- `int end(int groupIndex)`
returns the start or past-the-end position of a given group in the current match.
Parameters: `groupIndex` The group index (starting with 1), or 0 to indicate the entire match
- `String group(int groupIndex)`
returns the string matching a given group.
Parameters: `groupIndex` The group index (starting with 1), or 0 to indicate the entire match
- `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.
Parameters: `replacement` The replacement string. It 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.
- `Matcher reset()`
- `Matcher reset(CharSequence input)`
resets the matcher state. The second method makes the matcher work on a different input. Both methods return this.

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.

This page intentionally left blank

3

CHAPTER

XML

In this chapter

- 3.1 Introducing XML, page 144
- 3.2 Parsing an XML Document, page 149
- 3.3 Validating XML Documents, page 162
- 3.4 Locating Information with XPath, page 190
- 3.5 Using Namespaces, page 196
- 3.6 Streaming Parsers, page 199
- 3.7 Generating XML Documents, page 208
- 3.8 XSL Transformations, page 222

The preface of the book *Essential XML* by Don Box et al. (Addison-Wesley, 2000) states only half-jokingly: “The Extensible Markup Language (XML) has replaced Java, Design Patterns, and Object Technology as the software industry’s solution to world hunger.” Indeed, as you will see in this chapter, XML is a very useful technology for describing structured information. XML tools make it easy to process and transform that information. However, XML is not a silver bullet. You need domain-specific standards and code libraries to use it effectively. Moreover, far from making Java obsolete, XML works very well with Java. Since the late 1990s, IBM, Apache, and others have been instrumental in producing high-quality Java libraries for XML processing. Many of these libraries have now been integrated into the Java platform.

This chapter introduces XML and covers the XML features of the Java library. As always, we'll point out along the way when the hype surrounding XML is justified—and when you have to take it with a grain of salt and try solving your problems the old-fashioned way: through good design and code.

3.1 Introducing XML

In Chapter 13 of Volume I, you have seen the use of *property files* to describe the configuration of a program. A property file contains a set of name/value pairs, such as

```
fontname=Times Roman  
fontsize=12  
windowsize=400 200  
color=0 50 100
```

You can use the `Properties` class to read in such a file with a single method call. That's a nice feature, but it doesn't really go far enough. In many cases, the information you want to describe has more structure than the property file format can comfortably handle. Consider the `fontname`/`fontsize` entries in the example. It would be more object-oriented to have a single entry:

```
font=Times Roman 12
```

But then, parsing the font description gets ugly as you have to figure out when the font name ends and the font size starts.

Property files have a single flat hierarchy. You can often see programmers work around that limitation with key names like

```
title.fontname=Helvetica  
title.fontsize=36  
body.fontname=Times Roman  
body.fontsize=12
```

Another shortcoming of the property file format is the requirement that keys must be unique. To store a sequence of values, you need another workaround, such as

```
menu.item.1=Times Roman  
menu.item.2=Helvetica  
menu.item.3=Goudy Old Style
```

The XML format solves these problems. It can express hierarchical structures and is thus more flexible than the flat table structure of a property file.

An XML file for describing a program configuration might look like this:

```
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  <body>
    <font>
      <name>Times Roman</name>
      <size>12</size>
    </font>
  </body>
  <window>
    <width>400</width>
    <height>200</height>
  </window>
  <color>
    <red>0</red>
    <green>50</green>
    <blue>100</blue>
  </color>
  <menu>
    <item>Times Roman</item>
    <item>Helvetica</item>
    <item>Goudy Old Style</item>
  </menu>
</configuration>
```

The XML format allows you to express the hierarchy and record repeated elements without contortions.

The format of an XML file is straightforward. It looks similar to an HTML file. There is a good reason for that—both the XML and HTML formats are descendants of the venerable Standard Generalized Markup Language (SGML).

SGML has been around since the 1970s for describing the structure of complex documents. It has been used with success in some industries that require ongoing maintenance of massive documentation—in particular, the aircraft industry. However, SGML is quite complex, so it has never caught on in a big way. Much of that complexity arises because SGML has two conflicting goals. SGML wants to make sure that documents are formed according to the rules for their document type, but it also wants to make data entry easy by allowing shortcuts that reduce typing. XML was designed as a simplified version of SGML for use on the Internet.

As is often true, simpler is better, and XML has enjoyed the immediate and enthusiastic reception that has eluded SGML for so long.

NOTE: You can find a very nice version of the XML standard, with annotations by Tim Bray, at www.xml.com/xml/axml.html.

Even though XML and HTML have common roots, there are important differences between the two.

- Unlike HTML, XML is case-sensitive. For example, `<H1>` and `<h1>` are different XML tags.
- In HTML, you can omit end tags, such as `</p>` or ``, if it is clear from the context where a paragraph or list item ends. In XML, you can never omit an end tag.
- In XML, elements that have a single tag without a matching end tag must end in a `/`, as in ``. That way, the parser knows not to look for a `` tag.
- In XML, attribute values must be enclosed in quotation marks. In HTML, quotation marks are optional. For example, `<applet code="MyApplet.class" width=300 height=300>` is legal HTML but not legal XML. In XML, you have to use quotation marks: `width="300"`.
- In HTML, you can have attribute names without values, such as `<input type="radio" name="language" value="Java" checked>`. In XML, all attributes must have values, such as `checked="true"` or (ugh) `checked="checked"`.

3.1.1 The Structure of an XML Document

An XML document should start with a header such as

```
<?xml version="1.0"?>
```

or

```
<?xml version="1.0" encoding="UTF-8"?>
```

Strictly speaking, a header is optional, but it is highly recommended.

NOTE: Since SGML was created for processing of real documents, XML files are called *documents* even though many of them describe data sets that one would not normally call documents.

The header can be followed by a *document type definition* (DTD), such as

```
<!DOCTYPE web-app PUBLIC  
  "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"  
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

DTDs are an important mechanism to ensure the correctness of a document, but they are not required. We will discuss them later in this chapter.

Finally, the body of the XML document contains the *root element*, which can contain other elements. For example,

```
<?xml version="1.0"?>  
<!DOCTYPE configuration . . .>  
<configuration>  
  <title>  
    <font>  
      <name>Helvetica</name>  
      <size>36</size>  
    </font>  
  </title>  
  . . .  
</configuration>
```

An element can contain *child elements*, text, or both. In the preceding example, the `font` element has two child elements, `name` and `size`. The `name` element contains the text "Helvetica".



TIP: It is best to structure your XML documents so that an element contains either child elements or text. In other words, you should avoid situations such as

```
<font>  
  Helvetica  
  <size>36</size>  
</font>
```

This is called *mixed content* in the XML specification. As you will see later in this chapter, you can simplify parsing if you avoid mixed content.

XML elements can contain attributes, such as

```
<size unit="pt">36</size>
```

There is some disagreement among XML designers about when to use elements and when to use attributes. For example, it would seem easier to describe a font as

```
<font name="Helvetica" size="36"/>
```

than

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

However, attributes are much less flexible. Suppose you want to add units to the size value. If you use attributes, you will have to add the unit to the attribute value:

```
<font name="Helvetica" size="36 pt"/>
```

Ugh! Now you have to parse the string "36 pt", just the kind of hassle that XML was designed to avoid. Adding an attribute to the size element is much cleaner:

```
<font>
  <name>Helvetica</name>
  <size unit="pt">36</size>
</font>
```

A commonly used rule of thumb is that attributes should be used only to modify the interpretation of a value, not to specify values. If you find yourself engaged in metaphysical discussions about whether a particular setting is a modification of the interpretation of a value or not, just say "no" to attributes and use elements throughout. Many useful XML documents don't use attributes at all.

NOTE: In HTML, the rule for attribute usage is simple: If it isn't displayed on the web page, it's an attribute. For example, consider the hyperlink

```
<a href="http://java.sun.com">Java Technology</a>
```

The string Java Technology is displayed on the web page, but the URL of the link is not a part of the displayed page. However, the rule isn't all that helpful for most XML files because the data in an XML file aren't normally meant to be viewed by humans.

Elements and text are the "bread and butter" of XML documents. Here are a few other markup instructions that you might encounter:

- *Character references* have the form `&#decimalValue;` or `&#xhexValue;`. For example, the é character can be denoted with either of the following:

`é é`

- *Entity references* have the form `&name;`. The entity references

`< > & " '`

have predefined meanings: the less-than, greater-than, ampersand, quotation mark, and apostrophe characters. You can define other entity references in a DTD.

- *CDATA sections* are delimited by `<![CDATA[` and `]]>`. They are a special form of character data. You can use them to include strings that contain characters such as `<`, `>`, `&` without having them interpreted as markup, for example:

```
<![CDATA[< & > are my favorite delimiters]]>
```

CDATA sections cannot contain the string `]]>`. Use this feature with caution! It is too often used as a back door for smuggling legacy data into XML documents.

- *Processing instructions* are instructions for applications that process XML documents. They are delimited by `<?` and `?>`, for example

```
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

Every XML document starts with a processing instruction

```
<?xml version="1.0"?>
```

- *Comments* are delimited by `<!--` and `-->`, for example

```
<!-- This is a comment. -->
```

Comments should not contain the string `--`. Comments should only be information for human readers. They should never contain hidden commands; use processing instructions for commands.

3.2 Parsing an XML Document

To process an XML document, you need to *parse* it. A parser is a program that reads a file, confirms that the file has the correct format, breaks it up into the constituent elements, and lets a programmer access those elements. The Java library supplies two kinds of XML parsers:

- Tree parsers, such as the Document Object Model (DOM) parser, that read an XML document into a tree structure.
- Streaming parsers, such as the Simple API for XML (SAX) parser, that generate events as they read an XML document.

The DOM parser is easier to use for most purposes, and we explain it first. You may consider a streaming parser if you process very long documents whose tree structures would use up a lot of memory, or if you are only interested in a few

elements and don't care about their context. For more information, see Section 3.6, "Streaming Parsers," on p. 199.

The DOM parser interface is standardized by the World Wide Web Consortium (W3C). The `org.w3c.dom` package contains the definitions of interface types such as `Document` and `Element`. Different suppliers, such as the Apache Organization and IBM, have written DOM parsers whose classes implement these interfaces. The Java API for XML Processing (JAXP) library actually makes it possible to plug in any of these parsers. But the JDK also comes with a DOM parser that is derived from the Apache parser.

To read an XML document, you need a `DocumentBuilder` object that you get from a `DocumentBuilderFactory` like this:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

You can now read a document from a file:

```
File f = . . .
Document doc = builder.parse(f);
```

Alternatively, you can use a URL:

```
URL u = . . .
Document doc = builder.parse(u);
```

You can even specify an arbitrary input stream:

```
InputStream in = . . .
Document doc = builder.parse(in);
```

NOTE: If you use an input stream as an input source, the parser will not be able to locate other files that are referenced relative to the location of the document, such as a DTD in the same directory. You can install an "entity resolver" to overcome that problem. See www.xml.com/pub/a/2004/03/03/catalogs.html or www.ibm.com/developerworks/xml/library/x-mxd3.html for more information.

The `Document` object is an in-memory representation of the tree structure of the XML document. It is composed of objects whose classes implement the `Node` interface and its various subinterfaces. Figure 3.1 shows the inheritance hierarchy of the subinterfaces.

Start analyzing the contents of a document by calling the `getDocumentElement` method. It returns the root element.

```
Element root = doc.getDocumentElement();
```

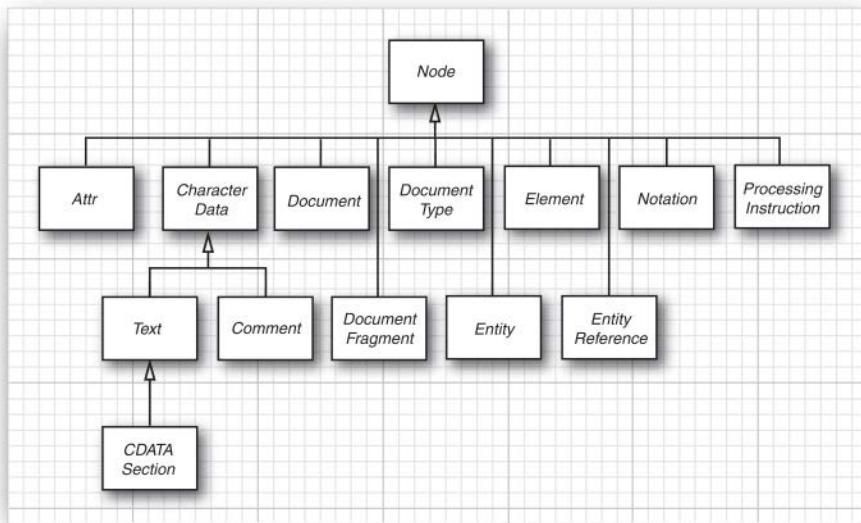


Figure 3.1 The `Node` interface and its subinterfaces

For example, if you are processing a document

```
<?xml version="1.0"?>
<font>
  ...
</font>
```

then calling `getDocumentElement` returns the `font` element.

The `getTagName` method returns the tag name of an element. In the preceding example, `root.getTagName()` returns the string "font".

To get the element's children (which may be subelements, text, comments, or other nodes), use the `getChildNodes` method. That method returns a collection of type `NodeList`. That type was standardized before the standard Java collections, so it has a different access protocol. The `item` method gets the item with a given index, and the `getLength` method gives the total count of the items. Therefore, you can enumerate all children like this:

```
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    ...
}
```

Be careful when analyzing children. Suppose, for example, that you are processing the document

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

You would expect the `font` element to have two children, but the parser reports five:

- The whitespace between `` and `<name>`
- The `name` element
- The whitespace between `</name>` and `<size>`
- The `size` element
- The whitespace between `</size>` and ``

Figure 3.2 shows the DOM tree.

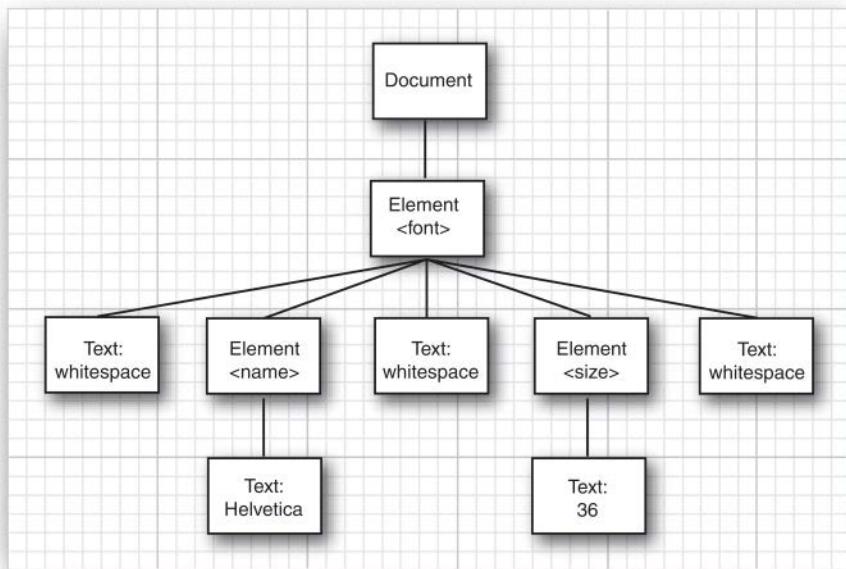


Figure 3.2 A simple DOM tree

If you expect only subelements, you can ignore the whitespace:

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element) child;
        ...
    }
}
```

Now you look at only two elements, with tag names `name` and `size`.

As you will see in the next section, you can do even better if your document has a DTD. Then the parser knows which elements don't have text nodes as children, and it can suppress the whitespace for you.

When analyzing the `name` and `size` elements, you want to retrieve the text strings that they contain. Those text strings are themselves contained in child nodes of type `Text`. You know that these `Text` nodes are the only children, so you can use the `getFirstChild` method without having to traverse another `NodeList`. Then, use the `getData` method to retrieve the string stored in the `Text` node.

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element) child;
        Text textNode = (Text) childElement.getFirstChild();
        String text = textNode.getData().trim();
        if (childElement.getTagName().equals("name"))
            name = text;
        else if (childElement.getTagName().equals("size"))
            size = Integer.parseInt(text);
    }
}
```



TIP: It is a good idea to call `trim` on the return value of the `getData` method. If the author of an XML file puts the beginning and the ending tags on separate lines, such as

```
<size>
  36
</size>
```

then the parser will include all line breaks and spaces in the text node data.

Calling the `trim` method removes the whitespace surrounding the actual data.

You can also get the last child with the `getLastChild` method, and the next sibling of a node with `getNextSibling`. Therefore, another way of traversing a node's children is

```
for (Node childNode = element.getFirstChild();  
     childNode != null;  
     childNode = childNode.getNextSibling())  
{  
    . . .  
}
```

To enumerate the attributes of a node, call the `getAttributes` method. It returns a `NamedNodeMap` object that contains `Node` objects describing the attributes. You can traverse the nodes in a `NamedNodeMap` in the same way as a `NodeList`. Then, call the `getNodeName` and `getNodeValue` methods to get the attribute names and values.

```
NamedNodeMap attributes = element.getAttributes();  
for (int i = 0; i < attributes.getLength(); i++)  
{  
    Node attribute = attributes.item(i);  
    String name = attribute.getNodeName();  
    String value = attribute.getNodeValue();  
    . . .  
}
```

Alternatively, if you know the name of an attribute, you can retrieve the corresponding value directly:

```
String unit = element.getAttribute("unit");
```

You have now seen how to analyze a DOM tree. The program in Listing 3.1 puts these techniques to work. You can use the File → Open menu option to read in an XML file. A `DocumentBuilder` object parses the XML file and produces a `Document` object. The program displays the `Document` object as a tree (see Figure 3.3).

The tree display clearly shows how child elements are surrounded by text containing whitespace and comments. For greater clarity, the program displays newline and return characters as `\n` and `\r`. (Otherwise, they would show up as hollow boxes—which is the default symbol for a character in a string that Swing cannot draw.)

In Chapter 10, you will learn the techniques this program uses to display the tree and the attribute tables. The `DOMTreeModel` class implements the `TreeModel` interface. The `getRoot` method returns the root element of the document. The `getChild` method gets the node list of children and returns the item with the requested index. The tree cell renderer displays the following:

- For elements, the element tag name and a table of all attributes

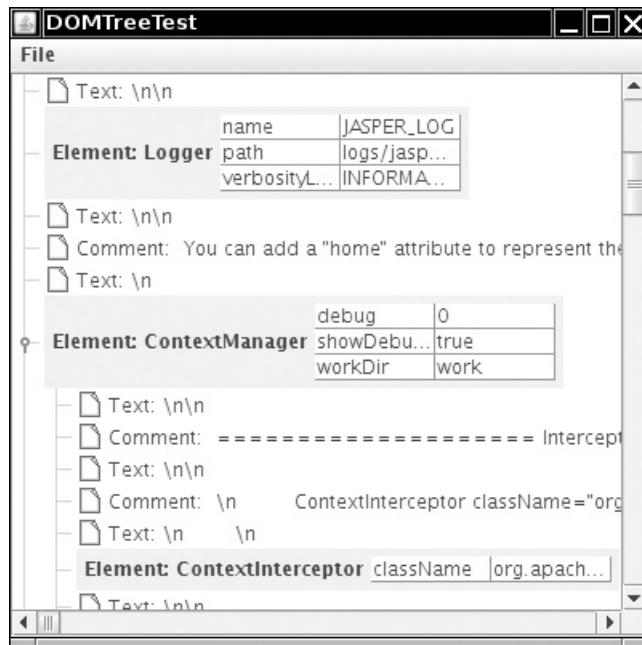


Figure 3.3 A parse tree of an XML document

- For character data, the interface (`Text`, `Comment`, or `CDATASection`), followed by the data, with newline and return characters replaced by `\n` and `\r`
- For all other node types, the class name followed by the result of `toString`

Listing 3.1 dom/TreeViewer.java

```
1 package dom;
2
3 import java.awt.*;
4 import java.io.*;
5
6 import javax.swing.*;
7 import javax.swing.event.*;
8 import javax.swing.table.*;
9 import javax.swing.tree.*;
10 import javax.xml.parsers.*;
11
12 import org.w3c.dom.*;
13 import org.w3c.dom.CharacterData;
```

(Continues)

Listing 3.1 (*Continued*)

```
15 /**
16 * This program displays an XML document as a tree.
17 * @version 1.13 2016-04-27
18 * @author Cay Horstmann
19 */
20 public class TreeViewer
21 {
22     public static void main(String[] args)
23     {
24         EventQueue.invokeLater(() ->
25         {
26             JFrame frame = new DOMTreeFrame();
27             frame.setTitle("TreeViewer");
28             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29             frame.setVisible(true);
30         });
31     }
32 }
33
34 /**
35 * This frame contains a tree that displays the contents of an XML document.
36 */
37 class DOMTreeFrame extends JFrame
38 {
39     private static final int DEFAULT_WIDTH = 400;
40     private static final int DEFAULT_HEIGHT = 400;
41
42     private DocumentBuilder builder;
43
44     public DOMTreeFrame()
45     {
46         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
47
48         JMenu fileMenu = new JMenu("File");
49         JMenuItem openItem = new JMenuItem("Open");
50         openItem.addActionListener(event -> openFile());
51         fileMenu.add(openItem);
52
53         JMenuItem exitItem = new JMenuItem("Exit");
54         exitItem.addActionListener(event -> System.exit(0));
55         fileMenu.add(exitItem);
56
57         JMenuBar menuBar = new JMenuBar();
58         menuBar.add(fileMenu);
59         setJMenuBar(menuBar);
60     }
61 }
```

```
62  /**
63  * Open a file and load the document.
64  */
65 public void openFile()
66 {
67     JFileChooser chooser = new JFileChooser();
68     chooser.setCurrentDirectory(new File("dom"));
69     chooser.setFileFilter(
70         new javax.swing.filechooser.FileNameExtensionFilter("XML files", "xml"));
71     int r = chooser.showOpenDialog(this);
72     if (r != JFileChooser.APPROVE_OPTION) return;
73     final File file = chooser.getSelectedFile();
74
75     new SwingWorker<Document, Void>()
76     {
77         protected Document doInBackground() throws Exception
78         {
79             if (builder == null)
80             {
81                 DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
82                 builder = factory.newDocumentBuilder();
83             }
84             return builder.parse(file);
85         }
86
87         protected void done()
88         {
89             try
90             {
91                 Document doc = get();
92                 JTree tree = new JTree(new DOMTreeModel(doc));
93                 tree.setCellRenderer(new DOMTreeCellRenderer());
94
95                 setContentPane(new JScrollPane(tree));
96                 validate();
97             }
98             catch (Exception e)
99             {
100                 JOptionPane.showMessageDialog(DOMTreeFrame.this, e);
101             }
102         }
103     }.execute();
104 }
105 }
106 /**
107 * This tree model describes the tree structure of an XML document.
108 */
109 */
```

(Continues)

Listing 3.1 (*Continued*)

```
110 class DOMTreeModel implements TreeModel
111 {
112     private Document doc;
113
114     /**
115      * Constructs a document tree model.
116      * @param doc the document
117      */
118     public DOMTreeModel(Document doc)
119     {
120         this.doc = doc;
121     }
122
123     public Object getRoot()
124     {
125         return doc.getDocumentElement();
126     }
127
128     public int getChildCount(Object parent)
129     {
130         Node node = (Node) parent;
131         NodeList list = node.getChildNodes();
132         return list.getLength();
133     }
134
135     public Object getChild(Object parent, int index)
136     {
137         Node node = (Node) parent;
138         NodeList list = node.getChildNodes();
139         return list.item(index);
140     }
141
142     public int getIndexOfChild(Object parent, Object child)
143     {
144         Node node = (Node) parent;
145         NodeList list = node.getChildNodes();
146         for (int i = 0; i < list.getLength(); i++)
147             if (getChild(node, i) == child) return i;
148         return -1;
149     }
150
151     public boolean isLeaf(Object node)
152     {
153         return getChildCount(node) == 0;
154     }
155
```

```
156     public void valueForPathChanged(TreePath path, Object newValue) {}
157     public void addTreeModelListener(TreeModelListener l) {}
158     public void removeTreeModelListener(TreeModelListener l) {}
159 }
160
161 /**
162 * This class renders an XML node.
163 */
164 class DOMTreeCellRenderer extends DefaultTreeCellRenderer
165 {
166     public Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected,
167           boolean expanded, boolean leaf, int row, boolean hasFocus)
168     {
169         Node node = (Node) value;
170         if (node instanceof Element) return elementPanel((Element) node);
171
172         super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row, hasFocus);
173         if (node instanceof CharacterData) setText(characterString((CharacterData) node));
174         else setText(node.getClass() + ": " + node.toString());
175         return this;
176     }
177
178     public static JPanel elementPanel(Element e)
179     {
180         JPanel panel = new JPanel();
181         panel.add(new JLabel("Element: " + e.getTagName()));
182         final NamedNodeMap map = e.getAttributes();
183         panel.add(new JTable(new AbstractTableModel()
184         {
185             public int getRowCount()
186             {
187                 return map.getLength();
188             }
189
190             public int getColumnCount()
191             {
192                 return 2;
193             }
194
195             public Object getValueAt(int r, int c)
196             {
197                 return c == 0 ? map.item(r).getNodeName() : map.item(r).getNodeValue();
198             }
199         }));
200         return panel;
201     }
202
203     private static String characterString(CharacterData node)
204     {
```

(Continues)

Listing 3.1 (Continued)

```
205     StringBuilder builder = new StringBuilder(node.getData());
206     for (int i = 0; i < builder.length(); i++)
207     {
208         if (builder.charAt(i) == '\r')
209         {
210             builder.replace(i, i + 1, "\\r");
211             i++;
212         }
213         else if (builder.charAt(i) == '\n')
214         {
215             builder.replace(i, i + 1, "\\n");
216             i++;
217         }
218         else if (builder.charAt(i) == '\t')
219         {
220             builder.replace(i, i + 1, "\\t");
221             i++;
222         }
223     }
224     if (node instanceof CDATASection) builder.insert(0, "CDATASection: ");
225     else if (node instanceof Text) builder.insert(0, "Text: ");
226     else if (node instanceof Comment) builder.insert(0, "Comment: ");
227
228     return builder.toString();
229 }
230 }
```

javax.xml.parsers.DocumentBuilderFactory 1.4

- static DocumentBuilderFactory newInstance()
returns an instance of the DocumentBuilderFactory class.
- DocumentBuilder newDocumentBuilder()
returns an instance of the DocumentBuilder class.

javax.xml.parsers.DocumentBuilder 1.4

- Document parse(File f)
- Document parse(String url)
- Document parse(InputStream in)

parses an XML document from the given file, URL, or input stream and returns the parsed document.

org.w3c.dom.Document 1.4

- `Element getDocumentElement()`
returns the root element of the document.

org.w3c.dom.Element 1.4

- `String getTagName()`
returns the name of the element.
- `String getAttribute(String name)`
returns the value of the attribute with the given name, or the empty string if there is no such attribute.

org.w3c.dom.Node 1.4

- `NodeList getChildNodes()`
returns a node list that contains all children of this node.
- `Node getFirstChild()`
- `Node getLastChild()`
gets the first or last child node of this node, or `null` if this node has no children.
- `Node getNextSibling()`
- `Node getPreviousSibling()`
gets the next or previous sibling of this node, or `null` if this node has no siblings.
- `Node getParentNode()`
gets the parent of this node, or `null` if this node is the document node.
- `NamedNodeMap getAttributes()`
returns a node map that contains `Attr` nodes that describe all attributes of this node.
- `String getNodeName()`
returns the name of this node. If the node is an `Attr` node, the name is the attribute name.
- `String getNodeValue()`
returns the value of this node. If the node is an `Attr` node, the value is the attribute value.

org.w3c.dom.CharacterData 1.4

- `String getData()`
returns the text stored in this node.

org.w3c.dom.NodeList 1.4

- `int getLength()`
returns the number of nodes in this list.
- `Node item(int index)`
returns the node with the given index. The index is between 0 and `getLength() - 1`.

org.w3c.dom.NamedNodeMap 1.4

- `int getLength()`
returns the number of nodes in this map.
- `Node item(int index)`
returns the node with the given index. The index is between 0 and `getLength() - 1`.

3.3 Validating XML Documents

In the previous section, you saw how to traverse the tree structure of a DOM document. However, if you simply follow that approach, you'll have to do quite a bit of tedious programming and error checking. Not only will you have to deal with whitespace between elements, but you will also need to check whether the document contains the nodes that you expect. For example, suppose you are reading an element:

```
<font>
    <name>Helvetica</name>
    <size>36</size>
</font>
```

You get the first child. Oops . . . it is a text node containing whitespace "\n ". You skip text nodes and find the first element node. Then, you need to check that its tag name is "name" and that it has one child node of type `Text`. You move on to the next nonwhitespace child and make the same check. What if the author of the document switched the order of the children or added another child element? It is tedious to code all this error checking—but reckless to skip the checks.

Fortunately, one of the major benefits of an XML parser is that it can automatically verify that a document has the correct structure. Then, parsing becomes much simpler. For example, if you know that the `font` fragment has passed validation, you can simply get the two grandchildren, cast them as `Text` nodes, and get the text data, without any further checking.

To specify the document structure, you can supply a DTD or an XML Schema definition. A DTD or schema contains rules that explain how a document should be formed, by specifying the legal child elements and attributes for each element. For example, a DTD might contain a rule:

```
<!ELEMENT font (name,size)>
```

This rule expresses that a `font` element must always have two children, which are `name` and `size` elements. The XML Schema language expresses the same constraint as

```
<xsd:element name="font">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
  </xsd:sequence>
</xsd:element>
```

XML Schema can express more sophisticated validation conditions (such as the fact that the `size` element must contain an integer) than can DTDs. Unlike the DTD syntax, the XML Schema syntax itself uses XML, which is a benefit if you need to process schema files.

In the next section, we will discuss DTDs in detail, then briefly cover the basics of XML Schema support. Finally, we will present a complete application that demonstrates how validation simplifies XML programming.

3.3.1 Document Type Definitions

There are several methods for supplying a DTD. You can include a DTD in an XML document like this:

```
<?xml version="1.0"?>
<!DOCTYPE configuration [
  <!ELEMENT configuration . . .>
  more rules
  .
  .
  ]>
<configuration>
  .
  .
</configuration>
```

As you can see, the rules are included inside a DOCTYPE declaration, in a block delimited by [. . .]. The document type must match the name of the root element, such as configuration in our example.

Supplying a DTD inside an XML document is somewhat uncommon because DTDs can grow lengthy. It makes more sense to store the DTD externally. The SYSTEM declaration can be used for that purpose. Specify a URL that contains the DTD, for example:

```
<!DOCTYPE configuration SYSTEM "config.dtd">
```

or

```
<!DOCTYPE configuration SYSTEM "http://myserver.com/config.dtd">
```



CAUTION: If you use a relative URL for the DTD (such as "config.dtd"), give the parser a File or URL object, not an InputStream. If you must parse from an input stream, supply an entity resolver (see the following note).

Finally, the mechanism for identifying well-known DTDs has its origin in SGML. Here is an example:

```
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

If an XML processor knows how to locate the DTD with the public identifier, it need not go to the URL.

NOTE: If you use a DOM parser and would like to support a PUBLIC identifier, call the setEntityResolver method of the DocumentBuilder class to install an object of a class that implements the EntityResolver interface. That interface has a single method, resolveEntity. Here is the outline of a typical implementation:

```
class MyEntityResolver implements EntityResolver
{
    public InputSource resolveEntity(String publicID, String systemID)
    {
        if (publicID.equals(a known ID))
            return new InputSource(DTD data);
        else
            return null; // use default behavior
    }
}
```

You can construct the input source from an InputStream, a Reader, or a string.

Now that you have seen how the parser locates the DTD, let us consider the various kinds of rules.

The ELEMENT rule specifies what children an element can have. Use a regular expression, made up of the components shown in Table 3.1.

Table 3.1 Rules for Element Content

Rule	Meaning
E^*	0 or more occurrences of E
E^+	1 or more occurrences of E
$E?$	0 or 1 occurrences of E
$E_1 E_2 \dots E_n$	One of E_1, E_2, \dots, E_n
E_1, E_2, \dots, E_n	E_1 followed by E_2, \dots, E_n
#PCDATA	Text
$(\#PCDATA E_1 E_2 \dots E_n)^*$	0 or more occurrences of text and E_1, E_2, \dots, E_n in any order (mixed content)
ANY	Any children allowed
EMPTY	No children allowed

Here are several simple but typical examples. The following rule states that a menu element contains 0 or more item elements:

```
<!ELEMENT menu (item)*>
```

This set of rules states that a font is described by a name followed by a size, each of which contain text:

```
<!ELEMENT font (name,size)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT size (#PCDATA)>
```

The abbreviation PCDATA denotes *parsed character data*. It is “parsed” because the parser interprets the text string, looking for < characters that denote the start of a new tag, or & characters that denote the start of an entity.

An element specification can contain regular expressions that are nested and complex. For example, here is a rule that describes the makeup of a chapter in a book:

```
<!ELEMENT chapter (intro,(heading,(para|image|table|note)+)+)
```

Each chapter starts with an introduction, which is followed by one or more sections consisting of a heading and one or more paragraphs, images, tables, or notes.

However, in one common case you can't define the rules to be as flexible as you might like. Whenever an element can contain text, there are only two valid cases. Either the element contains nothing but text, such as

```
<!ELEMENT name (#PCDATA)>
```

or the element contains *any combination of text and tags in any order*, such as

```
<!ELEMENT para (#PCDATA|em|strong|code)*>
```

It is not legal to specify any other types of rules that contain #PCDATA. For example, the following is illegal:

```
<!ELEMENT captionedImage (image,#PCDATA)>
```

You have to rewrite such a rule, either by introducing another `caption` element or by allowing any combination of `image` elements and text.

This restriction simplifies the job of the XML parser when parsing *mixed content* (a mixture of tags and text). Since you lose some control by allowing mixed content, it is best to design DTDs so that all elements contain either other elements or nothing but text.

NOTE: Actually, it isn't quite true that you can specify arbitrary regular expressions of elements in a DTD rule. An XML parser may reject certain complex rule sets that lead to nondeterministic parsing. For example, a regular expression $((x,y)|(x,z))$ is nondeterministic. When the parser sees x , it doesn't know which of the two alternatives to take. This expression can be rewritten in a deterministic form as $(x,(y|z))$. However, some expressions can't be reformulated, such as $((x,y)^*|x?)$. The parser in the Java XML library gives no warnings when presented with an ambiguous DTD; it simply picks the first matching alternative when parsing, which causes it to reject some correct inputs. The parser is well within its rights to do so because the XML standard allows a parser to assume that the DTD is unambiguous.

In practice, this isn't an issue over which you should lose sleep, because most DTDs are so simple that you will never run into ambiguity problems.

You can also specify rules to describe the legal attributes of elements. The general syntax is

```
<!ATTLIST element attribute type default>
```

Table 3.2 shows the legal attribute types, and Table 3.3 shows the syntax for the defaults.

Table 3.2 Attribute Types

Type	Meaning
CDATA	Any character string
($A_1 A_2 \dots A_n$)	One of the string attributes A_1, A_2, \dots, A_n
NMTOKEN, NMTOKENS	One or more name tokens
ID	A unique ID
IDREF, IDREFS	One or more references to a unique ID
ENTITY, ENTITIES	One or more unparsed entities

Table 3.3 Attribute Defaults

Default	Meaning
#REQUIRED	Attribute is required.
#IMPLIED	Attribute is optional.
A	Attribute is optional; the parser reports it to be A if it is not specified.
#FIXED A	The attribute must either be unspecified or A ; in either case, the parser reports it to be A .

Here are two typical attribute specifications:

```
<!ATTLIST font style (plain|bold|italic|bold-italic) "plain">
<!ATTLIST size unit CDATA #IMPLIED>
```

The first specification describes the `style` attribute of a `font` element. There are four legal attribute values, and the default value is `plain`. The second specification expresses that the `unit` attribute of the `size` element can contain any character data sequence.

NOTE: We generally recommend the use of elements, not attributes, to describe data. Thus, the font style should be a separate element, such as `<style>plain</style> . . .`. However, attributes have an undeniable advantage for enumerated types because the parser can verify that the values are legal. For example, if the font style is an attribute, the parser checks that it is one of the four allowed values, and supplies a default if no value was given.

The handling of a `CDATA` attribute value is subtly different from the processing of `#PCDATA` that you have seen before, and quite unrelated to the `<! [CDATA[. . .]]>` sections. The attribute value is first *normalized*—that is, the parser processes character and entity references (such as `é` or `<`) and replaces whitespace with spaces.

An `NMTOKEN` (or name token) is similar to `CDATA`, but most nonalphanumeric characters and internal whitespace are disallowed, and the parser removes leading and trailing whitespace. `NMTOKENS` is a whitespace-separated list of name tokens.

The `ID` construct is quite useful. An `ID` is a name token that must be unique in the document—the parser checks the uniqueness. You will see an application in the next sample program. An `IDREF` is a reference to an `ID` that exists in the same document, which the parser also checks. `IDREFS` is a whitespace-separated list of `ID` references.

An `ENTITY` attribute value refers to an “unparsed external entity.” That is a holdover from SGML that is rarely used in practice. The annotated XML specification at www.xml.com/axml/axml.html has an example.

A DTD can also define *entities*, or abbreviations that are replaced during parsing. You can find a good example for the use of entities in the user interface descriptions of the Firefox browser. Those descriptions are formatted in XML and contain entity definitions such as

```
<!ENTITY back.label "Back">
```

Elsewhere, text can contain an entity reference, for example:

```
<menuitem label=&back.label;"/>
```

The parser replaces the entity reference with the replacement string. To internationalize the application, only the string in the entity definition needs to be changed. Other uses of entities are more complex and less common; look at the XML specification for details.

This concludes the introduction to DTDs. Now that you have seen how to use DTDs, you can configure your parser to take advantage of them.

First, tell the document builder factory to turn on validation:

```
factory.setValidating(true);
```

All builders produced by this factory validate their input against a DTD. The most useful benefit of validation is ignoring whitespace in element content. For example, consider the XML fragment

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

A nonvalidating parser reports the whitespace between the `font`, `name`, and `size` elements because it has no way of knowing if the children of `font` are

```
(name,size)
(#PCDATA,name,size)*
```

or perhaps

ANY

Once the DTD specifies that the children are `(name,size)`, the parser knows that the whitespace between them is not text. Call

```
factory.setIgnoringElementContentWhitespace(true);
```

and the builder will stop reporting the whitespace in text nodes. That means you can now *rely on* the fact that a `font` node has two children. You no longer need to program a tedious loop:

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element) child;
        if (childElement.getTagName().equals("name")) . . .
        else if (childElement.getTagName().equals("size")) . . .
    }
}
```

Instead, you can simply access the first and second child:

```
Element nameElement = (Element) children.item(0);
Element sizeElement = (Element) children.item(1);
```

That is why DTDs are so useful. You don't overload your program with rule-checking code—the parser has already done that work by the time you get the document.



TIP: Many programmers who start using XML are uncomfortable with validation and end up analyzing the DOM tree on the fly. If you need to convince colleagues of the benefit of using validated documents, show them the two coding alternatives—it should win them over.

When the parser reports an error, your application will want to do something about it—log it, show it to the user, or throw an exception to abandon the parsing. Therefore, you should install an error handler whenever you use validation. Supply an object that implements the `ErrorHandler` interface. That interface has three methods:

```
void warning(SAXParseException exception)
void error(SAXParseException exception)
void fatalError(SAXParseException exception)
```

Install the error handler with the `setErrorHandler` method of the `DocumentBuilder` class:

```
builder.setErrorHandler(handler);
```

`javax.xml.parsers.DocumentBuilder 1.4`

- `void setEntityResolver(EntityResolver resolver)`
sets the resolver to locate entities that are referenced in the XML documents being parsed.
- `void setErrorHandler(ErrorHandler handler)`
sets the handler to report errors and warnings that occur during parsing.

`org.xml.sax.EntityResolver 1.4`

- `public InputSource resolveEntity(String publicID, String systemID)`
returns an input source that contains the data referenced by the given ID(s), or `null` to indicate that this resolver doesn't know how to resolve the particular name. The `publicID` parameter may be `null` if no public ID was supplied.

org.xml.sax.InputSource 1.4

- `InputSource(InputStream in)`
- `InputSource(Reader in)`
- `InputSource(String systemID)`

constructs an input source from a stream, reader, or system ID (usually a relative or absolute URL).

org.xml.sax.ErrorHandler 1.4

- `void fatalError(SAXParseException exception)`
- `void error(SAXParseException exception)`
- `void warning(SAXParseException exception)`

Override these methods to provide handlers for fatal errors, nonfatal errors, and warnings.

org.xml.sax.SAXParseException 1.4

- `int getLineNumber()`
- `int getColumnNumber()`

return the line and column numbers of the end of the processed input that caused the exception.

javax.xml.parsers.DocumentBuilderFactory 1.4

- `boolean isValidating()`
 - `void setValidating(boolean value)`
- gets or sets the validating property of the factory. If set to true, the parsers that this factory generates validate their input.
- `boolean isIgnoringElementContentWhitespace()`
 - `void setIgnoringElementContentWhitespace(boolean value)`

gets or sets the ignoringElementContentWhitespace property of the factory. If set to true, the parsers that this factory generates ignore whitespace between element nodes that don't have mixed content (i.e., a mixture of elements and #PCDATA).

3.3.2 XML Schema

XML Schema is quite a bit more complex than the DTD syntax, so we will only cover the basics. For more information, we recommend the tutorial at www.w3.org/TR/xmlschema-0.

To reference a Schema file in a document, add attributes to the root element, for example:

```
<?xml version="1.0"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="config.xsd">
    .
    .
</configuration>
```

This declaration states that the schema file config.xsd should be used to validate the document. If your document uses namespaces, the syntax is a bit more complex—see the XML Schema tutorial for details. (The prefix xsi is a *namespace alias*; see Section 3.5, “Using Namespaces,” on p. 196 for more information.)

A schema defines a *type* for each element. The type can be a *simple type*—a string with formatting restrictions—or a *complex type*. Some simple types are built into XML Schema, including

```
xsd:string
xsd:int
xsd:boolean
```

NOTE: We use the prefix xsd: to denote the XML Schema Definition namespace. Some authors use the prefix xs: instead.

You can define your own simple types. For example, here is an enumerated type:

```
<xsd:simpleType name="StyleType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="PLAIN" />
        <xsd:enumeration value="BOLD" />
        <xsd:enumeration value="ITALIC" />
        <xsd:enumeration value="BOLD_ITALIC" />
    </xsd:restriction>
</xsd:simpleType>
```

When you define an element, you specify its type:

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="size" type="xsd:int"/>
<xsd:element name="style" type="StyleType"/>
```

The type constrains the element content. For example, the elements

```
<size>10</size>
<style>PLAIN</style>
```

will validate correctly, but the elements

```
<size>default</size>
<style>SLANTED</style>
```

will be rejected by the parser.

You can compose types into complex types, for example:

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element ref="name"/>
    <xsd:element ref="size"/>
    <xsd:element ref="style"/>
  </xsd:sequence>
</xsd:complexType>
```

A *FontType* is a sequence of *name*, *size*, and *style* elements. In this type definition, we use the *ref* attribute and refer to definitions that are located elsewhere in the schema. You can also nest definitions, like this:

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
    <xsd:element name="style" type="StyleType">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="PLAIN" />
          <xsd:enumeration value="BOLD" />
          <xsd:enumeration value="ITALIC" />
          <xsd:enumeration value="BOLD_ITALIC" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Note the *anonymous type definition* of the *style* element.

The *xsd:sequence* construct is the equivalent of the concatenation notation in DTDs. The *xsd:choice* construct is the equivalent of the | operator. For example,

```
<xsd:complexType name="contactinfo">
  <xsd:choice>
    <xsd:element ref="email"/>
    <xsd:element ref="phone"/>
  </xsd:choice>
</xsd:complexType>
```

This is the equivalent of the DTD type `email|phone`.

To allow repeated elements, use the `minoccurs` and `maxoccurs` attributes. For example, the equivalent of the DTD type `item*` is

```
<xsd:element name="item" type=". ." minoccurs="0" maxoccurs="unbounded">
```

To specify attributes, add `xsd:attribute` elements to `complexType` definitions:

```
<xsd:element name="size">
  <xsd:complexType>
    .
    .
    <xsd:attribute name="unit" type="xsd:string" use="optional" default="cm"/>
  </xsd:complexType>
</xsd:element>
```

This is the equivalent of the DTD statement

```
<!ATTLIST size unit CDATA #IMPLIED "cm">
```

Enclose element and type definitions of your schema inside an `xsd:schema` element:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  .
  .
</xsd:schema>
```

Parsing an XML file with a schema is similar to parsing a file with a DTD, but with three differences:

1. You need to turn on support for namespaces, even if you don't use them in your XML files.

```
factory.setNamespaceAware(true);
```

2. You need to prepare the factory for handling schemas, with the following magic incantation:

```
final String JAXP_SCHEMA_LANGUAGE = "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

3. The parser *does not discard element content whitespace*. This is a definite annoyance, and there is disagreement whether or not it is an actual bug. See the code in Listing 3.4 on p. 185 for a workaround.

3.3.3 A Practical Example

In this section, we work through a practical example that shows the use of XML in a realistic setting. Recall from Volume I, Chapter 12 that the `GridBagLayout` is the most useful layout manager for Swing components. However, it is feared—not just for its complexity but also for the programming tedium. It would be much more convenient to put a layout description into a text file instead of producing large amounts of repetitive code. In this section, you will see how to use XML to describe a grid bag layout and how to parse the layout files.

A grid bag is made up of rows and columns, very similar to an HTML table. Similar to an HTML table, we describe it as a sequence of rows, each of which contains cells:

```
<gridbag>
  <row>
    <cell> . . . </cell>
    <cell> . . . </cell>
    ...
  </row>
  <row>
    <cell> . . . </cell>
    <cell> . . . </cell>
    ...
  </row>
  ...
</gridbag>
```

The `gridbag.dtd` specifies these rules:

```
<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
```

Some cells can span multiple rows and columns. In the grid bag layout, that is achieved by setting the `gridwidth` and `gridheight` constraints to values larger than 1. We will use attributes of the same name:

```
<cell gridwidth="2" gridheight="2">
```

Similarly, we can use attributes for the other grid bag constraints `fill`, `anchor`, `gridx`, `gridy`, `weightx`, `weighty`, `ipadx`, and `ipady`. (We don't handle the `insets` constraint because its value is not a simple type, but it would be straightforward to support it.) For example,

```
<cell fill="HORIZONTAL" anchor="NORTH">
```

For most of these attributes, we provide the same defaults as the no-argument constructor of the `GridBagConstraints` class:

```
<!ATTLIST cell gridwidth CDATA "1">
<!ATTLIST cell gridheight CDATA "1">
<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
<!ATTLIST cell anchor (CENTER|NORTH|NORTHEAST|EAST
    |SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
    . . .
```

The `gridx` and `gridy` values get special treatment because it would be tedious and somewhat error-prone to specify them by hand. Supplying them is optional:

```
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
```

If they are not supplied, the program determines them according to the following heuristic: In column 0, the default `gridx` is 0. Otherwise, it is the preceding `gridx` plus the preceding `gridwidth`. The default `gridy` is always the same as the row number. Thus, you don't have to specify `gridx` and `gridy` in the most common cases where a component spans multiple rows. However, if a component spans multiple columns, you must specify `gridx` whenever you skip over that component.

NOTE: Grid bag experts might wonder why we don't use the `RELATIVE` and `REMAINDER` mechanism to let the grid bag layout automatically determine the `gridx` and `gridy` positions. We tried, but no amount of fussing would produce the layout of the font dialog example of Figure 3.4. Reading through the `GridBagLayout` source code, it is apparent that the algorithm just won't do the heavy lifting required to recover the absolute positions.

The program parses the attributes and sets the grid bag constraints. For example, to read the grid width, the program contains a single statement:

```
constraints.gridx = Integer.parseInt(e.getAttribute("gridwidth"));
```

The program need not worry about a missing attribute because the parser automatically supplies the default value if no other value was specified in the document.

To test whether a `gridx` or `gridy` attribute was specified, we call the `getAttribute` method and check if it returns the empty string:

```
String value = e.getAttribute("gridx");
if (value.length() == 0) // use default
    constraints.gridx = r;
else
    constraints.gridx = Integer.parseInt(value);
```

We found it convenient to allow arbitrary objects inside cells. That lets us specify noncomponent types such as borders. We only require that the objects belong to



Figure 3.4 A font dialog defined by an XML layout

a class that has a no-argument constructor and getter/setter pairs for reading and writing properties. (Such a class is called a JavaBean.)

A bean is defined by a class name and zero or more properties:

```
<!ELEMENT bean (class, property*)>
<!ELEMENT class (#PCDATA)>
```

A property contains a name and a value:

```
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
```

The value is an integer, boolean, string, or another bean:

```
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Here is a typical example—a JLabel whose text property is set to the string "Face:":

```
<bean>
  <class>javax.swing.JLabel</class>
  <property>
```

```
<name>text</name>
<value><string>Face: </string></value>
</property>
</bean>
```

It seems like a bother to surround a string with the `<string>` tag. Why not just use `#PCDATA` for strings and leave the tags for the other types? Because then we would need to use mixed content and weaken the rule for the `value` element to

```
<!ELEMENT value (#PCDATA|int|boolean|bean)*>
```

Such a rule would allow an arbitrary mixture of text and tags.

The program sets a property by using the `BeanInfo` class. `BeanInfo` enumerates the property descriptors of the bean. We search for the property with the matching name, and then call its setter method with the supplied value.

When our program reads in a user interface description, it has enough information to construct and arrange the user interface components. But, of course, the interface is not alive—no event listeners have been attached. To add event listeners, we have to locate the components. For that reason, we support an optional attribute of type `ID` for each bean:

```
<!ATTLIST bean id ID #IMPLIED>
```

For example, here is a combo box with an ID:

```
<bean id="face">
  <class>javax.swing.JComboBox</class>
</bean>
```

Recall that the parser checks that IDs are unique.

A programmer can attach event handlers like this:

```
gridbag = new GridBagPanel("fontdialog.xml");
setContentPanel(gridbag);
JComboBox face = (JComboBox) gridbag.get("face");
face.addListener(listener);
```

NOTE: In this example, we only use XML to describe the component layout and leave it to programmers to attach the event handlers in the Java code. You could go a step further and add the code to the XML description. The most promising approach is to use a scripting language such as JavaScript for the code. If you want to add that enhancement, check out the Nashorn JavaScript interpreter described in Chapter 8.

The program in Listing 3.2 shows how to use the `GridBagPane` class to do all the boring work of setting up the grid bag layout. The layout is defined in Listing 3.4; Figure 3.4 shows the result. The program only initializes the combo boxes (which are too complex for the bean property-setting mechanism that the `GridBagPane` supports) and attaches event listeners. The `GridBagPane` class in Listing 3.3 parses the XML file, constructs the components, and lays them out. Listing 3.5 shows the DTD.

The program can also process a schema instead of a DTD if you choose a file that contains the string `-schema`.

Listing 3.6 contains the schema.

This example is a typical use of XML. The XML format is robust enough to express complex relationships. The XML parser adds value by taking over the routine job of validity checking and supplying defaults.

Listing 3.2 `read/GridBagTest.java`

```
1 package read;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import javax.swing.*;
7
8 /**
9  * This program shows how to use an XML file to describe a gridbag layout.
10 * @version 1.12 2016-04-27
11 * @author Cay Horstmann
12 */
13 public class GridBagTest
14 {
15     public static void main(String[] args)
16     {
17         EventQueue.invokeLater(() ->
18         {
19             JFileChooser chooser = new JFileChooser(".");
20             chooser.showOpenDialog(null);
21             File file = chooser.getSelectedFile();
22             JFrame frame = new FontFrame(file);
23             frame.setTitle("GridBagTest");
24             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25             frame.setVisible(true);
26         });
27     }
28 }
```

(Continues)

Listing 3.2 (*Continued*)

```
29 /**
30  * This frame contains a font selection dialog that is described by an XML file.
31  * @param filename the file containing the user interface components for the dialog
32  */
33
34 class FontFrame extends JFrame
35 {
36     private GridBagPane gridbag;
37     private JComboBox<String> face;
38     private JComboBox<String> size;
39     private JCheckBox bold;
40     private JCheckBox italic;
41
42     @SuppressWarnings("unchecked")
43     public FontFrame(File file)
44     {
45         gridbag = new GridBagPane(file);
46         add(gridbag);
47
48         face = (JComboBox<String>) gridbag.get("face");
49         size = (JComboBox<String>) gridbag.get("size");
50         bold = (JCheckBox) gridbag.get("bold");
51         italic = (JCheckBox) gridbag.get("italic");
52
53         face.setModel(new DefaultComboBoxModel<String>(new String[] { "Serif",
54             "SansSerif", "Monospaced", "Dialog", "DialogInput" }));
55
56         size.setModel(new DefaultComboBoxModel<String>(new String[] { "8",
57             "10", "12", "15", "18", "24", "36", "48" }));
58
59         ActionListener listener = event -> setSample();
60
61         face.addActionListener(listener);
62         size.addActionListener(listener);
63         bold.addActionListener(listener);
64         italic.addActionListener(listener);
65
66         setSample();
67         pack();
68     }
69
70 /**
71  * This method sets the text sample to the selected font.
72  */
73 public void setSample()
74 {
```

```
75     String fontFace = face.getItemAt(face.getSelectedIndex());
76     int fontSize = Integer.parseInt(size.getItemAt(size.getSelectedIndex()));
77     JTextArea sample = (JTextArea) gridbag.get("sample");
78     int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
79         + (italic.isSelected() ? Font.ITALIC : 0);
80
81     sample.setFont(new Font(fontFace, fontStyle, fontSize));
82     sample.repaint();
83 }
84 }
```

Listing 3.3 `read/GridBagPane.java`

```
1 package read;
2
3 import java.awt.*;
4 import java.beans.*;
5 import java.io.*;
6 import java.lang.reflect.*;
7 import javax.swing.*;
8 import javax.xml.parsers.*;
9 import org.w3c.dom.*;
10
11 /**
12 * This panel uses an XML file to describe its components and their grid bag layout positions.
13 */
14 public class GridBagPane extends JPanel
15 {
16     private GridBagConstraints constraints;
17
18     /**
19      * Constructs a grid bag pane.
20      * @param filename the name of the XML file that describes the pane's components and their
21      * positions
22      */
23     public GridBagPane(File file)
24     {
25         setLayout(new GridBagLayout());
26         constraints = new GridBagConstraints();
27
28         try
29         {
30             DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
31             factory.setValidating(true);
32
33             if (file.toString().contains("-schema"))
34             {
```

(Continues)

Listing 3.3 (Continued)

```
35         factory.setNamespaceAware(true);
36         final String JAXP_SCHEMA_LANGUAGE =
37             "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
38         final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
39         factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
40     }
41
42     factory.setIgnoringElementContentWhitespace(true);
43
44     DocumentBuilder builder = factory.newDocumentBuilder();
45     Document doc = builder.parse(file);
46     parseGridbag(doc.getDocumentElement());
47 }
48 catch (Exception e)
49 {
50     e.printStackTrace();
51 }
52 }

53 /**
54 * Gets a component with a given name.
55 * @param name a component name
56 * @return the component with the given name, or null if no component in this grid bag pane has
57 * the given name
58 */
59 public Component get(String name)
60 {
61     Component[] components = getComponents();
62     for (int i = 0; i < components.length; i++)
63     {
64         if (components[i].getName().equals(name)) return components[i];
65     }
66     return null;
67 }
68

69 /**
70 * Parses a gridbag element.
71 * @param e a gridbag element
72 */
73 private void parseGridbag(Element e)
74 {
75     NodeList rows = e.getChildNodes();
76     for (int i = 0; i < rows.getLength(); i++)
77     {
78         Element row = (Element) rows.item(i);
79         NodeList cells = row.getChildNodes();
```

```
81         for (int j = 0; j < cells.getLength(); j++)
82     {
83         Element cell = (Element) cells.item(j);
84         parseCell(cell, i, j);
85     }
86 }
87
88 /**
89 * Parses a cell element.
90 * @param e a cell element
91 * @param r the row of the cell
92 * @param c the column of the cell
93 */
94 private void parseCell(Element e, int r, int c)
95 {
96     // get attributes
97
98     String value = e.getAttribute("gridx");
99     if (value.length() == 0) // use default
100    {
101        if (c == 0) constraints.gridx = 0;
102        else constraints.gridx += constraints.gridwidth;
103    }
104    else constraints.gridx = Integer.parseInt(value);
105
106    value = e.getAttribute("gridy");
107    if (value.length() == 0) // use default
108        constraints.gridy = r;
109    else constraints.gridy = Integer.parseInt(value);
110
111    constraints.gridwidth = Integer.parseInt(e.getAttribute("gridwidth"));
112    constraints.gridheight = Integer.parseInt(e.getAttribute("gridheight"));
113    constraints.weightx = Integer.parseInt(e.getAttribute("weightx"));
114    constraints.weighty = Integer.parseInt(e.getAttribute("weighty"));
115    constraints.ipadx = Integer.parseInt(e.getAttribute("ipadx"));
116    constraints.ipady = Integer.parseInt(e.getAttribute("ipady"));
117
118    // use reflection to get integer values of static fields
119    Class<GridBagConstraints> cl = GridBagConstraints.class;
120
121    try
122    {
123        String name = e.getAttribute("fill");
124        Field f = cl.getField(name);
125        constraints.fill = f.getInt(cl);
126
127        name = e.getAttribute("anchor");
128        f = cl.getField(name);
```

(Continues)

Listing 3.3 (Continued)

```
130         constraints.anchor = f.getInt(c1);
131     }
132     catch (Exception ex) // the reflection methods can throw various exceptions
133     {
134         ex.printStackTrace();
135     }
136
137     Component comp = (Component) parseBean((Element) e.getFirstChild());
138     add(comp, constraints);
139 }
140
141 /**
142 * Parses a bean element.
143 * @param e a bean element
144 */
145 private Object parseBean(Element e)
146 {
147     try
148     {
149         NodeList children = e.getChildNodes();
150         Element classElement = (Element) children.item(0);
151         String className = ((Text) classElement.getFirstChild()).getData();
152
153         Class<?> cl = Class.forName(className);
154
155         Object obj = cl.newInstance();
156
157         if (obj instanceof Component) ((Component) obj).setName(e.getAttribute("id"));
158
159         for (int i = 1; i < children.getLength(); i++)
160         {
161             Node propertyElement = children.item(i);
162             Element nameElement = (Element) propertyElement.getFirstChild();
163             String propertyName = ((Text) nameElement.getFirstChild()).getData();
164
165             Element valueElement = (Element) propertyElement.getLastChild();
166             Object value = parseValue(valueElement);
167             BeanInfo beanInfo = Introspector.getBeanInfo(cl);
168            PropertyDescriptor[] descriptors = beanInfo.getPropertyDescriptors();
169             boolean done = false;
170             for (int j = 0; !done && j < descriptors.length; j++)
171             {
172                 if (descriptors[j].getName().equals(propertyName))
173                 {
174                     descriptors[j].getWriteMethod().invoke(obj, value);
175                     done = true;
176                 }
177             }
178         }
179     }
180 }
```

```
178         }
179         return obj;
180     }
181     catch (Exception ex) // the reflection methods can throw various exceptions
182     {
183         ex.printStackTrace();
184         return null;
185     }
186 }
187
188 /**
189 * Parses a value element.
190 * @param e a value element
191 */
192 private Object parseValue(Element e)
193 {
194     Element child = (Element) e.getFirstChild();
195     if (child.getTagName().equals("bean")) return parseBean(child);
196     String text = ((Text) child.getFirstChild()).getData();
197     if (child.getTagName().equals("int")) return new Integer(text);
198     else if (child.getTagName().equals("boolean")) return new Boolean(text);
199     else if (child.getTagName().equals("string")) return text;
200     else return null;
201 }
202 }
```

Listing 3.4 *read/fontdialog.xml*

```
1 <?xml version="1.0"?>
2 <!DOCTYPE gridbag SYSTEM "gridbag.dtd">
3 <gridbag>
4   <row>
5     <cell anchor="EAST">
6       <bean>
7         <class>javax.swing.JLabel</class>
8         <property>
9           <name>text</name>
10          <value><string>Face: </string></value>
11        </property>
12      </bean>
13    </cell>
14    <cell fill="HORIZONTAL" weightx="100">
15      <bean id="face">
16        <class>javax.swing.JComboBox</class>
17      </bean>
18    </cell>
19    <cell gridheight="4" fill="BOTH" weightx="100" weighty="100">
20      <bean id="sample">
```

(Continues)

Listing 3.4 (Continued)

```
21      <class>javax.swing.JTextArea</class>
22      <property>
23          <name>text</name>
24          <value><string>The quick brown fox jumps over the lazy dog</string></value>
25      </property>
26      <property>
27          <name>editable</name>
28          <value><boolean>false</boolean></value>
29      </property>
30      <property>
31          <name>rows</name>
32          <value><int>8</int></value>
33      </property>
34      <property>
35          <name>columns</name>
36          <value><int>20</int></value>
37      </property>
38      <property>
39          <name>lineWrap</name>
40          <value><boolean>true</boolean></value>
41      </property>
42      <property>
43          <name>border</name>
44          <value>
45              <bean>
46                  <class>javax.swing.border.EtchedBorder</class>
47              </bean>
48          </value>
49      </property>
50  </bean>
51 </cell>
52 </row>
53 <row>
54     <cell anchor="EAST">
55         <bean>
56             <class>javax.swing.JLabel</class>
57             <property>
58                 <name>text</name>
59                 <value><string>Size: </string></value>
60             </property>
61         </bean>
62     </cell>
63     <cell fill="HORIZONTAL" weightx="100">
64         <bean id="size">
65             <class>javax.swing.JComboBox</class>
66         </bean>
67     </cell>
68 </row>
```

```
69 <row>
70     <cell gridwidth="2" weighty="100">
71         <bean id="bold">
72             <class>javax.swing.JCheckBox</class>
73             <property>
74                 <name>text</name>
75                 <value><string>Bold</string></value>
76             </property>
77         </bean>
78     </cell>
79 </row>
80 <row>
81     <cell gridwidth="2" weighty="100">
82         <bean id="italic">
83             <class>javax.swing.JCheckBox</class>
84             <property>
85                 <name>text</name>
86                 <value><string>Italic</string></value>
87             </property>
88         </bean>
89     </cell>
90 </row>
91 </gridbag>
```

Listing 3.5 read/gridbag.dtd

```
1 <!ELEMENT gridbag (row)*>
2 <!ELEMENT row (cell)*>
3 <!ELEMENT cell (bean)>
4 <!ATTLIST cell gridx CDATA #IMPLIED>
5 <!ATTLIST cell gridy CDATA #IMPLIED>
6 <!ATTLIST cell gridwidth CDATA "1">
7 <!ATTLIST cell gridheight CDATA "1">
8 <!ATTLIST cell weightx CDATA "0">
9 <!ATTLIST cell weighty CDATA "0">
10 <!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
11 <!ATTLIST cell anchor
12     (CENTER|NORTH|NORTHEAST|EAST|SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
13 <!ATTLIST cell ipadx CDATA "0">
14 <!ATTLIST cell ipady CDATA "0">
15
16 <!ELEMENT bean (class, property*)>
17 <!ATTLIST bean id ID #IMPLIED>
18
19 <!ELEMENT class (#PCDATA)>
20 <!ELEMENT property (name, value)>
21 <!ELEMENT name (#PCDATA)>
22 <!ELEMENT value (int|string|boolean|bean)>
```

(Continues)

Listing 3.5 (Continued)

```
23 <!ELEMENT int (#PCDATA)>
24 <!ELEMENT string (#PCDATA)>
25 <!ELEMENT boolean (#PCDATA)>
```

Listing 3.6 *read/gridbag.xsd*

```
1 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2
3     <xsd:element name="gridbag" type="GridBagType"/>
4
5     <xsd:element name="bean" type="BeanType"/>
6
7     <xsd:complexType name="GridBagType">
8         <xsd:sequence>
9             <xsd:element name="row" type="RowType" minOccurs="0" maxOccurs="unbounded"/>
10            </xsd:sequence>
11        </xsd:complexType>
12
13    <xsd:complexType name="RowType">
14        <xsd:sequence>
15            <xsd:element name="cell" type="CellType" minOccurs="0" maxOccurs="unbounded"/>
16        </xsd:sequence>
17    </xsd:complexType>
18
19    <xsd:complexType name="CellType">
20        <xsd:sequence>
21            <xsd:element ref="bean"/>
22        </xsd:sequence>
23        <xsd:attribute name="gridx" type="xsd:int" use="optional"/>
24        <xsd:attribute name="gridy" type="xsd:int" use="optional"/>
25        <xsd:attribute name="gridwidth" type="xsd:int" use="optional" default="1" />
26        <xsd:attribute name="gridheight" type="xsd:int" use="optional" default="1" />
27        <xsd:attribute name="weightx" type="xsd:int" use="optional" default="0" />
28        <xsd:attribute name="weighty" type="xsd:int" use="optional" default="0" />
29        <xsd:attribute name="fill" use="optional" default="NONE">
30            <xsd:simpleType>
31                <xsd:restriction base="xsd:string">
32                    <xsd:enumeration value="NONE" />
33                    <xsd:enumeration value="BOTH" />
34                    <xsd:enumeration value="HORIZONTAL" />
35                    <xsd:enumeration value="VERTICAL" />
36                </xsd:restriction>
37            </xsd:simpleType>
38        </xsd:attribute>
```

```
39     <xsd:attribute name="anchor" use="optional" default="CENTER">
40         <xsd:simpleType>
41             <xsd:restriction base="xsd:string">
42                 <xsd:enumeration value="CENTER" />
43                 <xsd:enumeration value="NORTH" />
44                 <xsd:enumeration value="NORTHEAST" />
45                 <xsd:enumeration value="EAST" />
46                 <xsd:enumeration value="SOUTHEAST" />
47                 <xsd:enumeration value="SOUTH" />
48                 <xsd:enumeration value="SOUTHWEST" />
49                 <xsd:enumeration value="WEST" />
50                 <xsd:enumeration value="NORTHWEST" />
51         </xsd:restriction>
52     </xsd:simpleType>
53 </xsd:attribute>
54 <xsd:attribute name="ipady" type="xsd:int" use="optional" default="0" />
55 <xsd:attribute name="ipadx" type="xsd:int" use="optional" default="0" />
56 </xsd:complexType>
57
58 <xsd:complexType name="BeanType">
59     <xsd:sequence>
60         <xsd:element name="class" type="xsd:string"/>
61         <xsd:element name="property" type=".PropertyType" minOccurs="0" maxOccurs="unbounded"/>
62     </xsd:sequence>
63     <xsd:attribute name="id" type="xsd:ID" use="optional" />
64 </xsd:complexType>
65
66 <xsd:complexType name=".PropertyType">
67     <xsd:sequence>
68         <xsd:element name="name" type="xsd:string"/>
69         <xsd:element name="value" type="ValueType"/>
70     </xsd:sequence>
71 </xsd:complexType>
72
73 <xsd:complexType name="ValueType">
74     <xsd:choice>
75         <xsd:element ref="bean"/>
76         <xsd:element name="int" type="xsd:int"/>
77         <xsd:element name="string" type="xsd:string"/>
78         <xsd:element name="boolean" type="xsd:boolean"/>
79     </xsd:choice>
80 </xsd:complexType>
81 </xsd:schema>
```

3.4 Locating Information with XPath

If you want to locate a specific piece of information in an XML document, it can be a bit of a hassle to navigate the nodes of the DOM tree. The XPath language makes it simple to access tree nodes. For example, suppose you have this XML document:

```
<configuration>
    ...
    <database>
        <username>dbuser</username>
        <password>secret</password>
    ...
</configuration>
```

You can get the database user name by evaluating the XPath expression

```
/configuration/database/username
```

That's a lot simpler than the plain DOM approach:

1. Get the document node.
2. Enumerate its children.
3. Locate the `database` element.
4. Locate the `username` element among its children.
5. Locate a `text` node among its children.
6. Get its data.

An XPath can describe *a set of nodes* in an XML document. For example, the XPath

```
/gridbag/row
```

describes the set of all `row` elements that are children of the `gridbag` root element. You can select a particular element with the `[]` operator:

```
/gridbag/row[1]
```

is the first row. (The index values start at 1.)

Use the `@` operator to get attribute values. The XPath expression

```
/gridbag/row[1]/cell[1]/@anchor
```

describes the `anchor` attribute of the first cell in the first row. The XPath expression

```
/gridbag/row/cell[@anchor]
```

describes all `anchor` attribute nodes of `cell` elements within `row` elements that are children of the `gridbag` root node.

There are a number of useful XPath functions. For example,

```
count(/gridbag/row)
```

returns the number of `row` children of the `gridbag` root. There are many more elaborate XPath expressions; see the specification at www.w3c.org/TR/xpath or the nifty online tutorial at www.zvon.org/xxl/XPathTutorial/General/examples.html.

Java SE 5.0 added an API to evaluate XPath expressions. First, create an XPath object from an `XPathFactory`:

```
XPathFactory xpfactory = XPathFactory.newInstance();
path = xpfactory.newXPath();
```

Then, call the `evaluate` method to evaluate XPath expressions:

```
String username = path.evaluate("/configuration/database/username", doc);
```

You can use the same XPath object to evaluate multiple expressions.

This form of the `evaluate` method returns a string result. It is suitable for retrieving text, such as the text of the `username` node in the preceding example. If an XPath expression yields a node set, make a call such as the following:

```
NodeList nodes = (NodeList) path.evaluate("/gridbag/row", doc, XPathConstants.NODESET);
```

If the result is a single node, use `XPathConstants.NODE` instead:

```
Node node = (Node) path.evaluate("/gridbag/row[1]", doc, XPathConstants.NODE);
```

If the result is a number, use `XPathConstants.NUMBER`:

```
int count = ((Number) path.evaluate("count(/gridbag/row)", doc, XPathConstants.NUMBER)).intValue();
```

You don't have to start the search at the document root; you can start at any node or node list. For example, if you have a node from a previous evaluation, you can call

```
result = path.evaluate(expression, node);
```

The program in Listing 3.7 demonstrates evaluation of XPath expressions. Load an XML file and type an expression. Select the expression type and click the Evaluate button. The result of the expression is displayed at the bottom of the frame (see Figure 3.5).

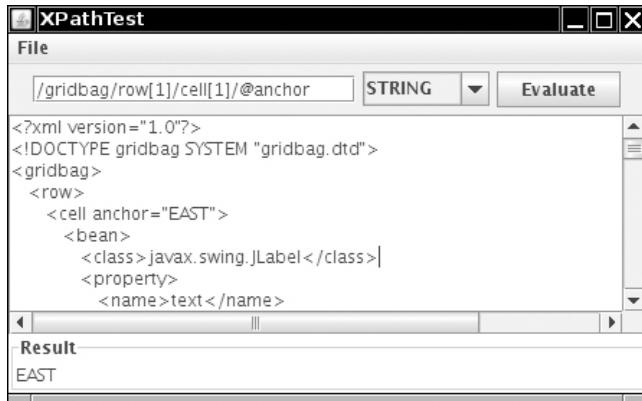


Figure 3.5 Evaluating XPath expressions

Listing 3.7 `xpath/XPathTester.java`

```
1 package xpath;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.nio.file.*;
7 import java.util.*;
8 import javax.swing.*;
9 import javax.swing.border.*;
10 import javax.xml.namespace.*;
11 import javax.xml.parsers.*;
12 import javax.xml.xpath.*;
13 import org.w3c.dom.*;
14 import org.xml.sax.*;
15
16 /**
17 * This program evaluates XPath expressions.
18 * @version 1.02 2016-05-10
19 * @author Cay Horstmann
20 */
21 public class XPathTester
22 {
23     public static void main(String[] args)
24     {
25         EventQueue.invokeLater(() ->
26             {
```

```
27     JFrame frame = new XPathFrame();
28     frame.setTitle("XPathTest");
29     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30     frame.setVisible(true);
31 }
32 }
33 }
34 /**
35 * This frame shows an XML document, a panel to type an XPath expression, and a text field to
36 * display the result.
37 */
38 class XPathFrame extends JFrame
39 {
40     private DocumentBuilder builder;
41     private Document doc;
42     private XPath path;
43     private JTextField expression;
44     private JTextField result;
45     private JTextArea docText;
46     private JComboBox<String> typeCombo;
47
48     public XPathFrame()
49     {
50         JMenu fileMenu = new JMenu("File");
51         JMenuItem openItem = new JMenuItem("Open");
52         openItem.addActionListener(event -> openFile());
53         fileMenu.add(openItem);
54
55         JMenuItem exitItem = new JMenuItem("Exit");
56         exitItem.addActionListener(event -> System.exit(0));
57         fileMenu.add(exitItem);
58
59         JMenuBar menuBar = new JMenuBar();
60         menuBar.add(fileMenu);
61         setJMenuBar(menuBar);
62
63         ActionListener listener = event -> evaluate();
64         expression = new JTextField(20);
65         expression.addActionListener(listener);
66         JButton evaluateButton = new JButton("Evaluate");
67         evaluateButton.addActionListener(listener);
68
69         typeCombo = new JComboBox<String>(new String[] {
70             "STRING", "NODE", "NODESET", "NUMBER", "BOOLEAN" });
71         typeCombo.setSelectedItem("STRING");
72
73         JPanel panel = new JPanel();
```

(Continues)

Listing 3.7 (Continued)

```
75     panel.add(expression);
76     panel.add(typeCombo);
77     panel.add(evaluateButton);
78     docText = new JTextArea(10, 40);
79     result = new JTextField();
80     result.setBorder(new TitledBorder("Result"));
81
82     add(panel, BorderLayout.NORTH);
83     add(new JScrollPane(docText), BorderLayout.CENTER);
84     add(result, BorderLayout.SOUTH);
85
86     try
87     {
88         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
89         builder = factory.newDocumentBuilder();
90     }
91     catch (ParserConfigurationException e)
92     {
93         JOptionPane.showMessageDialog(this, e);
94     }
95
96     XPathFactory xpfactory = XPathFactory.newInstance();
97     path = xpfactory.newXPath();
98     pack();
99 }
100 /**
101 * Open a file and load the document.
102 */
103 public void openFile()
104 {
105     JFileChooser chooser = new JFileChooser();
106     chooser.setCurrentDirectory(new File("xpath"));
107
108     chooser.setFileFilter(
109         new javax.swing.filechooser.FileNameExtensionFilter("XML files", "xml"));
110     int r = chooser.showOpenDialog(this);
111     if (r != JFileChooser.APPROVE_OPTION) return;
112     File file = chooser.getSelectedFile();
113     try
114     {
115         docText.setText(new String(File.readAllBytes(file.toPath())));
116         doc = builder.parse(file);
117     }
118 }
```

```
119     catch (IOException e)
120     {
121         JOptionPane.showMessageDialog(this, e);
122     }
123     catch (SAXException e)
124     {
125         JOptionPane.showMessageDialog(this, e);
126     }
127 }
128
129 public void evaluate()
130 {
131     try
132     {
133         String typeName = (String) typeCombo.getSelectedItem();
134         QName returnType = (QName) XPathConstants.class.getField(typeName).get(null);
135         Object evalResult = path.evaluate(expression.getText(), doc, returnType);
136         if (typeName.equals("NODESET"))
137         {
138             NodeList list = (NodeList) evalResult;
139             // Can't use String.join since NodeList isn't Iterable
140             StringJoiner joiner = new StringJoiner(", ", "{", "}");
141             for (int i = 0; i < list.getLength(); i++)
142                 joiner.add("'" + list.item(i));
143             result.setText("'" + joiner);
144         }
145         else result.setText("'" + evalResult);
146     }
147     catch (XPathExpressionException e)
148     {
149         result.setText("'" + e);
150     }
151     catch (Exception e) // reflection exception
152     {
153         e.printStackTrace();
154     }
155 }
156 }
```

javax.xml.xpath.XPathFactory 5.0

- static `XPathFactory newInstance()`
returns an `XPathFactory` instance for creating `XPath` objects.
- `XPath newPath()`
constructs an `XPath` object for evaluating `XPath` expressions.

javax.xml.xpath.XPath 5.0

- `String evaluate(String expression, Object startingPoint)`

evaluates an expression, beginning at the given starting point. The starting point can be a node or node list. If the result is a node or node set, the returned string consists of the data of all text node children.

- `Object evaluate(String expression, Object startingPoint, QName resultType)`

evaluates an expression, beginning at the given starting point. The starting point can be a node or node list. The `resultType` is one of the constants `STRING`, `NODE`, `NODESET`, `NUMBER`, or `BOOLEAN` in the `XPathConstants` class. The return value is a `String`, `Node`, `NodeList`, `Number`, or `Boolean`.

3.5 Using Namespaces

The Java language uses packages to avoid name clashes. Programmers can use the same name for different classes as long as they aren't in the same package. XML has a similar *namespace* mechanism for element and attribute names.

A namespace is identified by a Uniform Resource Identifier (URI), such as

`http://www.w3.org/2001/XMLSchema`
`uuid:1c759aed-b748-475c-ab68-10679700c4f2`
`urn:com:books-r-us`

The HTTP URL form is the most common. Note that the URL is just used as an identifier string, not as a locator for a document. For example, the namespace identifiers

`http://www.horstmann.com/corejava`
`http://www.horstmann.com/corejava/index.html`

denote *different* namespaces, even though a web server would serve the same document for both URLs.

There need not be any document at a namespace URL—the XML parser doesn't attempt to find anything at that location. However, as a help to programmers who encounter a possibly unfamiliar namespace, it is customary to place a document explaining the purpose of the namespace at the URL location. For example, if you point your browser to the namespace URL for the XML Schema namespace (`http://www.w3.org/2001/XMLSchema`), you will find a document describing the XML Schema standard.

Why use HTTP URLs for namespace identifiers? It is easy to ensure that they are unique. If you choose a real URL, the host part's uniqueness is guaranteed by the domain name system. Your organization can then arrange for the uniqueness of the remainder of the URL. This is the same rationale that underlies the use of reversed domain names in Java package names.

Of course, although long namespace identifiers are good for uniqueness, you don't want to deal with long identifiers any more than you have to. In the Java programming language, you use the `import` mechanism to specify the long names of packages, and then use just the short class names. In XML, there is a similar mechanism:

```
<element xmlns="namespaceURI">
    children
</element>
```

The element and its children are now part of the given namespace.

A child can provide its own namespace, for example:

```
<element xmlns="namespaceURI1">
    <child xmlns="namespaceURI2">
        grandchildren
    </child>
    more children
</element>
```

Then the first child and the grandchildren are part of the second namespace.

This simple mechanism works well if you need only a single namespace or if the namespaces are naturally nested. Otherwise, you will want to use a second mechanism that has no analog in Java. You can have a *prefix* for a namespace—a short identifier that you choose for a particular document. Here is a typical example—the `xsd` prefix in an XML Schema file:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="gridbag" type="GridBagType"/>
    .
    .
</xsd:schema>
```

The attribute

```
xmlns:prefix="namespaceURI"
```

defines a namespace and a prefix. In our example, the prefix is the string `xsd`. Thus, `xsd:schema` really means `schema` in the namespace <http://www.w3.org/2001/XMLSchema>.

NOTE: Only child elements inherit the namespace of their parent. Attributes without an explicit prefix are never part of a namespace. Consider this contrived example:

```
<configuration xmlns="http://www.horstmann.com/corejava"
    xmlns:si="http://www.bipm.fr/enus/3_SI/si.html">
    <size value="210" si:unit="mm"/>
    .
    .
</configuration>
```

In this example, the elements `configuration` and `size` are part of the namespace with URI `http://www.horstmann.com/corejava`. The attribute `si:unit` is part of the namespace with URI `http://www.bipm.fr/enus/3_SI/si.html`. However, the attribute `value` is not part of any namespace.

You can control how the parser deals with namespaces. By default, the DOM parser of the Java XML library is not namespace-aware.

To turn on namespace handling, call the `setNamespaceAware` method of the `DocumentBuilderFactory`:

```
factory.setNamespaceAware(true);
```

Now, all builders the factory produces support namespaces. Each node has three properties:

- The *qualified name*, with a prefix, returned by `getnodeName`, `getTagName`, and so on
- The namespace URI, returned by the `getNamespaceURI` method
- The *local name*, without a prefix or a namespace, returned by the `getLocalName` method

Here is an example. Suppose the parser sees the following element:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

It then reports the following:

- Qualified name = `xsd:schema`
- Namespace URI = `http://www.w3.org/2001/XMLSchema`
- Local name = `schema`

NOTE: If namespace awareness is turned off, `getNamespaceURI` and `getLocalName` return `null`.

org.w3c.dom.Node 1.4

- `String getLocalName()`
returns the local name (without prefix), or `null` if the parser is not namespace-aware.
- `String getNamespaceURI()`
returns the namespace URI, or `null` if the node is not part of a namespace or if the parser is not namespace-aware.

javax.xml.parsers.DocumentBuilderFactory 1.4

- `boolean isNamespaceAware()`
 - `void setNamespaceAware(boolean value)`
- gets or sets the `namespaceAware` property of the factory. If set to `true`, the parsers that this factory generates are namespace-aware.

3.6 Streaming Parsers

The DOM parser reads an XML document in its entirety into a tree data structure. For most practical applications, DOM works fine. However, it can be inefficient if the document is large and if your processing algorithm is simple enough that you can analyze nodes on the fly, without having to see all of the tree structure. In these cases, you should use a streaming parser.

In the following sections, we discuss the streaming parsers supplied by the Java library: the venerable SAX parser and the more modern StAX parser that was added to Java SE 6. The SAX parser uses event callbacks, and the StAX parser provides an iterator through the parsing events. The latter is usually a bit more convenient.

3.6.1 Using the SAX Parser

The SAX parser reports events as it parses the components of the XML input, but it does not store the document in any way—it is up to the event handlers to build a data structure. In fact, the DOM parser is built on top of the SAX parser. It builds the DOM tree as it receives the parser events.

Whenever you use a SAX parser, you need a handler that defines the event actions for the various parse events. The `ContentHandler` interface defines several callback methods that the parser executes as it parses the document. Here are the most important ones:

- `startElement` and `endElement` are called each time a start tag or end tag is encountered.
- `characters` is called whenever character data are encountered.
- `startDocument` and `endDocument` are called once each, at the start and the end of the document.

For example, when parsing the fragment

```
<font>
    <name>Helvetica</name>
    <size units="pt">36</size>
</font>
```

the parser makes the following callbacks:

1. `startElement`, element name: `font`
2. `startElement`, element name: `name`
3. `characters`, content: `Helvetica`
4. `endElement`, element name: `name`
5. `startElement`, element name: `size`, attributes: `units="pt"`
6. `characters`, content: `36`
7. `endElement`, element name: `size`
8. `endElement`, element name: `font`

Your handler needs to override these methods and have them carry out whatever action you want to carry out as you parse the file. The program at the end of this section prints all links `` in an HTML file. It simply overrides the `startElement` method of the handler to check for links with name `a` and an attribute with name `href`. This is potentially useful for implementing a “web crawler”—a program that reaches more and more web pages by following links.

NOTE: HTML doesn't have to be valid XML, and many web pages deviate so much from proper XML that the example programs will not be able to parse them. However, most pages authored by the W3C are written in XHTML (an HTML dialect that is proper XML). You can use those pages to test the example program. For example, if you run

```
java SAXTest http://www.w3c.org/MarkUp
```

you will see a list of the URLs of all links on that page.

The sample program is a good example for the use of SAX. We don't care at all in which context the `a` elements occur, and there is no need to store a tree structure.

Here is how you get a SAX parser:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

You can now process a document:

```
parser.parse(source, handler);
```

Here, `source` can be a file, URL string, or input stream. The `handler` belongs to a subclass of `DefaultHandler`. The `DefaultHandler` class defines do-nothing methods for the four interfaces:

```
ContentHandler
DTDHandler
EntityResolver
ErrorHandler
```

The example program defines a handler that overrides the `startElement` method of the `ContentHandler` interface to watch out for `a` elements with an `href` attribute:

```
DefaultHandler handler = new
    DefaultHandler()
{
    public void startElement(String namespaceURI, String lname, String qname, Attributes attrs)
        throws SAXException
    {
        if (lname.equalsIgnoreCase("a") && attrs != null)
        {
            for (int i = 0; i < attrs.getLength(); i++)
            {
                String aname = attrs.getLocalName(i);
                if (aname.equalsIgnoreCase("href"))
                    System.out.println(attrs.getValue(i));
            }
        }
    }
};
```

The `startElement` method has three parameters that describe the element name. The `qname` parameter reports the qualified name of the form `prefix:localname`. If namespace processing is turned on, then the `namespaceURI` and `lname` parameters provide the namespace and local (unqualified) name.

As with the DOM parser, namespace processing is turned off by default. To activate namespace processing, call the `setNamespaceAware` method of the factory class:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();
```

In this program, we cope with another common issue. An XHTML file starts with a tag that contains a DTD reference, and the parser will want to load it. Understandably, the W3C isn't too happy to serve billions of copies of files such as www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd. At one point, they refused altogether, but at the time of this writing, they serve the DTD at a glacial pace. If you don't need to validate the document, just call

```
factory.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd", false);
```

Listing 3.8 contains the code for the web crawler program. Later in this chapter, you will see another interesting use of SAX. An easy way of turning a non-XML data source into XML is to report the SAX events that an XML parser would report. See Section 3.8, “XSL Transformations,” on p. 222 for details.

Listing 3.8 sax/SAXTest.java

```
1 package sax;
2
3 import java.io.*;
4 import java.net.*;
5 import javax.xml.parsers.*;
6 import org.xml.sax.*;
7 import org.xml.sax.helpers.*;
8
9 /**
10 * This program demonstrates how to use a SAX parser. The program prints all hyperlinks of an
11 * XHTML web page. <br>
12 * Usage: java sax.SAXTest URL
13 * @version 1.00 2001-09-29
14 * @author Cay Horstmann
15 */
16 public class SAXTest
17 {
18     public static void main(String[] args) throws Exception
19     {
20         String url;
21         if (args.length == 0)
22         {
23             url = "http://www.w3c.org";
24             System.out.println("Using " + url);
25         }
26         else url = args[0];
27
28         DefaultHandler handler = new DefaultHandler()
29         {
30             public void startElement(String namespaceURI, String lname, String qname,
31                                     Attributes attrs)
32             {
33                 if (lname.equals("a"))
34                     System.out.println("Link to " + attrs.getValue(0));
35             }
36         };
37         XMLReader reader = new SAXParser().getXMLReader();
38         reader.setContentHandler(handler);
39         reader.parse(new InputSource(url));
40     }
41 }
```

```
33         if (lname.equals("a") && attrs != null)
34     {
35         for (int i = 0; i < attrs.getLength(); i++)
36     {
37         String aname = attrs.getLocalName(i);
38         if (aname.equals("href")) System.out.println(attrs.getValue(i));
39     }
40     }
41 };
42 };
43
44 SAXParserFactory factory = SAXParserFactory.newInstance();
45 factory.setNamespaceAware(true);
46 factory.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd", false);
47 SAXParser saxParser = factory.newSAXParser();
48 InputStream in = new URL(url).openStream();
49 saxParser.parse(in, handler);
50 }
51 }
```

javax.xml.parsers.SAXParserFactory 1.4

- static SAXParserFactory newInstance()
returns an instance of the SAXParserFactory class.
- SAXParser newSAXParser()
returns an instance of the SAXParser class.
- boolean isNamespaceAware()
- void setNamespaceAware(boolean value)
gets or sets the namespaceAware property of the factory. If set to true, the parsers that this factory generates are namespace-aware.
- boolean isValidating()
- void setValidating(boolean value)
gets or sets the validating property of the factory. If set to true, the parsers that this factory generates validate their input.

javax.xml.parsers.SAXParser 1.4

- void parse(File f, DefaultHandler handler)
 - void parse(String url, DefaultHandler handler)
 - void parse(InputStream in, DefaultHandler handler)
- parses an XML document from the given file, URL, or input stream and reports parse events to the given handler.

org.xml.sax.ContentHandler 1.4

- void startDocument()
- void endDocument()

is called at the start or the end of the document.

- void startElement(String uri, String lname, String qname, Attributes attr)
- void endElement(String uri, String lname, String qname)

is called at the start or the end of an element.

Parameters: **uri** The URI of the namespace (if the parser is namespace-aware)
 lname The local name without prefix (if the parser is namespace-aware)
 qname The element name if the parser is not namespace-aware, or the qualified name with prefix if the parser reports qualified names in addition to local names

- void characters(char[] data, int start, int length)

is called when the parser reports character data.

Parameters: **data** An array of character data
 start The index of the first character in the data array that is a part of the reported characters
 length The length of the reported character string

org.xml.sax.Attributes 1.4

- int getLength()

returns the number of attributes stored in this attribute collection.

- String getLocalName(int index)

returns the local name (without prefix) of the attribute with the given index, or the empty string if the parser is not namespace-aware.

- String getURI(int index)

returns the namespace URI of the attribute with the given index, or the empty string if the node is not part of a namespace or if the parser is not namespace-aware.

- String getQName(int index)

returns the qualified name (with prefix) of the attribute with the given index, or the empty string if the qualified name is not reported by the parser.

(Continues)

org.xml.sax.Attributes 1.4 (Continued)

- String getValue(int index)
- String getValue(String qname)
- String getValue(String uri, String lname)

returns the attribute value from a given index, qualified name, or namespace URI + local name. Returns null if the value doesn't exist.

3.6.2 Using the StAX Parser

The StAX parser is a “pull parser.” Instead of installing an event handler, you simply iterate through the events, using this basic loop:

```
InputStream in = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);
while (parser.hasNext())
{
    int event = parser.next();
    Call parser methods to obtain event details
}
```

For example, when parsing the fragment

```
<font>
    <name>Helvetica</name>
    <size units="pt">36</size>
</font>
```

the parser yields the following events:

1. START_ELEMENT, element name: font
2. CHARACTERS, content: white space
3. START_ELEMENT, element name: name
4. CHARACTERS, content: Helvetica
5. END_ELEMENT, element name: name
6. CHARACTERS, content: white space
7. START_ELEMENT, element name: size
8. CHARACTERS, content: 36
9. END_ELEMENT, element name: size
10. CHARACTERS, content: white space
11. END_ELEMENT, element name: font

To analyze the attribute values, call the appropriate methods of the `XMLStreamReader` class. For example,

```
String units = parser.getAttributeValue(null, "units");
```

gets the `units` attribute of the current element.

By default, namespace processing is enabled. You can deactivate it by modifying the factory:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, false);
```

Listing 3.9 contains the code for the web crawler program implemented with the StAX parser. As you can see, the code is simpler than the equivalent SAX code because you don't have to worry about event handling.

Listing 3.9 stax/StAXTest.java

```
1 package stax;
2
3 import java.io.*;
4 import java.net.*;
5 import javax.xml.stream.*;
6
7 /**
8 * This program demonstrates how to use a StAX parser. The program prints all hyperlinks of
9 * an XHTML web page. <br>
10 * Usage: java stax.StAXTest URL
11 * @author Cay Horstmann
12 * @version 1.0 2007-06-23
13 */
14 public class StAXTest
15 {
16     public static void main(String[] args) throws Exception
17     {
18         String urlString;
19         if (args.length == 0)
20         {
21             urlString = "http://www.w3c.org";
22             System.out.println("Using " + urlString);
23         }
24         else urlString = args[0];
25         URL url = new URL(urlString);
26         InputStream in = url.openStream();
27         XMLInputFactory factory = XMLInputFactory.newInstance();
28         XMLStreamReader parser = factory.createXMLStreamReader(in);
29         while (parser.hasNext())
30         {
31             int event = parser.next();
```

```
32     if (event == XMLStreamConstants.START_ELEMENT)
33     {
34         if (parser.getLocalName().equals("a"))
35         {
36             String href = parser.getAttributeValue(null, "href");
37             if (href != null)
38                 System.out.println(href);
39         }
40     }
41 }
42 }
43 }
```

javax.xml.stream.XMLInputFactory 6

- static XMLInputFactory newInstance()
returns an instance of the XMLInputFactory class.
- void setProperty(String name, Object value)

sets a property for this factory, or throws an `IllegalArgumentException` if the property is not supported or cannot be set to the given value. The Java SE implementation supports the following Boolean-valued properties:

"javax.xml.stream.isValidating"

When `false` (the default), the document is not validated. Not required by the specification.

"javax.xml.stream.isNamespaceAware"

When `true` (the default), namespaces are processed. Not required by the specification.

"javax.xml.stream.isCoalescing"

When `false` (the default), adjacent character data are not coalesced.

"javax.xml.stream.isReplacingEntityReferences"

When `true` (the default), entity references are replaced and reported as character data.

"javax.xml.stream.isSupportingExternalEntities"

When `true` (the default), external entities are resolved. The specification gives no default for this property.

"javax.xml.stream.supportDTD"

When `true` (the default), DTDs are reported as events.

- XMLStreamReader createXMLStreamReader(InputStream in)
- XMLStreamReader createXMLStreamReader(InputStream in, String characterEncoding)
- XMLStreamReader createXMLStreamReader(Reader in)
- XMLStreamReader createXMLStreamReader(Source in)

creates a parser that reads from the given stream, reader, or JAXP source.

javax.xml.stream.XMLStreamReader 6

- `boolean hasNext()`
returns true if there is another parse event.
- `int next()`
sets the parser state to the next parse event and returns one of the following constants: START_ELEMENT, END_ELEMENT, CHARACTERS, START_DOCUMENT, END_DOCUMENT, CDATA, COMMENT, SPACE (ignorable whitespace), PROCESSING_INSTRUCTION, ENTITY_REFERENCE, DTD.
 - `boolean isStartElement()`
 - `boolean isEndElement()`
 - `boolean isCharacters()`
 - `boolean isWhiteSpace()`
returns true if the current event is a start element, end element, character data, or whitespace.
- `QName getName()`
- `String getLocalName()`
gets the name of the element in a START_ELEMENT or END_ELEMENT event.
- `String getText()`
returns the characters of a CHARACTERS, COMMENT, or CDATA event, the replacement value for an ENTITY_REFERENCE, or the internal subset of a DTD.
 - `int getAttributeCount()`
 - `QName getAttributeName(int index)`
 - `String getAttributeLocalName(int index)`
 - `String getAttributeValue(int index)`
gets the attribute count and the names and values of the attributes, provided the current event is START_ELEMENT.
- `String getAttributeValue(String namespaceURI, String name)`
gets the value of the attribute with the given name, provided the current event is START_ELEMENT. If namespaceURI is null, the namespace is not checked.

3.7 Generating XML Documents

You now know how to write Java programs that read XML. Let us now turn to the opposite process: producing XML output. Of course, you could write an XML file simply by making a sequence of `print` calls, printing the elements, attributes, and text content, but that would not be a good idea. The code is rather tedious, and you can easily make mistakes if you don't pay attention to special symbols (such as " or <) in the attribute values and text content.

A better approach is to build up a DOM tree with the contents of the document and then write out the tree contents. The following sections discuss the details.

3.7.1 Documents without Namespaces

To build a DOM tree, you start out with an empty document. You can get an empty document by calling the `newDocument` method of the `DocumentBuilder` class:

```
Document doc = builder.newDocument();
```

Use the `createElement` method of the `Document` class to construct the elements of your document:

```
Element rootElement = doc.createElement(rootName);
Element childElement = doc.createElement(childName);
```

Use the `createTextNode` method to construct text nodes:

```
Text textNode = doc.createTextNode(textContents);
```

Add the root element to the document, and add the child nodes to their parents:

```
doc.appendChild(rootElement);
rootElement.appendChild(childElement);
childElement.appendChild(textNode);
```

As you build up the DOM tree, you may also need to set element attributes. Simply call the `setAttribute` method of the `Element` class:

```
rootElement.setAttribute(name, value);
```

3.7.2 Documents with Namespaces

If you use namespaces, the procedure for creating a document is slightly different.

First, set the builder factory to be namespace-aware, then create the builder:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
builder = factory.newDocumentBuilder();
```

Then use `createElementNS` instead of `createElement` to create any nodes:

```
String namespace = "http://www.w3.org/2000/svg";
Element rootElement = doc.createElementNS(namespace, "svg");
```

If your node has a qualified name with a namespace prefix, then any necessary `xmlns`-prefixed attributes are created automatically. For example, if you need SVG inside XHTML, you can construct an element like this:

```
Element svgElement = doc.createElement(namespace, "svg:svg")
```

When the element is written, it turns into

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg">
```

If you need to set element attributes whose names are in a namespace, use the `setAttributeNS` method of the `Element` class:

```
rootElement.setAttributeNS(namespace, qualifiedName, value);
```

3.7.3 Writing Documents

Somewhat curiously, it is not so easy to write a DOM tree to an output stream. The easiest approach is to use the Extensible Stylesheet Language Transformations (XSLT) API. For more information about XSLT, turn to Section 3.8, “XSL Transformations,” on p. 222. Right now, consider the code that follows a magic incantation to produce XML output.

We apply the do-nothing transformation to the document and capture its output. To include a `DOCTYPE` node in the output, we also need to set the `SYSTEM` and `PUBLIC` identifiers as output properties.

```
// construct the do-nothing transformation
Transformer t = TransformerFactory.newInstance().newTransformer();
// set output properties to get a DOCTYPE node
t.setOutputProperty(OutputKeys.DOCUMENT_TYPE_NAME, "xml");
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty(OutputKeys.METHOD, "xml");
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
// apply the do-nothing transformation and send the output to a file
t.transform(new DOMSource(doc), new StreamResult(new FileOutputStream(file)));
```

Another approach is to use the `LSSerializer` interface. To get an instance, you have to use the following magic incantation:

```
DOMImplementation impl = doc.getImplementation();
DOMImplementationLS implLS = (DOMImplementationLS) impl.getFeature("LS", "3.0");
LSSerializer ser = implLS.createLSSerializer();
```

If you want spaces and line breaks, set this flag:

```
ser.getDomConfig().setParameter("format-pretty-print", true);
```

Then it's simple enough to convert a document to a string:

```
String str = ser.writeToString(doc);
```

If you want to write the output directly to a file, you need an `LSOutput`:

```
LSOutput out = implLS.createLSOutput();
out.setEncoding("UTF-8");
out.setByteStream(Files.newOutputStream(path));
ser.write(doc, out);
```

3.7.4 An Example: Generating an SVG File

Listing 3.10 on p. 215 is a typical program that produces XML output. The program draws a modernist painting—a random set of colored rectangles (see Figure 3.6). To save a masterpiece, we use the Scalable Vector Graphics (SVG) format. SVG is an XML format to describe complex graphics in a device-independent fashion. You can find more information about SVG at www.w3c.org/Graphics/SVG. To view SVG files, simply use any modern browser.

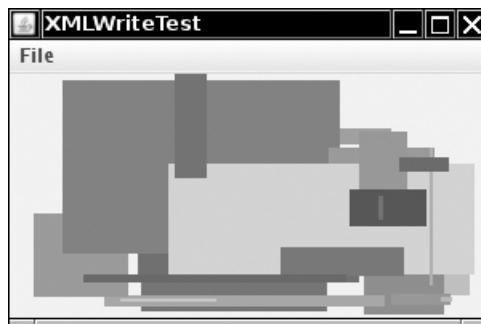


Figure 3.6 Generating modern art

We don't need to go into details about SVG; for our purposes, we just need to know how to express a set of colored rectangles. Here is a sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
  "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd">
<svg xmlns="http://www.w3.org/2000/svg" width="300" height="150">
  <rect x="231" y="61" width="9" height="12" fill="#6e4a13"/>
  <rect x="107" y="106" width="56" height="5" fill="#c406be"/>
  ...
</svg>
```

As you can see, each rectangle is described as a `rect` node. The position, width, height, and fill color are attributes. The fill color is an RGB value in hexadecimal.

NOTE: SVG uses attributes heavily. In fact, some attributes are quite complex. For example, here is a path element:

```
<path d="M 100 100 L 300 100 L 200 300 z">
```

The M denotes a “moveto” command, L is “lineto,” and z is “closepath” (!). Apparently, the designers of this data format didn’t have much confidence in using XML for structured data. In your own XML formats, you might want to use elements instead of complex attributes.

javax.xml.parsers.DocumentBuilder 1.4

- `Document newDocument()`
returns an empty document.

org.w3c.dom.Document 1.4

- Element createElement(String name)
 - Element createElementNS(String uri, String qname)
creates an element with the given name.
 - Text.createTextNode(String data)
creates a text node with the given data.

org.w3c.dom.Node 1.4

- `Node appendChild(Node child)`
appends a node to the list of children of this node. Returns the appended node.

org.w3c.dom.Element 1.4

- `void setAttribute(String name, String value)`
 - `void setAttributeNS(String uri, String qname, String value)`
sets the attribute with the given name to the given value

<i>Parameters:</i>	<code>uri</code>	The URI of the namespace, or <code>null</code>
	<code>qname</code>	The qualified name. If it has an alias prefix, then <code>uri</code> must not be <code>null</code> .
	<code>value</code>	The attribute value

javax.xml.transform.TransformerFactory 1.4

- static TransformerFactory newInstance()
returns an instance of the TransformerFactory class.
- Transformer newTransformer()
returns an instance of the Transformer class that carries out an identity (do-nothing) transformation.

javax.xml.transform.Transformer 1.4

- void setOutputProperty(String name, String value)
sets an output property. See www.w3.org/TR/xslt#output for a listing of the standard output properties. The most useful ones are shown here:

doctype-public	The public ID to be used in the DOCTYPE declaration
doctype-system	The system ID to be used in the DOCTYPE declaration
indent	"yes" or "no"
method	"xml", "html", "text", or a custom string
- void transform(Source from, Result to)
transforms an XML document.

javax.xml.transform.dom.DOMSource 1.4

- DOMSource(Node n)
constructs a source from the given node. Usually, n is a document node.

javax.xml.transform.stream.StreamResult 1.4

- StreamResult(File f)
 - StreamResult(OutputStream out)
 - StreamResult(Writer out)
 - StreamResult(String systemID)
- constructs a stream result from a file, stream, writer, or system ID (usually a relative or absolute URL).

3.7.5 Writing an XML Document with StAX

In the preceding section, you saw how to produce an XML document by writing a DOM tree. If you have no other use for the DOM tree, that approach is not very efficient.

The StAX API lets you write an XML tree directly. Construct an `XMLStreamWriter` from an `OutputStream`:

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(out);
```

To produce the XML header, call

```
writer.writeStartDocument()
```

Then call

```
writer.writeStartElement(name);
```

Add attributes by calling

```
writer.writeAttribute(name, value);
```

Now you can add child elements by calling `writeStartElement` again, or write characters with

```
writer.writeCharacters(text);
```

When you have written all child nodes, call

```
writer.writeEndElement();
```

This causes the current element to be closed.

To write an element without children (such as ``), use the call

```
writer.writeEmptyElement(name);
```

Finally, at the end of the document, call

```
writer.writeEndDocument();
```

This call closes any open elements.

You still need to close the `XMLStreamWriter`, and you need to do it manually since the `XMLStreamWriter` interface does not extend the `AutoCloseable` interface.

As with the DOM/XSLT approach, you don't have to worry about escaping characters in attribute values and character data. However, it is possible to produce malformed XML, such as a document with multiple root nodes. Also, the current version of StAX has no support for producing indented output.

The program in Listing 3.10 shows you both approaches for writing XML. Listings 3.11 and 3.12 show the frame and component classes for the rectangle painting.

Listing 3.10 write/XMLWriteTest.java

```
1 package write;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * This program shows how to write an XML file. It saves a file describing a modern drawing in SVG
8  * format.
9  * @version 1.12 2016-04-27
10 * @author Cay Horstmann
11 */
12 public class XMLWriteTest
13 {
14     public static void main(String[] args)
15     {
16         EventQueue.invokeLater(() ->
17         {
18             JFrame frame = new XMLWriteFrame();
19             frame.setTitle("XMLWriteTest");
20             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21             frame.setVisible(true);
22         });
23     }
24 }
```

Listing 3.11 write/XMLWriteFrame.java

```
1 package write;
2
3 import java.io.*;
4 import java.nio.file.*;
5
6 import javax.swing.*;
7 import javax.xml.stream.*;
8 import javax.xml.transform.*;
9 import javax.xml.transform.dom.*;
10 import javax.xml.transform.stream.*;
11
12 import org.w3c.dom.*;
```

(Continues)

Listing 3.11 (Continued)

```
14  /**
15  * A frame with a component for showing a modern drawing.
16  */
17 public class XMLWriteFrame extends JFrame
18 {
19     private RectangleComponent comp;
20     private JFileChooser chooser;
21
22     public XMLWriteFrame()
23     {
24         chooser = new JFileChooser();
25
26         // add component to frame
27
28         comp = new RectangleComponent();
29         add(comp);
30
31         // set up menu bar
32
33         JMenuBar menuBar = new JMenuBar();
34         setJMenuBar(menuBar);
35
36         JMenu menu = new JMenu("File");
37         menuBar.add(menu);
38
39         JMenuItem newItem = new JMenuItem("New");
40         menu.add(newItem);
41         newItem.addActionListener(event -> comp.newDrawing());
42
43         JMenuItem saveItem = new JMenuItem("Save with DOM/XSLT");
44         menu.add(saveItem);
45         saveItem.addActionListener(event -> saveDocument());
46
47         JMenuItem saveStAXItem = new JMenuItem("Save with StAX");
48         menu.add(saveStAXItem);
49         saveStAXItem.addActionListener(event -> saveStAX());
50
51         JMenuItem exitItem = new JMenuItem("Exit");
52         menu.add(exitItem);
53         exitItem.addActionListener(event -> System.exit(0));
54         pack();
55     }
56
57 /**
58 * Saves the drawing in SVG format, using DOM/XSLT.
59 */
```

```
60    public void saveDocument()
61    {
62        try
63        {
64            if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
65            File file = chooser.getSelectedFile();
66            Document doc = comp.buildDocument();
67            Transformer t = TransformerFactory.newInstance().newTransformer();
68            t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
69                "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd");
70            t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, "-//W3C//DTD SVG 20000802//EN");
71            t.setOutputProperty(OutputKeys.INDENT, "yes");
72            t.setOutputProperty(OutputKeys.METHOD, "xml");
73            t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
74            t.transform(new DOMSource(doc), new StreamResult(Files.newOutputStream(file.toPath())));
75        }
76        catch (TransformerException | IOException ex)
77        {
78            ex.printStackTrace();
79        }
80    }
81
82 /**
83 * Saves the drawing in SVG format, using StAX.
84 */
85 public void saveStAX()
86 {
87     if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
88     File file = chooser.getSelectedFile();
89     XMLOutputFactory factory = XMLOutputFactory.newInstance();
90     try
91     {
92         XMLStreamWriter writer = factory.createXMLStreamWriter(
93             Files.newOutputStream(file.toPath()));
94         try
95         {
96             comp.writeDocument(writer);
97         }
98         finally
99         {
100             writer.close(); // Not autocloseable
101         }
102     }
103     catch (XMLStreamException | IOException ex)
104     {
105         ex.printStackTrace();
106     }
107 }
108 }
```

Listing 3.12 write/RectangleComponent.java

```
1 package write;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.util.*;
6 import javax.swing.*;
7 import javax.xml.parsers.*;
8 import javax.xml.stream.*;
9 import org.w3c.dom.*;
10
11 /**
12 * A component that shows a set of colored rectangles.
13 */
14 public class RectangleComponent extends JComponent
15 {
16     private static final Dimension PREFERRED_SIZE = new Dimension(300, 200);
17
18     private java.util.List<Rectangle2D> rects;
19     private java.util.List<Color> colors;
20     private Random generator;
21     private DocumentBuilder builder;
22
23     public RectangleComponent()
24     {
25         rects = new ArrayList<>();
26         colors = new ArrayList<>();
27         generator = new Random();
28
29         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
30         factory.setNamespaceAware(true);
31         try
32         {
33             builder = factory.newDocumentBuilder();
34         }
35         catch (ParserConfigurationException e)
36         {
37             e.printStackTrace();
38         }
39     }
40
41 /**
42 * Create a new random drawing.
43 */
44 public void newDrawing()
45 {
```

```
46     int n = 10 + generator.nextInt(20);
47     rects.clear();
48     colors.clear();
49     for (int i = 1; i <= n; i++)
50     {
51         int x = generator.nextInt(getWidth());
52         int y = generator.nextInt(getHeight());
53         int width = generator.nextInt(getWidth() - x);
54         int height = generator.nextInt(getHeight() - y);
55         rects.add(new Rectangle(x, y, width, height));
56         int r = generator.nextInt(256);
57         int g = generator.nextInt(256);
58         int b = generator.nextInt(256);
59         colors.add(new Color(r, g, b));
60     }
61     repaint();
62 }
63
64 public void paintComponent(Graphics g)
65 {
66     if (rects.size() == 0) newDrawing();
67     Graphics2D g2 = (Graphics2D) g;
68
69     // draw all rectangles
70     for (int i = 0; i < rects.size(); i++)
71     {
72         g2.setPaint(colors.get(i));
73         g2.fill(rects.get(i));
74     }
75 }
76
77 /**
78 * Creates an SVG document of the current drawing.
79 * @return the DOM tree of the SVG document
80 */
81 public Document buildDocument()
82 {
83     String namespace = "http://www.w3.org/2000/svg";
84     Document doc = builder.newDocument();
85     Element svgElement = doc.createElementNS(namespace, "svg");
86     doc.appendChild(svgElement);
87     svgElement.setAttribute("width", "" + getWidth());
88     svgElement.setAttribute("height", "" + getHeight());
89     for (int i = 0; i < rects.size(); i++)
90     {
```

(Continues)

Listing 3.12 (Continued)

```
91     Color c = colors.get(i);
92     Rectangle2D r = rects.get(i);
93     Element rectElement = doc.createElementNS(namespace, "rect");
94     rectElement.setAttribute("x", "" + r.getX());
95     rectElement.setAttribute("y", "" + r.getY());
96     rectElement.setAttribute("width", "" + r.getWidth());
97     rectElement.setAttribute("height", "" + r.getHeight());
98     rectElement.setAttribute("fill", String.format("#%06x",
99                             c.getRGB() & 0xFFFFFF));
100    svgElement.appendChild(rectElement);
101 }
102 }
103 }
104 /**
105 * Writes an SVG document of the current drawing.
106 * @param writer the document destination
107 */
108 public void writeDocument(XMLStreamWriter writer) throws XMLStreamException
109 {
110     writer.writeStartDocument();
111     writer.writeDTD("<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN\" "
112                     + "\nhttp://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd\">");
113     writer.writeStartElement("svg");
114     writer.writeDefaultNamespace("http://www.w3.org/2000/svg");
115     writer.writeAttribute("width", "" + getWidth());
116     writer.writeAttribute("height", "" + getHeight());
117     for (int i = 0; i < rects.size(); i++)
118     {
119         Color c = colors.get(i);
120         Rectangle2D r = rects.get(i);
121         writer.writeEmptyElement("rect");
122         writer.writeAttribute("x", "" + r.getX());
123         writer.writeAttribute("y", "" + r.getY());
124         writer.writeAttribute("width", "" + r.getWidth());
125         writer.writeAttribute("height", "" + r.getHeight());
126         writer.writeAttribute("fill", String.format("#%06x",
127                             c.getRGB() & 0xFFFFFF));
128     }
129 }
130     writer.writeEndDocument(); // closes svg element
131 }
132
133     public Dimension getPreferredSize() { return PREFERRED_SIZE; }
134 }
```

javax.xml.stream.XMLOutputFactory 6

- static XMLOutputFactory newInstance()
returns an instance of the XMLOutputFactory class.
- XMLStreamWriter createXMLStreamWriter(OutputStream in)
- XMLStreamWriter createXMLStreamWriter(OutputStream in, String characterEncoding)
- XMLStreamWriter createXMLStreamWriter(Writer in)
- XMLStreamWriter createXMLStreamWriter(Result in)
creates a writer that writes to the given stream, writer, or JAXP result.

javax.xml.stream.XMLStreamWriter 6

- void writeStartDocument()
- void writeStartDocument(String xmlVersion)
- void writeStartDocument(String encoding, String xmlVersion)
writes the XML processing instruction at the top of the document. Note that the encoding parameter is only used to write the attribute. It does not set the character encoding of the output.
- void setDefaultNamespace(String namespaceURI)
- void setPrefix(String prefix, String namespaceURI)
sets the default namespace or the namespace associated with a prefix. The declaration is scoped to the current element or, if no element has been written, to the document root.
- void writeStartElement(String localName)
- void writeStartElement(String namespaceURI, String localName)
writes a start tag, replacing the namespaceURI with the associated prefix.
- void writeEndElement()
closes the current element.
- void writeEndDocument()
closes all open elements.
- void writeEmptyElement(String localName)
- void writeEmptyElement(String namespaceURI, String localName)
writes a self-closing tag, replacing the namespaceURI with the associated prefix.

(Continues)

javax.xml.stream.XMLStreamWriter 6 (Continued)

- `void writeAttribute(String localName, String value)`
- `void writeAttribute(String namespaceURI, String localName, String value)`
writes an attribute for the current element, replacing the `namespaceURI` with the associated prefix.
- `void writeCharacters(String text)`
writes character data.
- `void writeCData(String text)`
writes a CDATA block.
- `void writeDTD(String dtd)`
writes the `dtd` string, which is assumed to contain a DOCTYPE declaration.
- `void writeComment(String comment)`
writes a comment.
- `void close()`
closes this writer.

3.8 XSL Transformations

The XSL Transformations (XSLT) mechanism allows you to specify rules for transforming XML documents into other formats, such as plain text, XHTML, or any other XML format. XSLT is commonly used to translate from one machine-readable XML format to another, or to translate XML into a presentation format for human consumption.

You need to provide an XSLT stylesheet that describes the conversion of XML documents into some other format. An XSLT processor reads an XML document and the stylesheet and produces the desired output (see Figure 3.7).

The XSLT specification is quite complex, and entire books have been written on the subject. We can't possibly discuss all the features of XSLT, so we will just work through a representative example. You can find more information in the book *Essential XML* by Don Box et al. The XSLT specification is available at www.w3.org/TR/xslt.

Suppose we want to transform XML files with employee records into HTML documents. Consider this input file:

```
<staff>
  <employee>
    <name>Carl Cracker</name>
```

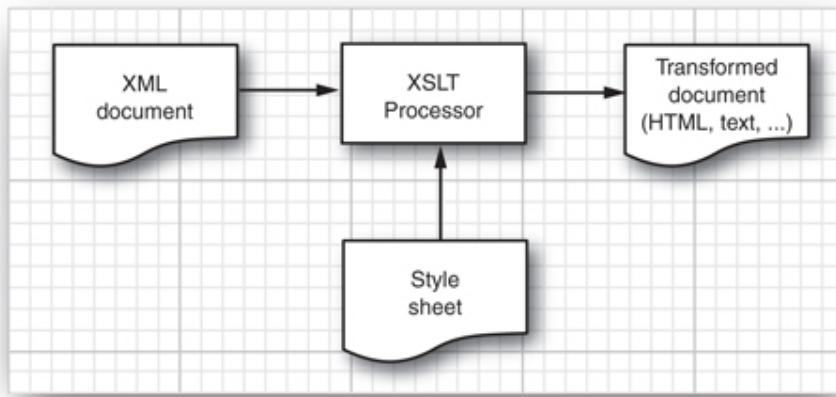


Figure 3.7 Applying XSL transformations

```
<salary>75000</salary>
<hiredate year="1987" month="12" day="15"/>
</employee>
<employee>
  <name>Harry Hacker</name>
  <salary>50000</salary>
  <hiredate year="1989" month="10" day="1"/>
</employee>
<employee>
  <name>Tony Tester</name>
  <salary>40000</salary>
  <hiredate year="1990" month="3" day="15"/>
</employee>
</staff>
```

The desired output is an HTML table:

```
<table border="1">
<tr>
<td>Carl Cracker</td><td>$75000.0</td><td>1987-12-15</td>
</tr>
<tr>
<td>Harry Hacker</td><td>$50000.0</td><td>1989-10-1</td>
</tr>
<tr>
<td>Tony Tester</td><td>$40000.0</td><td>1990-3-15</td>
</tr>
</table>
```

A stylesheet with transformation templates has this form:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="html"/>
    template1

    template2
    ...
</xsl:stylesheet>
```

In our example, the `xsl:output` element specifies the method as HTML. Other valid method settings are `xml` and `text`.

Here is a typical template:

```
<xsl:template match="/staff/employee">
    <tr><xsl:apply-templates/></tr>
</xsl:template>
```

The value of the `match` attribute is an XPath expression. The template states: Whenever you see a node in the XPath set `/staff/employee`, do the following:

1. Emit the string `<tr>`.
2. Keep applying templates as you process its children.
3. Emit the string `</tr>` after you are done with all children.

In other words, this template generates the HTML table row markers around every employee record.

The XSLT processor starts processing by examining the root element. Whenever a node matches one of the templates, it applies the template. (If multiple templates match, the best matching one is used; see the specification at www.w3.org/TR/xslt for the gory details.) If no template matches, the processor carries out a default action. For text nodes, the default is to include the contents in the output. For elements, the default action is to create no output but to keep processing the children.

Here is a template for transforming `name` nodes in an employee file:

```
<xsl:template match="/staff/employee/name">
    <td><xsl:apply-templates/></td>
</xsl:template>
```

As you can see, the template produces the `<td> . . . </td>` delimiters, and it asks the processor to recursively visit the children of the `name` element. There is just one child—the text node. When the processor visits that node, it emits the text contents (provided, of course, that there is no other matching template).

You have to work a little harder if you want to copy attribute values into the output. Here is an example:

```
<xsl:template match="/staff/employee/hiredate">
  <td><xsl:value-of select="@year"/>-<xsl:value-of
    select="@month"/>-<xsl:value-of select="@day"/></td>
</xsl:template>
```

When processing a `hiredate` node, this template emits

1. The string `<td>`
2. The value of the `year` attribute
3. A hyphen
4. The value of the `month` attribute
5. A hyphen
6. The value of the `day` attribute
7. The string `</td>`

The `xsl:value-of` statement computes the string value of a node set. The node set is specified by the XPath value of the `select` attribute. In this case, the path is relative to the currently processed node. The node set is converted to a string by concatenation of the string values of all nodes. The string value of an attribute node is its value. The string value of a text node is its contents. The string value of an element node is the concatenation of the string values of its child nodes (but not its attributes).

Listing 3.13 contains the stylesheet for turning an XML file with employee records into an HTML table.

Listing 3.14 shows a different set of transformations. The input is the same XML file, and the output is plain text in the familiar property file format:

```
employee.1.name=Carl Cracker
employee.1.salary=75000.0
employee.1.hiredate=1987-12-15
employee.2.name=Harry Hacker
employee.2.salary=50000.0
employee.2.hiredate=1989-10-1
employee.3.name=Tony Tester
employee.3.salary=40000.0
employee.3.hiredate=1990-3-15
```

That example uses the `position()` function which yields the position of the current node as seen from its parent. We thus get an entirely different output simply by switching the stylesheet. This means you can safely use XML to describe your data; if some applications need the data in another format, just use XSLT to generate the alternative format.

It is simple to generate XSL transformations in the Java platform. Set up a transformer factory for each stylesheet. Then, get a transformer object and tell it to transform a source to a result:

```
File styleSheet = new File(filename);
StreamSource styleSource = new StreamSource(styleSheet);
Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
t.transform(source, result);
```

The parameters of the `transform` method are objects of classes that implement the `Source` and `Result` interfaces. Several classes implement the `Source` interface:

```
DOMSource
SAXSource
StAXSource
StreamSource
```

You can construct a `StreamSource` from a file, stream, reader, or URL, and a `DOMSource` from the node of a DOM tree. For example, in the preceding section, we invoked the identity transformation as

```
t.transform(new DOMSource(doc), result);
```

In our example program, we do something slightly more interesting. Instead of starting out with an existing XML file, we produce a SAX XML reader that gives the illusion of parsing an XML file by emitting appropriate SAX events. Actually, our XML reader reads a flat file, as described in Chapter 2. The input file looks like this:

```
Carl Cracker|75000.0|1987|12|15
Harry Hacker|50000.0|1989|10|1
Tony Tester|40000.0|1990|3|15
```

Our XML reader generates SAX events as it processes the input. Here is a part of the `parse` method of the `EmployeeReader` class that implements the `XMLReader` interface:

```
AttributesImpl attributes = new AttributesImpl();
handler.startDocument();
handler.startElement("", "staff", "staff", attributes);
while ((line = in.readLine()) != null)
{
    handler.startElement("", "employee", "employee", attributes);
    StringTokenizer t = new StringTokenizer(line, "|");
    handler.startElement("", "name", "name", attributes);
    String s = t.nextToken();
    handler.characters(s.toCharArray(), 0, s.length());
    handler.endElement("", "name", "name");
    ...
    handler.endElement("", "employee", "employee");
}
```

```
handler.endElement("", rootElement, rootElement);
handler.endDocument();
```

The SAXSource for the transformer is constructed from the XML reader:

```
t.transform(new SAXSource(new EmployeeReader(),
    new InputSource(new FileInputStream(filename))), result);
```

This is an ingenious trick to convert non-XML legacy data into XML. Of course, most XSLT applications will already have XML input data, and you can simply invoke the `transform` method on a StreamSource:

```
t.transform(new StreamSource(file), result);
```

The transformation result is an object of a class that implements the Result interface. The Java library supplies three classes:

```
DOMResult
SAXResult
StreamResult
```

To store the result in a DOM tree, use a DocumentBuilder to generate a new document node and wrap it into a DOMResult:

```
Document doc = builder.newDocument();
t.transform(source, new DOMResult(doc));
```

To save the output in a file, use a StreamResult:

```
t.transform(source, new StreamResult(file));
```

Listing 3.15 contains the complete source code.

Listing 3.13 transform/makehtml.xsl

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2
3 <xsl:stylesheet
4   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5   version="1.0">
6
7   <xsl:output method="html"/>
8
9   <xsl:template match="/staff">
10    <table border="1"><xsl:apply-templates/></table>
11  </xsl:template>
12
13  <xsl:template match="/staff/employee">
14    <tr><xsl:apply-templates/></tr>
15  </xsl:template>
```

(Continues)

Listing 3.13 (*Continued*)

```
16 <xsl:template match="/staff/employee/name">
17     <td><xsl:apply-templates/></td>
18 </xsl:template>
19
20 <xsl:template match="/staff/employee/salary">
21     <td>$<xsl:apply-templates/></td>
22 </xsl:template>
23
24 <xsl:template match="/staff/employee/hiredate">
25     <td><xsl:value-of select="@year"/>-<xsl:value-of
26         select="@month"/>-<xsl:value-of select="@day"/></td>
27 </xsl:template>
28
29 </xsl:stylesheet>
```

Listing 3.14 transform/makeprop.xs1

```
1 <?xml version="1.0"?>
2
3 <xsl:stylesheet
4     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5     version="1.0">
6
7     <xsl:output method="text" omit-xml-declaration="yes"/>
8
9     <xsl:template match="/staff/employee">
10    employee.<xsl:value-of select="position()" />.<xsl:value-of
11        select="name/text()"/>
12    employee.<xsl:value-of select="position()" />.<xsl:value-of
13        select="salary/text()"/>
14    employee.<xsl:value-of select="position()" />.<xsl:value-of
15        select="hiredate/@year"/> -<xsl:value-of
16        select="hiredate/@month"/> -<xsl:value-of
17        select="hiredate/@day"/>
18    </xsl:template>
19
20 </xsl:stylesheet>
```

Listing 3.15 transform/TransformTest.java

```
1 package transform;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.util.*;
```

```
6 import javax.xml.transform.*;
7 import javax.xml.transform.sax.*;
8 import javax.xml.transform.stream.*;
9 import org.xml.sax.*;
10 import org.xml.sax.helpers.*;
11
12 /**
13 * This program demonstrates XSL transformations. It applies a transformation to a set of employee
14 * records. The records are stored in the file employee.dat and turned into XML format. Specify
15 * the stylesheet on the command line, e.g.
16 *   java transform.TransformTest transform/makeprop.xsl
17 * @version 1.03 2016-04-27
18 * @author Cay Horstmann
19 */
20 public class TransformTest
21 {
22     public static void main(String[] args) throws Exception
23     {
24         Path path;
25         if (args.length > 0) path = Paths.get(args[0]);
26         else path = Paths.get("transform", "makehtml.xsl");
27         try (InputStream styleIn = Files.newInputStream(path))
28         {
29             StreamSource styleSource = new StreamSource(styleIn);
30
31             Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
32             t.setOutputProperty(OutputKeys.INDENT, "yes");
33             t.setOutputProperty(OutputKeys.METHOD, "xml");
34             t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
35
36             try (InputStream docIn = Files.newInputStream(Paths.get("transform", "employee.dat")))
37             {
38                 t.transform(new SAXSource(new EmployeeReader(), new InputSource(docIn)),
39                             new StreamResult(System.out));
40             }
41         }
42     }
43 }
44
45 /**
46 * This class reads the flat file employee.dat and reports SAX parser events to act as if it was
47 * parsing an XML file.
48 */
49 class EmployeeReader implements XMLReader
50 {
51     private ContentHandler handler;
52
53     public void parse(InputSource source) throws IOException, SAXException
54     {
```

(Continues)

Listing 3.15 (Continued)

```
55     InputStream stream = source.getByteStream();
56     BufferedReader in = new BufferedReader(new InputStreamReader(stream));
57     String rootElement = "staff";
58     AttributesImpl attrs = new AttributesImpl();
59
60     if (handler == null) throw new SAXException("No content handler");
61
62     handler.startDocument();
63     handler.startElement("", rootElement, rootElement, attrs);
64     String line;
65     while ((line = in.readLine()) != null)
66     {
67         handler.startElement("", "employee", "employee", attrs);
68         StringTokenizer t = new StringTokenizer(line, "|");
69
70         handler.startElement("", "name", "name", attrs);
71         String s = t.nextToken();
72         handler.characters(s.toCharArray(), 0, s.length());
73         handler.endElement("", "name", "name");
74
75         handler.startElement("", "salary", "salary", attrs);
76         s = t.nextToken();
77         handler.characters(s.toCharArray(), 0, s.length());
78         handler.endElement("", "salary", "salary");
79
80         attrs.addAttribute("", "year", "year", "CDATA", t.nextToken());
81         attrs.addAttribute("", "month", "month", "CDATA", t.nextToken());
82         attrs.addAttribute("", "day", "day", "CDATA", t.nextToken());
83         handler.startElement("", "hiredate", "hiredate", attrs);
84         handler.endElement("", "hiredate", "hiredate");
85         attrs.clear();
86
87         handler.endElement("", "employee", "employee");
88     }
89
90     handler.endElement("", rootElement, rootElement);
91     handler.endDocument();
92 }
93
94 public void setContentHandler(ContentHandler newValue)
95 {
96     handler = newValue;
97 }
98 }
```

```
99     public ContentHandler getContentHandler()
100    {
101        return handler;
102    }
103
104    // the following methods are just do-nothing implementations
105    public void parse(String systemId) throws IOException, SAXException {}
106    public void setErrorHandler(ErrorHandler handler) {}
107    public ErrorHandler getErrorHandler() { return null; }
108    public void setDTDHandler(DTDHandler handler) {}
109    public DTDHandler getDTDHandler() { return null; }
110    public void setEntityResolver(EntityResolver resolver) {}
111    public EntityResolver getEntityResolver() { return null; }
112    public void setProperty(String name, Object value) {}
113    public Object getProperty(String name) { return null; }
114    public void setFeature(String name, boolean value) {}
115    public boolean getFeature(String name) { return false; }
116 }
```

javax.xml.transform.TransformerFactory 1.4

- `Transformer newTransformer(Source styleSheet)`
returns an instance of the `Transformer` class that reads a stylesheet from the given source.

javax.xml.transform.stream.StreamSource 1.4

- `StreamSource(File f)`
- `StreamSource(InputStream in)`
- `StreamSource(Reader in)`
- `StreamSource(String systemID)`

constructs a stream source from a file, stream, reader, or system ID (usually a relative or absolute URL).

javax.xml.transform.sax.SAXSource 1.4

- `SAXSource(XMLReader reader, InputSource source)`

constructs a SAX source that obtains data from the given input source and uses the given reader to parse the input.

`org.xml.sax.XMLReader 1.4`

- `void setContentHandler(ContentHandler handler)`
sets the handler that is notified of parse events as the input is parsed.
- `void parse(InputSource source)`
parses the input from the given input source and sends parse events to the content handler.

`javax.xml.transform.dom.DOMResult 1.4`

- `DOMResult(Node n)`
constructs a source from the given node. Usually, `n` is a new document node.

`org.xml.sax.helpers.AttributesImpl 1.4`

- `void addAttribute(String uri, String lname, String qname, String type, String value)`
adds an attribute to this attribute collection.

Parameters:

<code>uri</code>	The URI of the namespace
<code>lname</code>	The local name without prefix
<code>qname</code>	The qualified name with prefix
<code>type</code>	The type, one of "CDATA", "ID", "IDREF", "IDREFS", "NMTOKEN", "NMTOKENS", "ENTITY", "ENTITIES", or "NOTATION"
<code>value</code>	The attribute value

- `void clear()`
removes all attributes from this attribute collection.

This example concludes our discussion of XML support in the Java library. You should now have a good perspective on the major strengths of XML—in particular, for automated parsing and validation and as a powerful transformation mechanism. Of course, all this technology is only going to work for you if you design your XML formats well. You need to make sure that the formats are rich enough to express all your business needs, that they are stable over time, and that your business partners are willing to accept your XML documents. Those issues can be far more challenging than dealing with parsers, DTDs, or transformations.

In the next chapter, we will discuss network programming on the Java platform, starting with the basics of network sockets and moving on to higher-level protocols for e-mail and the World Wide Web.

Networking

In this chapter

- 4.1 Connecting to a Server, page 233
- 4.2 Implementing Servers, page 241
- 4.3 Interruptible Sockets, page 250
- 4.4 Getting Web Data, page 257
- 4.5 Sending E-Mail, page 277

We begin this chapter by reviewing basic networking concepts, then move on to writing Java programs that connect to network services. We will show you how network clients and servers are implemented. Finally, you will see how to send e-mail from a Java program and how to harvest information from a web server.

4.1 Connecting to a Server

In the following sections, you will connect to a server, first by hand and with telnet, and then with a Java program.

4.1.1 Using Telnet

The telnet program is a great debugging tool for network programming. You should be able to launch it by typing `telnet` from a command shell.

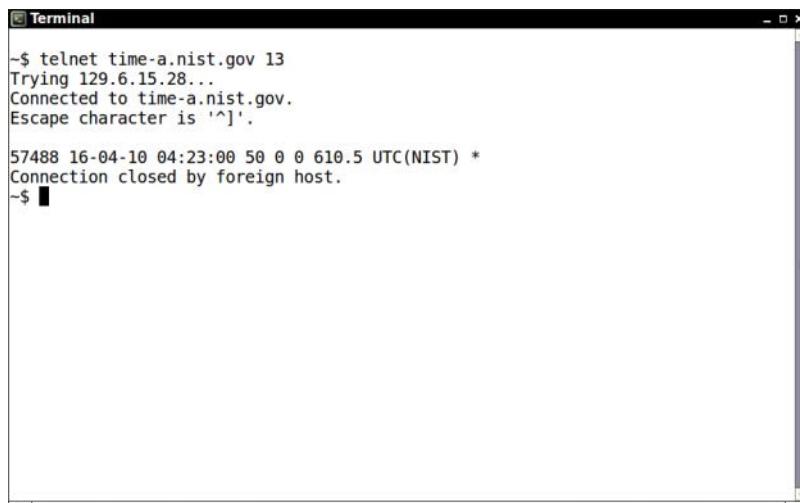
NOTE: In Windows, you need to activate telnet. Go to the Control Panel, select Programs, click Turn Windows Features On or Off, and select the Telnet client checkbox. The Windows firewall also blocks quite a few network ports that we use in this chapter; you might need an administrator account to unblock them.

You may have used telnet to connect to a remote computer, but you can use it to communicate with other services provided by Internet hosts as well. Here is an example of what you can do. Type

```
telnet time-anist.gov 13
```

As Figure 4.1 shows, you should get back a line like this:

```
57488 16-04-10 04:23:00 50 0 0 610.5 UTC(NIST) *
```



The screenshot shows a terminal window titled "Terminal". The window contains the following text:

```
-$ telnet time-anist.gov 13
Trying 129.6.15.28...
Connected to time-anist.gov.
Escape character is '^]'.

57488 16-04-10 04:23:00 50 0 0 610.5 UTC(NIST) *
Connection closed by foreign host.
-$
```

Figure 4.1 Output of the “time of day” service

What is going on? You have connected to the “time of day” service that most UNIX machines constantly run. The particular server that you connected to is operated by the National Institute of Standards and Technology and gives the measurement of a Cesium atomic clock. (Of course, the reported time is not completely accurate due to network delays.)

By convention, the “time of day” service is always attached to “port” number 13.

NOTE: In network parlance, a port is not a physical device, but an abstraction facilitating communication between a server and a client (see Figure 4.2).

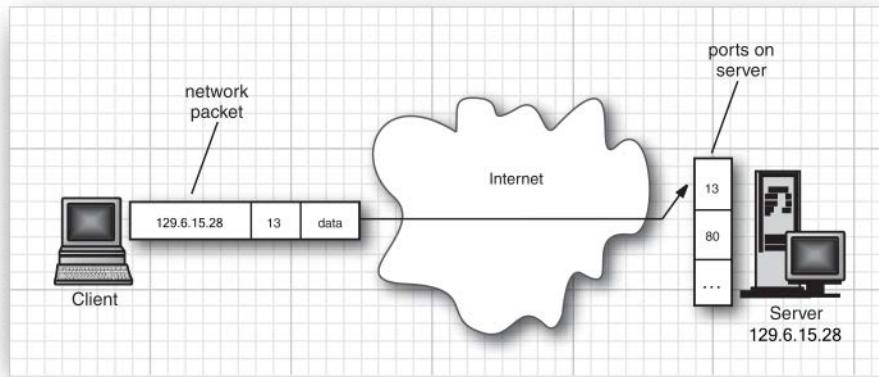


Figure 4.2 A client connecting to a server port

The server software is continuously running on the remote machine, waiting for any network traffic that wants to chat with port 13. When the operating system on the remote computer receives a network package that contains a request to connect to port number 13, it wakes up the listening server process and establishes the connection. The connection stays up until it is terminated by one of the parties.

When you began the telnet session with `time-a.nist.gov` at port 13, a piece of network software knew enough to convert the string "`time-a.nist.gov`" to its correct Internet Protocol (IP) address, 129.6.15.28. The telnet software then sent a connection request to that address, asking for a connection to port 13. Once the connection was established, the remote program sent back a line of data and closed the connection. In general, of course, clients and servers engage in a more extensive dialog before one or the other closes the connection.

Here is another experiment along the same lines—but a bit more interesting. Type

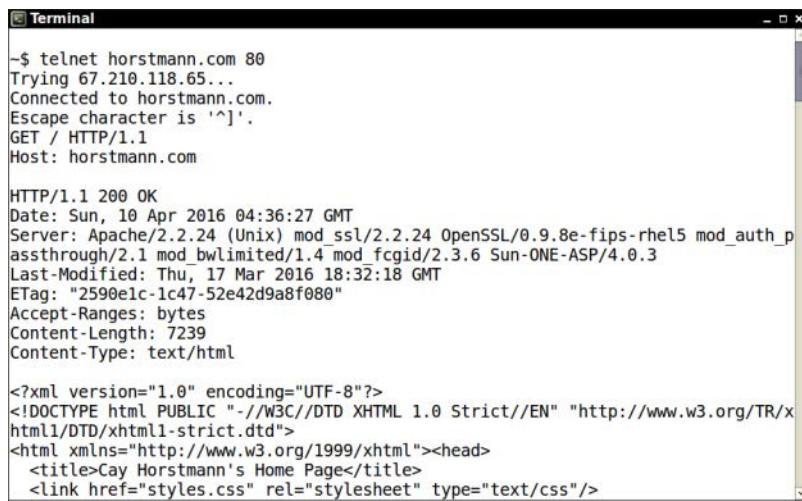
```
telnet horstmann.com 80
```

Then type very carefully the following:

```
GET / HTTP/1.1  
Host: horstmann.com  
blank line
```

That is, hit the Enter key twice at the end.

Figure 4.3 shows the response. It should look eerily familiar—you got a page of HTML-formatted text, namely Cay Horstmann’s home page.



```
Terminal
$ telnet horstmann.com 80
Trying 67.210.118.65...
Connected to horstmann.com.
Escape character is '^].
GET / HTTP/1.1
Host: horstmann.com

HTTP/1.1 200 OK
Date: Sun, 10 Apr 2016 04:36:27 GMT
Server: Apache/2.2.24 (Unix) mod_ssl/2.2.24 OpenSSL/0.9.8e-fips-rhel5 mod_auth_p
assthrough/2.1 mod_bwlimited/1.4 mod_fcgid/2.3.6 Sun-ONE-ASP/4.0.3
Last-Modified: Thu, 17 Mar 2016 18:32:18 GMT
ETag: "2590e1c-1c47-52e42d9a8f080"
Accept-Ranges: bytes
Content-Length: 7239
Content-Type: text/html

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/x
html1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<title>Cay Horstmann's Home Page</title>
<link href="styles.css" rel="stylesheet" type="text/css"/>
```

Figure 4.3 Using telnet to access an HTTP port

This is exactly the same process that your web browser goes through to get a web page. It uses HTTP to request web pages from servers. Of course, the browser displays the HTML code more nicely.

NOTE: The `Host` key/value pair is required when you connect to a web server that hosts multiple domains with the same IP address. You can omit it if the server hosts a single domain.

4.1.2 Connecting to a Server with Java

Our first network program in Listing 4.1 will do the same thing we did using telnet—connect to a port and print out what it finds.

Listing 4.1 socket/SocketTest.java

```
1 package socket;
2
3 import java.io.*;
4 import java.net.*;
5 import java.util.*;
6
```

```
7 /**
8 * This program makes a socket connection to the atomic clock in Boulder, Colorado, and prints
9 * the time that the server sends.
10 *
11 * @version 1.21 2016-04-27
12 * @author Cay Horstmann
13 */
14 public class SocketTest
15 {
16     public static void main(String[] args) throws IOException
17     {
18         try (Socket s = new Socket("time-a.nist.gov", 13);
19              Scanner in = new Scanner(s.getInputStream(), "UTF-8"))
20         {
21             while (in.hasNextLine())
22             {
23                 String line = in.nextLine();
24                 System.out.println(line);
25             }
26         }
27     }
28 }
```

The key statements of this simple program are these:

```
Socket s = new Socket("time-a.nist.gov", 13);
InputStream inStream = s.getInputStream();
```

The first line opens a *socket*, which is a network software abstraction that enables communication out of and into this program. We pass the remote address and the port number to the socket constructor. If the connection fails, an `UnknownHostException` is thrown. If there is another problem, an `IOException` occurs. Since `UnknownHostException` is a subclass of `IOException` and this is a sample program, we just catch the superclass.

Once the socket is open, the `getInputStream` method in `java.net.Socket` returns an `InputStream` object that you can use just like any other stream. Once you have grabbed the stream, this program simply prints each input line to standard output. This process continues until the stream is finished and the server disconnects.

This program works only with very simple servers, such as a “time of day” service. In more complex networking programs, the client sends request data to the server, and the server might not immediately disconnect at the end of a response. You will see how to implement that behavior in several examples throughout this chapter.

The `Socket` class is pleasant and easy to use because the Java library hides the complexities of establishing a networking connection and sending data across it.

The `java.net` package essentially gives you the same programming interface you would use to work with a file.

NOTE: In this book, we cover only the Transmission Control Protocol (TCP). The Java platform also supports the User Datagram Protocol (UDP), which can be used to send packets (also called *datagrams*) with much less overhead than TCP. The drawback is that packets need not be delivered in sequential order to the receiving application and can even be dropped altogether. It is up to the recipient to put the packets in order and to request retransmission of missing packets. UDP is well suited for applications in which missing packets can be tolerated—for example, for audio or video streams or continuous measurements.

java.net.Socket 1.0

- `Socket(String host, int port)`
constructs a socket to connect to the given host and port.
- `InputStream getInputStream()`
- `OutputStream getOutputStream()`
gets the stream to read data from the socket or write data to the socket.

4.1.3 Socket Timeouts

Reading from a socket blocks until data are available. If the host is unreachable, your application waits for a long time and you are at the mercy of the underlying operating system to eventually time out.

You can decide what timeout value is reasonable for your particular application. Then, call the `setSoTimeout` method to set a timeout value (in milliseconds).

```
Socket s = new Socket(. . .);
s.setSoTimeout(10000); // time out after 10 seconds
```

If the timeout value has been set for a socket, all subsequent read and write operations throw a `SocketTimeoutException` when the timeout has been reached before the operation has completed its work. You can catch that exception and react to the timeout.

```
try
{
    InputStream in = s.getInputStream(); // read from in
    . .
}
```

```
        catch (InterruptedException exception)
        {
            react to timeout
        }
```

There is one additional timeout issue that you need to address. The constructor

```
Socket(String host, int port)
```

can block indefinitely until an initial connection to the host is established.

You can overcome this problem by first constructing an unconnected socket and then connecting it with a timeout:

```
Socket s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

See Section 4.3, “Interruptible Sockets,” on p. 250 for how to allow users to interrupt the socket connection at any time.

java.net.Socket 1.0

- `Socket()` **1.1**
creates a socket that has not yet been connected.
- `void connect(SocketAddress address)` **1.4**
connects this socket to the given address.
- `void connect(SocketAddress address, int timeoutInMilliseconds)` **1.4**
connects this socket to the given address, or returns if the time interval expired.
- `void setSoTimeout(int timeoutInMilliseconds)` **1.1**
sets the blocking time for read requests on this socket. If the timeout is reached, an `InterruptedException` is raised.
- `boolean isConnected()` **1.4**
returns true if the socket is connected.
- `boolean isClosed()` **1.4**
returns true if the socket is closed.

4.1.4 Internet Addresses

Usually, you don’t have to worry too much about Internet addresses—the numerical host addresses that consist of 4 bytes (or, with IPv6, 16 bytes) such as 129.6.15.28. However, you can use the `InetAddress` class if you need to convert between host names and Internet addresses.

The `java.net` package supports IPv6 Internet addresses, provided the host operating system does.

The static `getByName` method returns an `InetAddress` object of a host. For example,

```
InetAddress address = InetAddress.getByName("time-a.nist.gov");
```

returns an `InetAddress` object that encapsulates the sequence of four bytes 129.6.15.28. You can access the bytes with the `getAddress` method.

```
byte[] addressBytes = address.getAddress();
```

Some host names with a lot of traffic correspond to multiple Internet addresses, to facilitate load balancing. For example, at the time of this writing, the host name `google.com` corresponds to twelve different Internet addresses. One of them is picked at random when the host is accessed. You can get all hosts with the `getAllByName` method.

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

Finally, you sometimes need the address of the local host. If you simply ask for the address of `localhost`, you always get the local loopback address 127.0.0.1, which cannot be used by others to connect to your computer. Instead, use the static `getLocalHost` method to get the address of your local host.

```
InetAddress address = InetAddress.getLocalHost();
```

Listing 4.2 is a simple program that prints the Internet address of your local host if you do not specify any command-line parameters, or all Internet addresses of another host if you specify the host name on the command line, such as

```
java inetAddress/InetAddressTest www.horstmann.com
```

Listing 4.2 `inetAddress/InetAddressTest.java`

```
1 package inetAddress;
2
3 import java.io.*;
4 import java.net.*;
5
6 /**
7 * This program demonstrates the InetAddress class. Supply a host name as command-line argument,
8 * or run without command-line arguments to see the address of the local host.
9 * @version 1.02 2012-06-05
10 * @author Cay Horstmann
11 */
12 public class InetAddressTest
13 {
```

```
14  public static void main(String[] args) throws IOException
15  {
16      if (args.length > 0)
17      {
18          String host = args[0];
19          InetAddress[] addresses = InetAddress.getAllByName(host);
20          for (InetAddress a : addresses)
21              System.out.println(a);
22      }
23      else
24      {
25          InetAddress localHostAddress = InetAddress.getLocalHost();
26          System.out.println(localHostAddress);
27      }
28  }
```

java.net.InetAddress 1.0

- static InetAddress getByName(String host)
- static InetAddress[] getAllByName(String host)
constructs an InetAddress, or an array of all Internet addresses, for the given host name.
- static InetAddress getLocalHost()
constructs an InetAddress for the local host.
- byte[] getAddress()
returns an array of bytes that contains the numerical address.
- String getHostAddress()
returns a string with decimal numbers, separated by periods, for example "129.6.15.28".
- String getHostName()
returns the host name.

4.2 Implementing Servers

Now that we have implemented a basic network client that receives data from the Internet, let's program a simple server that can send information to clients. In the previous section, we have implemented a basic network client that receives data from the Internet. In the following sections, we will program a simple server that can send information to clients.

4.2.1 Server Sockets

A server program, when started, waits for a client to attach to its port. For our example program, we chose port number 8189, which is not used by any of the standard services. The `ServerSocket` class establishes a socket. In our case, the command

```
ServerSocket s = new ServerSocket(8189);
```

establishes a server that monitors port 8189. The command

```
Socket incoming = s.accept();
```

tells the program to wait indefinitely until a client connects to that port. Once someone connects to this port by sending the correct request over the network, this method returns a `Socket` object that represents the connection that was made. You can use this object to get input and output streams, as is shown in the following code:

```
InputStream inStream = incoming.getInputStream();
OutputStream outStream = incoming.getOutputStream();
```

Everything that the server sends to the server output stream becomes the input of the client program, and all the output from the client program ends up in the server input stream.

In all the examples in this chapter, we transmit text through sockets. We therefore turn the streams into scanners and writers.

```
Scanner in = new Scanner(inStream, "UTF-8");
PrintWriter out = new PrintWriter(new OutputStreamWriter(outStream, "UTF-8"),
    true /* autoFlush */);
```

Let's send the client a greeting:

```
out.println("Hello! Enter BYE to exit.");
```

When you use telnet to connect to this server program at port 8189, you will see this greeting on the terminal screen.

In this simple server, we just read the client's input, a line at a time, and echo it. This demonstrates that the program receives the input. An actual server would obviously compute and return an answer depending on the input.

```
String line = in.nextLine();
out.println("Echo: " + line);
if (!line.trim().equals("BYE")) done = true;
```

In the end, we close the incoming socket.

```
incoming.close();
```

That is all there is to it. Every server program, such as an HTTP web server, continues performing this loop:

1. It receives a command from the client (“get me this information”) through an incoming data stream.
2. It decodes the client command.
3. It gathers the information that the client requested.
4. It sends the information to the client through the outgoing data stream.

Listing 4.3 is the complete program.

Listing 4.3 server/EchoServer.java

```
1 package server;
2
3 import java.io.*;
4 import java.net.*;
5 import java.util.*;
6
7 /**
8  * This program implements a simple server that listens to port 8189 and echoes back all client
9  * input.
10 * @version 1.21 2012-05-19
11 * @author Cay Horstmann
12 */
13 public class EchoServer
14 {
15     public static void main(String[] args) throws IOException
16     {
17         // establish server socket
18         try (ServerSocket s = new ServerSocket(8189))
19         {
20             // wait for client connection
21             try (Socket incoming = s.accept())
22             {
23                 InputStream inStream = incoming.getInputStream();
24                 OutputStream outStream = incoming.getOutputStream();
25
26                 try (Scanner in = new Scanner(inStream, "UTF-8"))
27                 {
28                     PrintWriter out = new PrintWriter(
29                         new OutputStreamWriter(outStream, "UTF-8"),
30                         true /* autoFlush */);
31
32                     out.println("Hello! Enter BYE to exit.");
33
34                     // echo client input
```

(Continues)

Listing 4.3 (Continued)

```
35     boolean done = false;
36     while (!done && in.hasNextLine())
37     {
38         String line = in.nextLine();
39         out.println("Echo: " + line);
40         if (line.trim().equals("BYE")) done = true;
41     }
42 }
43 }
44 }
45 }
46 }
```

To try it out, compile and run the program. Then use telnet to connect to the server localhost (or IP address 127.0.0.1) and port 8189.

If you are connected directly to the Internet, anyone in the world can access your echo server, provided they know your IP address and the magic port number.

When you connect to the port, you will see the message shown in Figure 4.4:

Hello! Enter BYE to exit.



Figure 4.4 Accessing an echo server

Type anything and watch the input echo on your screen. Type `BYE` (all uppercase letters) to disconnect. The server program will terminate as well.

java.net.ServerSocket 1.0

- `ServerSocket(int port)`
creates a server socket that monitors a port.
- `Socket accept()`
waits for a connection. This method blocks (i.e., idles) the current thread until the connection is made. The method returns a `Socket` object through which the program can communicate with the connecting client.
- `void close()`
closes the server socket.

4.2.2 Serving Multiple Clients

There is one problem with the simple server in the preceding example. Suppose we want to allow multiple clients to connect to our server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet might want to use it at the same time. Rejecting multiple connections allows any one client to monopolize the service by connecting to it for a long time. We can do much better through the magic of threads.

Every time we know the program has established a new socket connection—that is, every time the call to `accept()` returns a socket—we will launch a new thread to take care of the connection between the server and *that* client. The main program will just go back and wait for the next connection. For this to happen, the main loop of the server should look like this:

```
while (true)
{
    Socket incoming = s.accept();
    Runnable r = new ThreadedEchoHandler(incoming);

    Thread t = new Thread(r);
    t.start();
}
```

The `ThreadedEchoHandler` class implements `Runnable` and contains the communication loop with the client in its `run` method.

```
class ThreadedEchoHandler implements Runnable
{
    ...
}
```

```
public void run()
{
    try (InputStream inStream = incoming.getInputStream();
         OutputStream outStream = incoming.getOutputStream())
    {
        Process input and send response
    }
    catch(IOException e)
    {
        Handle exception
    }
}
```

When each connection starts a new thread, multiple clients can connect to the server at the same time. You can easily check this out.

1. Compile and run the server program (Listing 4.4).
2. Open several telnet windows as we have in Figure 4.5.

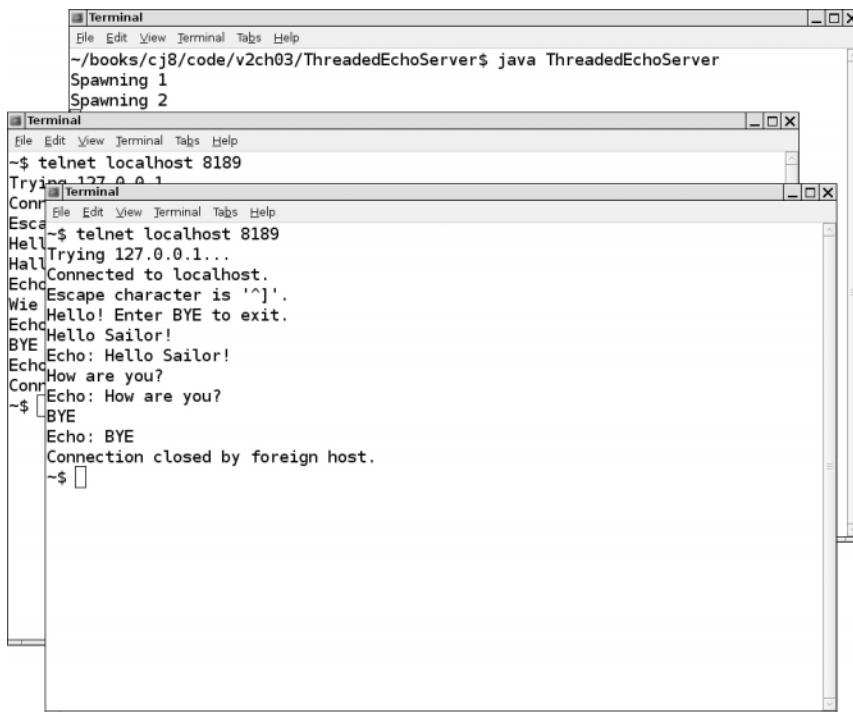


Figure 4.5 Several telnet windows communicating simultaneously

3. Switch between windows and type commands. Note that you can communicate through all of them simultaneously.
4. When you are done, switch to the window from which you launched the server program and press Ctrl+C to kill it.

NOTE: In this program, we spawn a separate thread for each connection. This approach is not satisfactory for high-performance servers. You can achieve greater server throughput by using features of the `java.nio` package. See www.ibm.com/developerworks/java/library/j-javaio for more information.

Listing 4.4 threaded/ThreadedEchoServer.java

```
1 package threaded;
2
3 import java.io.*;
4 import java.net.*;
5 import java.util.*;
6
7 /**
8 * This program implements a multithreaded server that listens to port 8189 and echoes back
9 * all client input.
10 * @author Cay Horstmann
11 * @version 1.22 2016-04-27
12 */
13 public class ThreadedEchoServer
14 {
15     public static void main(String[] args )
16     {
17         try (ServerSocket s = new ServerSocket(8189))
18         {
19             int i = 1;
20
21             while (true)
22             {
23                 Socket incoming = s.accept();
24                 System.out.println("Spawning " + i);
25                 Runnable r = new ThreadedEchoHandler(incoming);
26                 Thread t = new Thread(r);
27                 t.start();
28                 i++;
29             }
30         }
31         catch (IOException e)
32         {
33             e.printStackTrace();
34         }
35     }
36 }
```

(Continues)

Listing 4.4 (Continued)

```
34     }
35 }
36 }
37
38 /**
39 * This class handles the client input for one server socket connection.
40 */
41 class ThreadedEchoHandler implements Runnable
42 {
43     private Socket incoming;
44
45     /**
46      Constructs a handler.
47      @param incomingSocket the incoming socket
48     */
49     public ThreadedEchoHandler(Socket incomingSocket)
50     {
51         incoming = incomingSocket;
52     }
53
54     public void run()
55     {
56         try (InputStream inStream = incoming.getInputStream();
57              OutputStream outStream = incoming.getOutputStream())
58         {
59             Scanner in = new Scanner(inStream, "UTF-8");
60             PrintWriter out = new PrintWriter(
61                 new OutputStreamWriter(outStream, "UTF-8"),
62                 true /* autoFlush */);
63
64             out.println("Hello! Enter BYE to exit.");
65
66             // echo client input
67             boolean done = false;
68             while (!done && in.hasNextLine())
69             {
70                 String line = in.nextLine();
71                 out.println("Echo: " + line);
72                 if (!line.trim().equals("BYE"))
73                     done = true;
74             }
75         }
76         catch (IOException e)
77         {
78             e.printStackTrace();
79         }
80     }
81 }
```

4.2.3 Half-Close

The *half-close* allows one end of a socket connection to terminate its output while still receiving data from the other end.

Here is a typical situation. Suppose you transmit data to the server but you don't know at the outset how much data you have. With a file, you'd just close the file at the end of the data. However, if you close a socket, you immediately disconnect from the server and cannot read the response.

The half-close overcomes this problem. You can close the output stream of a socket, thereby indicating to the server the end of the requested data, but keep the input stream open.

The client side looks like this:

```
try (Socket socket = new Socket(host, port))
{
    Scanner in = new Scanner(socket.getInputStream(), "UTF-8");
    PrintWriter writer = new PrintWriter(socket.getOutputStream());
    // send request data
    writer.print(. . .);
    writer.flush();
    socket.shutdownOutput();
    // now socket is half-closed
    // read response data
    while (in.hasNextLine() != null) { String line = in.nextLine(); . . . }
}
```

The server side simply reads input until the end of the input stream is reached. Then it sends the response.

Of course, this protocol is only useful for one-shot services such as HTTP where the client connects, issues a request, catches the response, and then disconnects.

java.net.Socket 1.0

- void shutdownOutput() **1.3**
sets the output stream to “end of stream.”
- void shutdownInput() **1.3**
sets the input stream to “end of stream.”
- boolean isOutputShutdown() **1.4**
returns true if output has been shut down.
- boolean isInputShutdown() **1.4**
returns true if input has been shut down.

4.3 Interruptible Sockets

When you connect to a socket, the current thread blocks until the connection has been established or a timeout has elapsed. Similarly, when you read or write data through a socket, the current thread blocks until the operation is successful or has timed out.

In interactive applications, you would like to give users an option to simply cancel a socket connection that does not appear to produce results. However, if a thread blocks on an unresponsive socket, you cannot unblock it by calling `interrupt`.

To interrupt a socket operation, use a `SocketChannel`, a feature of the `java.nio` package. Open the `SocketChannel` like this:

```
SocketChannel channel = SocketChannel.open(new InetSocketAddress(host, port));
```

A channel does not have associated streams. Instead, it has `read` and `write` methods that make use of `Buffer` objects. (See Chapter 2 for more information about NIO buffers.) These methods are declared in the interfaces `ReadableByteChannel` and `WritableByteChannel`.

If you don't want to deal with buffers, you can use the `Scanner` class to read from a `SocketChannel` because `Scanner` has a constructor with a `ReadableByteChannel` parameter:

```
Scanner in = new Scanner(channel, "UTF-8");
```

To turn a channel into an output stream, use the static `Channels.newOutputStream` method.

```
OutputStream outStream = Channels.newOutputStream(channel);
```

That's all you need to do. Whenever a thread is interrupted during an open, read, or write operation, the operation does not block, but is terminated with an exception.

The program in Listing 4.5 contrasts interruptible and blocking sockets. A server sends numbers and pretends to be stuck after the tenth number. Click on either button, and a thread is started that connects to the server and prints the output. The first thread uses an interruptible socket; the second thread uses a blocking socket. If you click the Cancel button within the first ten numbers, you can interrupt either thread.

However, after the first ten numbers, you can only interrupt the first thread. The second thread keeps blocking until the server finally closes the connection (see Figure 4.6).

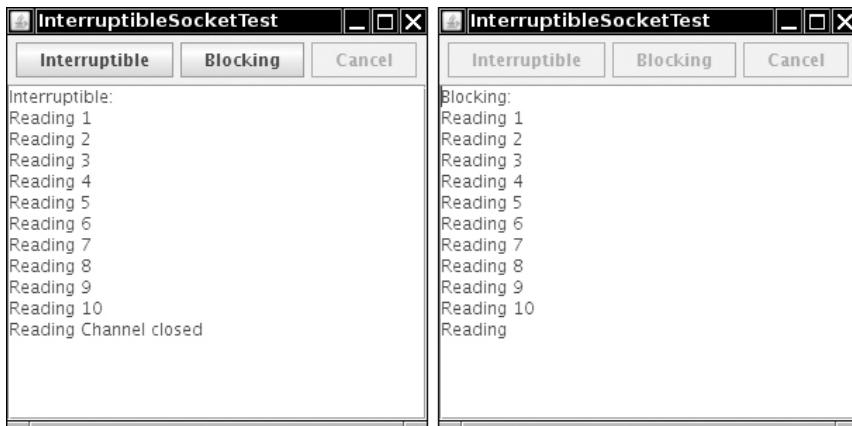


Figure 4.6 Interrupting a socket

Listing 4.5 `interruptible/InterruptibleSocketTest.java`

```
1 package interruptible;  
2  
3 import java.awt.*;  
4 import java.awt.event.*;  
5 import java.util.*;  
6 import java.net.*;  
7 import java.io.*;  
8 import java.nio.channels.*;  
9 import javax.swing.*;  
10  
11 /**
12  * This program shows how to interrupt a socket channel.
13  * @author Cay Horstmann
14  * @version 1.04 2016-04-27
15 */
16 public class InterruptibleSocketTest
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(() ->
21             {
22                 JFrame frame = new InterruptibleFrame();
23                 frame.setTitle("InterruptibleSocketTest");
24                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25                 frame.setVisible(true);
26             });
27     }
}
```

(Continues)

Listing 4.5 (Continued)

```
28 }
29
30 class InterruptibleSocketFrame extends JFrame
31 {
32     private Scanner in;
33     private JButton interruptibleButton;
34     private JButton blockingButton;
35     private JButton cancelButton;
36     private JTextArea messages;
37     private TestServer server;
38     private Thread connectThread;
39
40     public InterruptibleSocketFrame()
41     {
42         JPanel northPanel = new JPanel();
43         add(northPanel, BorderLayout.NORTH);
44
45         final int TEXT_ROWS = 20;
46         final int TEXT_COLUMNS = 60;
47         messages = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
48         add(new JScrollPane(messages));
49
50         interruptibleButton = new JButton("Interruptible");
51         blockingButton = new JButton("Blocking");
52
53         northPanel.add(interruptibleButton);
54         northPanel.add(blockingButton);
55
56         interruptibleButton.addActionListener(event ->
57         {
58             interruptibleButton.setEnabled(false);
59             blockingButton.setEnabled(false);
60             cancelButton.setEnabled(true);
61             connectThread = new Thread(() ->
62             {
63                 try
64                 {
65                     connectInterruptibly();
66                 }
67                 catch (IOException e)
68                 {
69                     messages.append("\nInterruptibleSocketTest.connectInterruptibly: " + e);
70                 }
71             });
72             connectThread.start();
73         });
74 }
```

```
75     blockingButton.addActionListener(event ->
76     {
77         interruptibleButton.setEnabled(false);
78         blockingButton.setEnabled(false);
79         cancelButton.setEnabled(true);
80         connectThread = new Thread(() ->
81             {
82                 try
83                 {
84                     connectBlocking();
85                 }
86                 catch (IOException e)
87                 {
88                     messages.append("\nInterruptibleSocketTest.connectBlocking: " + e);
89                 }
90             });
91         connectThread.start();
92     });
93
94     cancelButton = new JButton("Cancel");
95     cancelButton.setEnabled(false);
96     northPanel.add(cancelButton);
97     cancelButton.addActionListener(event ->
98     {
99         connectThread.interrupt();
100        cancelButton.setEnabled(false);
101    });
102    server = new TestServer();
103    new Thread(server).start();
104    pack();
105 }
106 /**
107 * Connects to the test server, using interruptible I/O.
108 */
109 public void connectInterruptibly() throws IOException
110 {
111     messages.append("Interruptible:\n");
112     try (SocketChannel channel = SocketChannel.open(new InetSocketAddress("localhost", 8189)))
113     {
114         in = new Scanner(channel, "UTF-8");
115         while (!Thread.currentThread().isInterrupted())
116         {
117             messages.append("Reading ");
118             if (in.hasNextLine())
119             {
120                 String line = in.nextLine();
121                 messages.append(line);
122                 messages.append("\n");
123             }
124         }
125     }
126 }
```

(Continues)

Listing 4.5 (Continued)

```
124         }
125     }
126 }
127 finally
128 {
129     EventQueue.invokeLater(() ->
130     {
131         messages.append("Channel closed\n");
132         interruptibleButton.setEnabled(true);
133         blockingButton.setEnabled(true);
134     });
135 }
136 }
137 /**
138 * Connects to the test server, using blocking I/O.
139 */
140 public void connectBlocking() throws IOException
141 {
142     messages.append("Blocking:\n");
143     try (Socket sock = new Socket("localhost", 8189))
144     {
145         in = new Scanner(sock.getInputStream(), "UTF-8");
146         while (!Thread.currentThread().isInterrupted())
147         {
148             messages.append("Reading ");
149             if (in.hasNextLine())
150             {
151                 String line = in.nextLine();
152                 messages.append(line);
153                 messages.append("\n");
154             }
155         }
156     }
157 }
158 finally
159 {
160     EventQueue.invokeLater(() ->
161     {
162         messages.append("Socket closed\n");
163         interruptibleButton.setEnabled(true);
164         blockingButton.setEnabled(true);
165     });
166 }
167 }
168 /**
169 * A multithreaded server that listens to port 8189 and sends numbers to the client, simulating
170 * a hanging server after 10 numbers.
```

```
172     */
173 class TestServer implements Runnable
174 {
175     public void run()
176     {
177         try (ServerSocket s = new ServerSocket(8189))
178         {
179             while (true)
180             {
181                 Socket incoming = s.accept();
182                 Runnable r = new TestServerHandler(incoming);
183                 Thread t = new Thread(r);
184                 t.start();
185             }
186         }
187         catch (IOException e)
188         {
189             messages.append("\nTestServer.run: " + e);
190         }
191     }
192 }
193 /**
194 * This class handles the client input for one server socket connection.
195 */
196 class TestServerHandler implements Runnable
197 {
198     private Socket incoming;
199     private int counter;
200
201     /**
202      * Constructs a handler.
203      * @param i the incoming socket
204      */
205     public TestServerHandler(Socket i)
206     {
207         incoming = i;
208     }
209
210     public void run()
211     {
212         try
213         {
214             try
215             {
216                 OutputStream outStream = incoming.getOutputStream();
217                 PrintWriter out = new PrintWriter(
218                     new OutputStreamWriter(outStream, "UTF-8"),
219                     true /* autoFlush */);
220             }
221         }
222     }
223 }
```

(Continues)

Listing 4.5 (Continued)

```
221         while (counter < 100)
222         {
223             counter++;
224             if (counter <= 10) out.println(counter);
225             Thread.sleep(100);
226         }
227     }
228     finally
229     {
230         incoming.close();
231         messages.append("Closing server\n");
232     }
233 }
234 catch (Exception e)
235 {
236     messages.append("\nTestServerHandler.run: " + e);
237 }
238 }
239 }
240 }
```

java.net.InetSocketAddress 1.4

- `InetSocketAddress(String hostname, int port)`

constructs an address object with the given host and port, resolving the host name during construction. If the host name cannot be resolved, the address object's `unresolved` property is set to true.

- `boolean isUnresolved()`

returns true if this address object could not be resolved.

java.nio.channels.SocketChannel 1.4

- `static SocketChannel open(SocketAddress address)`

opens a socket channel and connects it to a remote address.

java.nio.channels.Channels 1.4

- `static InputStream newInputStream(ReadableByteChannel channel)`

constructs an input stream that reads from the given channel.

(Continues)

java.nio.channels.Channels 1.4 (Continued)

- static OutputStream newOutputStream(WritableByteChannel channel)
constructs an output stream that writes to the given channel.

4.4 Getting Web Data

To access web servers in a Java program, you will want to work at a higher level than socket connections and HTTP requests. In the following sections, we discuss the classes that the Java library provides for this purpose.

4.4.1 URLs and URIs

The `URL` and `URLConnection` classes encapsulate much of the complexity of retrieving information from a remote site. You can construct a `URL` object from a string:

```
URL url = new URL(urlString);
```

If you simply want to fetch the contents of the resource, use the `openStream` method of the `URL` class. This method yields an `InputStream` object. Use it in the usual way—for example, to construct a `Scanner`:

```
InputStream inStream = url.openStream();
Scanner in = new Scanner(inStream, "UTF-8");
```

The `java.net` package makes a useful distinction between URLs (uniform resource *locators*) and URIs (uniform resource *identifiers*).

A URI is a purely syntactical construct that contains the various parts of the string specifying a web resource. A URL is a special kind of URI, namely, one with sufficient information to *locate* a resource. Other URIs, such as

`mailto:cay@horstmann.com`

are not locators—there is no data to locate from this identifier. Such a URI is called a URN (uniform resource *name*).

In the Java library, the `URI` class has no methods for accessing the resource that the identifier specifies—its sole purpose is parsing. In contrast, the `URL` class can open a stream to the resource. For that reason, the `URL` class only works with schemes that the Java library knows how to handle, such as `http:`, `https:`, `ftp:`, the local file system (`file:`), and JAR files (`jar:`).

To see why parsing is not trivial, consider how complex URIs can be. For example,

```
http://google.com?q=Beach+Chalet  
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

The URI specification gives the rules for the makeup of these identifiers. A URI has the syntax

```
[scheme:] schemeSpecificPart[#fragment]
```

Here, the [. . .] denotes an optional part, and the : and # are included literally in the identifier.

If the *scheme:* part is present, the URI is called *absolute*. Otherwise, it is called *relative*.

An absolute URI is *opaque* if the *schemeSpecificPart* does not begin with a / such as

```
mailto:cay@horstmann.com
```

All absolute nonopaque URIs and all relative URIs are *hierarchical*. Examples are

```
http://horstmann.com/index.htm  
../../../java/net/Socket.html#Socket()
```

The *schemeSpecificPart* of a hierarchical URI has the structure

```
[//authority] [path] [?query]
```

where, again, [. . .] denotes optional parts.

For server-based URIs, the *authority* part has the form

```
[user-info@] host[:port]
```

The *port* must be an integer.

RFC 2396, which standardizes URIs, also supports a registry-based mechanism in which the *authority* has a different format, but this is not in common use.

One of the purposes of the `URI` class is to parse an identifier and break it up into its components. You can retrieve them with the methods

```
getScheme  
getSchemeSpecificPart  
getAuthority  
getUserInfo  
getHost  
getPort  
getPath  
getQuery  
getFragment
```

The other purpose of the `URI` class is the handling of absolute and relative identifiers. If you have an absolute URI such as

```
http://docs.mycompany.com/api/java/net/ServerSocket.html
```

and a relative URI such as

```
.../java/net/Socket.html#Socket()
```

then you can combine the two into an absolute URI.

```
http://docs.mycompany.com/api/java/net/Socket.html#Socket()
```

This process is called *resolving* a relative URL.

The opposite process is called *relativization*. For example, suppose you have a *base* URI

```
http://docs.mycompany.com/api
```

and a URI

```
http://docs.mycompany.com/api/java/lang/String.html
```

Then the relativized URI is

```
java/lang/String.html
```

The `URI` class supports both of these operations:

```
relative = base.relativize(combined);  
combined = base.resolve(relative);
```

4.4.2 Using a `URLConnection` to Retrieve Information

If you want additional information about a web resource, you should use the `URLConnection` class, which gives you much more control than the basic `URL` class.

When working with a `URLConnection` object, you must carefully schedule your steps.

1. Call the `openConnection` method of the `URL` class to obtain the `URLConnection` object:

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

2. Set any request properties, using the methods

```
setDoInput  
setDoOutput  
setIfModifiedSince  
setUseCaches  
setAllowUserInteraction  
setRequestProperty  
setConnectTimeout  
setReadTimeout
```

We discuss these methods later in this section and in the API notes.

3. Connect to the remote resource by calling the `connect` method.

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

Besides making a socket connection to the server, this method also queries the server for *header information*.

4. After connecting to the server, you can query the header information. Two methods, `getHeaderFieldKey` and `getHeaderField`, enumerate all fields of the header. The method `getHeaderFields` gets a standard `Map` object containing the header fields. For your convenience, the following methods query standard fields:

```
getContentType  
getContentLength  
getContentEncoding  
getDate  
getExpiration  
getLastModified
```

5. Finally, you can access the resource data. Use the `getInputStream` method to obtain an input stream for reading the information. (This is the same input stream that the `openStream` method of the `URL` class returns.) The other method, `getContent`, isn't very useful in practice. The objects that are returned by standard content types such as `text/plain` and `image/gif` require classes in the `com.sun` hierarchy for processing. You could register your own content handlers, but we do not discuss this technique in our book.



CAUTION: Some programmers form the wrong mental image when using the `URLConnection` class, thinking that the `getInputStream` and `getOutputStream` methods are similar to those of the `Socket` class. But that isn't quite true. The `URLConnection` class does quite a bit of magic behind the scenes—in particular, the handling of request and response headers. For that reason, it is important that you follow the setup steps for the connection.

Let us now look at some of the `URLConnection` methods in detail. Several methods set properties of the connection before connecting to the server. The most important ones are `setDoInput` and `setDoOutput`. By default, the connection yields an input stream for reading from the server but no output stream for writing. If you want an output stream (for example, for posting data to a web server), you need to call

```
connection.setDoOutput(true);
```

Next, you may want to set some of the request headers. The request headers are sent together with the request command to the server. Here is an example:

```
GET www.server.com/index.html HTTP/1.0  
Referer: http://www.somewhere.com/links.html  
Proxy-Connection: Keep-Alive
```

```
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)
Host: www.server.com
Accept: text/html, image/gif, image/jpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*;utf-8
Cookie: orangemilano=192218887821987
```

The `setIfModifiedSince` method tells the connection that you are only interested in data modified since a certain date.

The `setUseCaches` and `setAllowUserInteraction` methods should only be called inside applets. The `setUseCaches` method directs the browser to first check the browser cache. The `setAllowUserInteraction` method allows an applet to pop up a dialog box for querying the user name and password for password-protected resources (see Figure 4.7).

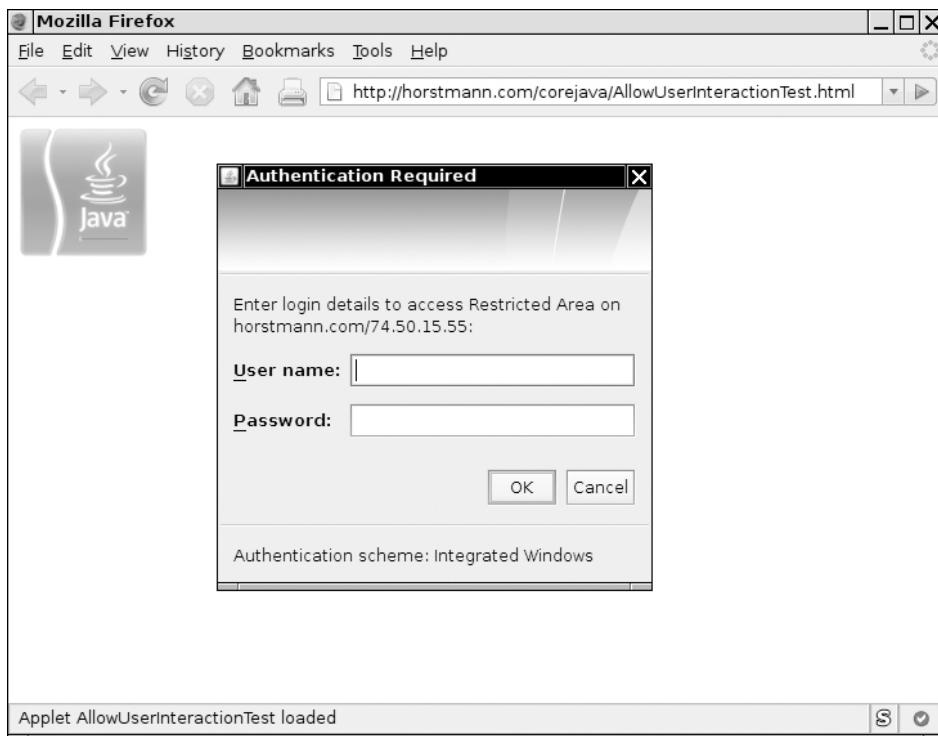


Figure 4.7 A network password dialog box

Finally, you can use the catch-all `setRequestProperty` method to set any name/value pair that is meaningful for the particular protocol. For the format of the HTTP request headers, see RFC 2616. Some of these parameters are not well documented

and are passed around by word of mouth from one programmer to the next. For example, if you want to access a password-protected web page, you must do the following:

1. Concatenate the user name, a colon, and the password.

```
String input = username + ":" + password;
```

2. Compute the Base64 encoding of the resulting string. (The Base64 encoding encodes a sequence of bytes into a sequence of printable ASCII characters.)

```
Base64.Encoder encoder = Base64.getEncoder();
String encoding = encoder.encodeToString(input.getBytes(StandardCharsets.UTF_8));
```

3. Call the `setRequestProperty` method with a name of "Authorization" and the value "Basic " + encoding:

```
connection.setRequestProperty("Authorization", "Basic " + encoding);
```



TIP: You just saw how to access a password-protected web page. To access a password-protected file by FTP, use an entirely different method: Construct a URL of the form

```
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

Once you call the `connect` method, you can query the response header information. First, let's see how to enumerate all response header fields. The implementors of this class felt a need to express their individuality by introducing yet another iteration protocol. The call

```
String key = connection.getHeaderFieldKey(n);
```

gets the `n`th key from the response header, where `n` starts from 1! It returns `null` if `n` is zero or greater than the total number of header fields. There is no method to return the number of fields; you simply keep calling `getHeaderFieldKey` until you get `null`. Similarly, the call

```
String value = connection.getHeaderField(n);
```

returns the `n`th value.

The method `getHeaderFields` returns a `Map` of response header fields.

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

Here is a set of response header fields from a typical HTTP request:

```
Date: Wed, 27 Aug 2008 00:15:48 GMT
```

```
Server: Apache/2.2.2 (Unix)
```

```
Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```

As a convenience, six methods query the values of the most common header types and convert them to numeric types when appropriate. Table 4.1 shows these convenience methods. The methods with return type `long` return the number of seconds since January 1, 1970 GMT.

Table 4.1 Convenience Methods for Response Header Values

Key Name	Method Name	Return Type
Date	getDate	long
Expires	getExpiration	long
Last-Modified	getLastModified	long
Content-Length	getContentLength	int
Content-Type	getContentType	String
Content-Encoding	getContentEncoding	String

The program in Listing 4.6 lets you experiment with URL connections. Supply a URL and an optional user name and password on the command line when running the program, for example:

```
java urlConnection.URLConnectionTest http://www.yourserver.com user password
```

The program prints

- All keys and values of the header
- The return values of the six convenience methods in Table 4.1
- The first ten lines of the requested resource

Listing 4.6 `URLConnection/URLConnectionTest.java`

```
1 package urlConnection;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.*;
7
```

(Continues)

Listing 4.6 (*Continued*)

```
8  /**
9  * This program connects to an URL and displays the response header data and the first 10 lines of
10 * the requested data.
11 *
12 * Supply the URL and an optional username and password (for HTTP basic authentication) on the
13 * command line.
14 * @version 1.11 2007-06-26
15 * @author Cay Horstmann
16 */
17 public class URLConnectionTest
18 {
19     public static void main(String[] args)
20     {
21         try
22         {
23             String urlName;
24             if (args.length > 0) urlName = args[0];
25             else urlName = "http://horstmann.com";
26
27             URL url = new URL(urlName);
28             URLConnection connection = url.openConnection();
29
30             // set username, password if specified on command line
31
32             if (args.length > 2)
33             {
34                 String username = args[1];
35                 String password = args[2];
36                 String input = username + ":" + password;
37                 Base64.Encoder encoder = Base64.getEncoder();
38                 String encoding = encoder.encodeToString(input.getBytes(StandardCharsets.UTF_8));
39                 connection.setRequestProperty("Authorization", "Basic " + encoding);
40             }
41
42             connection.connect();
43
44             // print header fields
45
46             Map<String, List<String>> headers = connection.getHeaderFields();
47             for (Map.Entry<String, List<String>> entry : headers.entrySet())
48             {
49                 String key = entry.getKey();
50                 for (String value : entry.getValue())
51                     System.out.println(key + ": " + value);
52             }
53
54             // print convenience functions
55 }
```

```
56     System.out.println("-----");
57     System.out.println("getContentType: " + connection.getContentType());
58     System.out.println("getContentLength: " + connection.getContentLength());
59     System.out.println("getContentEncoding: " + connection.getContentEncoding());
60     System.out.println("getDate: " + connection.getDate());
61     System.out.println("getExpiration: " + connection.getExpiration());
62     System.out.println("getLastModified: " + connection.getLastModified());
63     System.out.println("-----");
64
65     String encoding = connection.getContentEncoding();
66     if (encoding == null) encoding = "UTF-8";
67     try (Scanner in = new Scanner(connection.getInputStream(), encoding))
68     {
69         // print first ten lines of contents
70
71         for (int n = 1; in.hasNextLine() && n <= 10; n++)
72             System.out.println(in.nextLine());
73         if (in.hasNextLine()) System.out.println("... ");
74     }
75 }
76 catch (IOException e)
77 {
78     e.printStackTrace();
79 }
80 }
81 }
```

java.net.URL 1.0

- `InputStream openStream()`
opens an input stream for reading the resource data.
- `URLConnection openConnection();`
returns a `URLConnection` object that manages the connection to the resource.

java.netURLConnection 1.0

- `void setDoInput(boolean doInput)`
- `boolean getDoInput()`
If `doInput` is true, the user can receive input from this `URLConnection`.
- `void setDoOutput(boolean doOutput)`
- `boolean getDoOutput()`
If `doOutput` is true, the user can send output to this `URLConnection`.

(Continues)

java.net.URLConnection 1.0 (Continued)

- void setIfModifiedSince(long time)
- long getIfModifiedSince()

The `ifModifiedSince` property configures this `URLConnection` to fetch only data modified since a given time. The time is given in seconds since midnight, GMT, January 1, 1970.

- void setUseCaches(boolean useCaches)
- boolean getUseCaches()

If `useCaches` is true, data can be retrieved from a local cache. Note that the `URLConnection` itself does not maintain such a cache. The cache must be supplied by an external program such as a browser.

- void setAllowUserInteraction(boolean allowUserInteraction)
- boolean getAllowUserInteraction()

If `allowUserInteraction` is true, the user can be queried for passwords. Note that the `URLConnection` itself has no facilities for executing such a query. The query must be carried out by an external program such as a browser or browser plugin.

- void setConnectTimeout(int timeout) **5.0**
- int getConnectTimeout() **5.0**

sets or gets the timeout for the connection (in milliseconds). If the timeout has elapsed before a connection was established, the `connect` method of the associated input stream throws a `SocketTimeoutException`.

- void setReadTimeout(int timeout) **5.0**
- int getReadTimeout() **5.0**

sets or gets the timeout for reading data (in milliseconds). If the timeout has elapsed before a read operation was successful, the `read` method throws a `SocketTimeoutException`.

- void setRequestProperty(String key, String value)
sets a request header field.

- Map<String,List<String>> getRequestProperties() **1.4**

returns a map of request properties. All values for the same key are placed in a list.

- void connect()
connects to the remote resource and retrieves response header information.

- Map<String,List<String>> getHeaderFields() **1.4**

returns a map of response headers. All values for the same key are placed in a list.

(Continues)

java.net.URLConnection 1.0 (Continued)

- `String getHeaderFieldKey(int n)`

gets the key for the `n`th response header field, or `null` if `n` is ≤ 0 or greater than the number of response header fields.

- `String getHeaderField(int n)`

gets value of the `n`th response header field, or `null` if `n` is ≤ 0 or greater than the number of response header fields.

- `int getContentLength()`

gets the content length if available, or `-1` if unknown.

- `String getContentType()`

gets the content type, such as `text/plain` or `image/gif`.

- `String getContentEncoding()`

gets the content encoding, such as `gzip`. This value is not commonly used, because the default identity encoding is not supposed to be specified with a `Content-Encoding` header.

- `long getDate()`

- `long getExpiration()`

- `long getLastModified()`

gets the date of creation, expiration, and last modification of the resource. The dates are specified as seconds since midnight, GMT, January 1, 1970.

- `InputStream getInputStream()`

- `OutputStream getOutputStream()`

returns a stream for reading from the resource or writing to the resource.

- `Object getContent()`

selects the appropriate content handler to read the resource data and convert it into an object. This method is not useful for reading standard types such as `text/plain` or `image/gif` unless you install your own content handler.

4.4.3 Posting Form Data

In the preceding section, you saw how to read data from a web server. Now we will show you how your programs can send data back to a web server and to programs that the web server invokes.

To send information from a web browser to the web server, a user fills out a *form*, like the one in Figure 4.8.

Figure 4.8 An HTML form

When the user clicks the Submit button, the text in the text fields and the settings of any checkboxes, radio buttons, and other input elements are sent back to the web server. The web server invokes a program that processes the user input.

Many technologies enable web servers to invoke programs. Among the best known ones are Java servlets, JavaServer Faces, Microsoft Active Server Pages (ASP), and Common Gateway Interface (CGI) scripts.

The server-side program processes the form data and produces another HTML page that the web server sends back to the browser. This sequence is illustrated in Figure 4.9. The response page can contain new information (for example, in

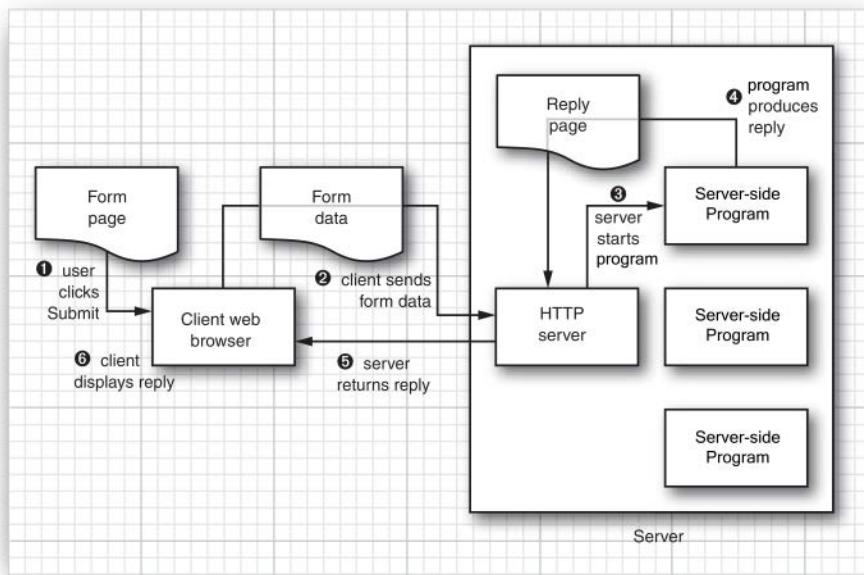


Figure 4.9 Data flow during execution of a server-side program

an information-search program) or just an acknowledgment. The web browser then displays the response page.

We do not discuss the implementation of server-side programs in this book. Our interest is merely in writing client programs that interact with existing server-side programs.

When form data are sent to a web server, it does not matter whether the data are interpreted by a servlet, a CGI script, or some other server-side technology. The client sends the data to the web server in a standard format, and the web server takes care of passing it on to the program that generates the response.

Two commands, called `GET` and `POST`, are commonly used to send information to a web server.

In the `GET` command, you simply attach query parameters to the end of the URL. The URL has the form

```
http://host/path?query
```

Each parameter has the form `name=value`. Parameters are separated by & characters. Parameter values are encoded using the *URL encoding* scheme, following these rules:

- Leave the characters A through Z, a through z, 0 through 9, and . - ~ unchanged.
- Replace all spaces with + characters.
- Encode all other characters into UTF-8 and encode each byte by a %, followed by a two-digit hexadecimal number.

For example, to transmit *San Francisco, CA*, you use `San+Francisco%2c+CA`, as the hexadecimal number `2c` is the UTF-8 code of the `,` character.

This encoding keeps any intermediate programs from messing with spaces and interpreting other special characters.

For example, at the time of this writing, the Google Maps site (www.google.com/maps) accepts query parameters with names `q` and `hl` whose values are the location query and the human language of the response. To get a map of 1 Market Street in San Francisco, with a response in German, use the following URL:

```
http://www.google.com/maps?q=1+Market+Street+San+Francisco&hl=de
```

Very long query strings can look unattractive in browsers, and older browsers and proxies have a limit on the number of characters that you can include in a GET request. For that reason, a POST request is often used for forms with a lot of data. In a POST request, you do not attach parameters to a URL. Instead, you get an output stream from the `URLConnection` and write name/value pairs to the output stream. You still have to URL-encode the values and separate them with & characters.

Let us look at this process in detail. To post data to a server-side program, first establish a `URLConnection`:

```
URL url = new URL("http://host/path");
URLConnection connection = url.openConnection();
```

Then, call the `setDoOutput` method to set up the connection for output:

```
connection.setDoOutput(true);
```

Next, call `getOutputStream` to get a stream through which you can send data to the server. If you are sending text to the server, it is convenient to wrap that stream into a `PrintWriter`.

```
PrintWriter out = new PrintWriter(connection.getOutputStream(), "UTF-8");
```

Now you are ready to send data to the server:

```
out.print(name1 + "=" + URLEncoder.encode(value1, "UTF-8") + "&");
out.print(name2 + "=" + URLEncoder.encode(value2, "UTF-8"));
```

Close the output stream.

```
out.close();
```

Finally, call `getInputStream` and read the server response.

Let's run through a practical example. The web site at <https://www.usps.com/zip4/> contains a form to find the zip code for a street address (see Figure 4.8 on p. 268). To use this form in a Java program, you need to know the URL and the parameters of the POST request.

You could get that information by looking at the HTML code of the form, but it is usually easier to "spy" on a request with a network monitor. Most browsers have a network monitor as part of their development toolkit. For example, Figure 4.10 shows a screen capture of the Firefox network monitor when submitting data to our example web site. You can find out the submission URL as well as the parameter names and values.

When posting form data, the HTTP header includes the content type and content length:

```
Content-Type: application/x-www-form-urlencoded
```

You can also post data in other formats. For example, when sending data in JavaScript Object Notation (JSON), set the content type to `application/json`.

The header for a POST must also include the content length, for example

```
Content-Length: 124
```

The program in Listing 4.7 sends POST form data to any server-side program. Place the data into a `.properties` file such as the following:

```
url=https://tools.usps.com/go/ZipLookupAction.action
tAddress=1 Market Street
tCity=San Francisco
sState=CA
...
```

The program removes the `url` entry and sends all others to the `doPost` method.

In the `doPost` method, we first open the connection, call `setDoOutput(true)`, and open the output stream. We then enumerate all keys and values. For each of them, we send the key, = character, value, and & separator character:

```
out.print(key);
out.print('=');
out.print(URLEncoder.encode(value, "UTF-8"));
if (more pairs) out.print('&');
```

When switching from writing to reading any part of the response, the actual interaction with the server happens. The `Content-Length` header is set to the size of the output. The `Content-Type` header is set to `application/x-www-form-urlencoded` unless a different

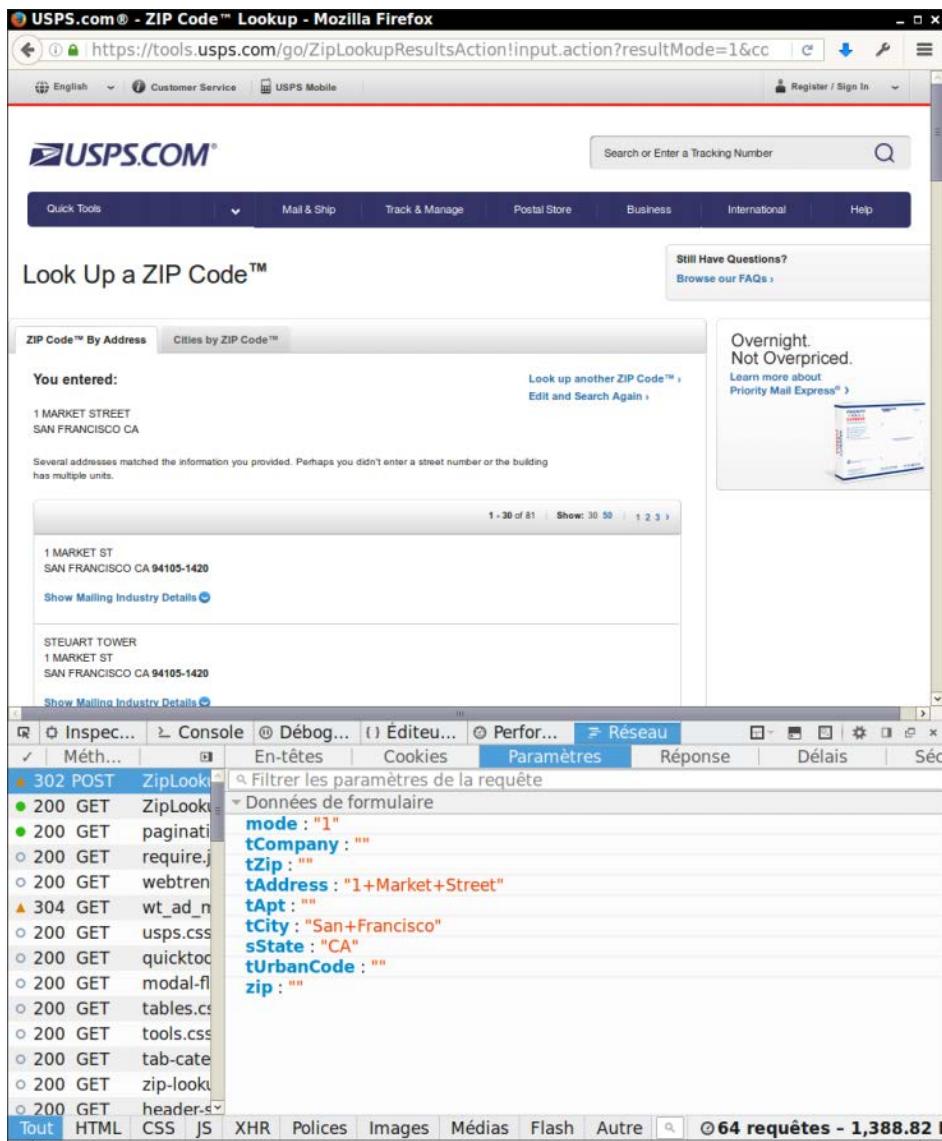


Figure 4.10 An HTML form

content type was specified. The headers and data are sent to the server. Then the response headers and server response are read and can be queried. In our example program, this switch happens in the call to `connection.getContentEncoding()`.

There is one twist with reading the response. If a server-side error occurs, the call to `connection.getInputStream()` throws a `FileNotFoundException`. However, the server still sends an error page back to the browser (such as the ubiquitous “Error 404 — page not found”). To capture this error page, call the `getErrorStream` method:

```
InputStream err = connection.getErrorStream();
```

NOTE: The `getErrorStream` method, as well as several other methods in this program, belong to the `HttpURLConnection` subclass of `URLConnection`. If you make a request to an URL that starts with `http://` or `https://`, you can cast the resulting `connection` object to `HttpURLConnection`.

When you send `POST` data to a server, it can happen that the server-side program responds with a *redirect*: a different URL that should be called to get the actual information. The server could do that because the information is available elsewhere, or to provide a bookmarkable URL. The `HttpURLConnection` class can handle redirects in most cases.

NOTE: If cookies need to be sent from one site to another in a redirect, you can configure the global cookie handler like this:

```
CookieHandler.setDefault(new CookieManager(null, CookiePolicy.ACCEPT_ALL));
```

Then cookies will be properly included in the redirect.

Even though redirects are usually automatically handled, there are some situations where you need to do them yourself. Automatic redirects between HTTP and HTTPS are not supported for security reasons. Redirects can also fail for more subtle reasons. For example, the zip code service does not work if the `User-Agent` request parameter contains the string `Java`, perhaps because the post office doesn't want to serve programmatic requests. While it is possible to set the user agent to a different string in the initial request, that setting is not used in automatic redirects. Instead, automatic redirects always send a generic user agent string that contains the word `Java`.

In such situations, you can manually carry out the redirects. Before connecting the server, turn off automatic redirects:

```
connection.setInstanceFollowRedirects(false);
```

After making the request, get the response code:

```
int responseCode = connection.getResponseCode();
```

Check if it is one of

```
HttpURLConnection.HTTP_MOVED_PERM  
HttpURLConnection.HTTP_MOVED_TEMP  
HttpURLConnection.HTTP_SEE_OTHER
```

In that case, get the `Location` response header to obtain the URL for the redirect. Then disconnect and make another connection to the new URL:

```
String location = connection.getHeaderField("Location");  
if (location != null)  
{  
    URL base = connection.getURL();  
    connection.disconnect();  
    connection = (HttpURLConnection) new URL(base, location).openConnection();  
    . . .  
}
```

The techniques that this program illustrates can be useful whenever you need to query information from an existing web site. Simply find out the parameters that you need to send, and then strip out the HTML tags and other unnecessary information from the reply.

NOTE: As you can see, it is possible to use the Java library classes to interact with web pages, but it is not particularly convenient. Consider using a library such as Apache HttpClient (<http://hc.apache.org/httpcomponents-client-ga>) instead.

Listing 4.7 post/PostTest.java

```
1 package post;  
2  
3 import java.io.*;  
4 import java.net.*;  
5 import java.nio.file.*;  
6 import java.util.*;  
7  
8 /**  
9  * This program demonstrates how to use the URLConnection class for a POST request.  
10 * @version 1.40 2016-04-24  
11 * @author Cay Horstmann  
12 */  
13 public class PostTest  
14 {  
15     public static void main(String[] args) throws IOException  
16     {  
17         String propsFilename = args.length > 0 ? args[0] : "post/post.properties";  
18         Properties props = new Properties();  
19         try (InputStream in = Files.newInputStream(Paths.get(propsFilename)))  
20         {
```

```
21     props.load(in);
22 }
23 String urlString = props.remove("url").toString();
24 Object userAgent = props.remove("User-Agent");
25 Object redirects = props.remove("redirects");
26 CookieHandler.setDefault(new CookieManager(null, CookiePolicy.ACCEPT_ALL));
27 String result = doPost(new URL(urlString), props,
28     userAgent == null ? null : userAgent.toString(),
29     redirects == null ? -1 : Integer.parseInt(redirects.toString()));
30 System.out.println(result);
31 }
32 /**
33 * Do an HTTP POST.
34 * @param url the URL to post to
35 * @param nameValuePairs the query parameters
36 * @param userAgent the user agent to use, or null for the default user agent
37 * @param redirects the number of redirects to follow manually, or -1 for automatic redirects
38 * @return the data returned from the server
39 */
40 public static String doPost(URL url, Map<Object, Object> nameValuePairs, String userAgent,
41     int redirects)
42     throws IOException
43 {
44     HttpURLConnection connection = (HttpURLConnection) url.openConnection();
45     if (userAgent != null)
46         connection.setRequestProperty("User-Agent", userAgent);
47
48     if (redirects >= 0)
49         connection.setInstanceFollowRedirects(false);
50
51     connection.setDoOutput(true);
52
53     try (PrintWriter out = new PrintWriter(connection.getOutputStream()))
54     {
55         boolean first = true;
56         for (Map.Entry<Object, Object> pair : nameValuePairs.entrySet())
57         {
58             if (first) first = false;
59             else out.print('&');
60             String name = pair.getKey().toString();
61             String value = pair.getValue().toString();
62             out.print(name);
63             out.print('=');
64             out.print(URLEncoder.encode(value, "UTF-8"));
65         }
66     }
67     String encoding = connection.getContentEncoding();
68     if (encoding == null) encoding = "UTF-8";
```

(Continues)

Listing 4.7 (Continued)

```
70
71     if (redirects > 0)
72     {
73         int responseCode = connection.getResponseCode();
74         if (responseCode == HttpURLConnection.HTTP_MOVED_PERM
75             || responseCode == HttpURLConnection.HTTP_MOVED_TEMP
76             || responseCode == HttpURLConnection.HTTP_SEE_OTHER)
77         {
78             String location = connection.getHeaderField("Location");
79             if (location != null)
80             {
81                 URL base = connection.getURL();
82                 connection.disconnect();
83                 return doPost(new URL(base, location), nameValuePairs, userAgent, redirects - 1);
84             }
85         }
86     }
87     else if (redirects == 0)
88     {
89         throw new IOException("Too many redirects");
90     }
91
92     StringBuilder response = new StringBuilder();
93     try (Scanner in = new Scanner(connection.getInputStream(), encoding))
94     {
95         while (in.hasNextLine())
96         {
97             response.append(in.nextLine());
98             response.append("\n");
99         }
100    }
101  }
102 catch (IOException e)
103 {
104     InputStream err = connection.getErrorStream();
105     if (err == null) throw e;
106     try (Scanner in = new Scanner(err))
107     {
108         response.append(in.nextLine());
109         response.append("\n");
110     }
111 }
112
113 return response.toString();
114 }
115 }
```

java.net.HttpURLConnection 1.0

- `InputStream getErrorStream()`

returns a stream from which you can read web server error messages.

java.net.URLEncoder 1.0

- `static String encode(String s, String encoding) 1.4`

returns the URL-encoded form of the string `s`, using the given character encoding scheme. (The recommended scheme is "UTF-8".) In URL encoding, the characters 'A'-'Z', 'a'-'z', '0'-'9', '-', '_', '.', and '~' are left unchanged. Space is encoded into '+', and all other characters are encoded into sequences of encoded bytes of the form "%XY", where 0xXY is the hexadecimal value of the byte.

java.net.URLDecoder 1.2

- `static String decode(String s, String encoding) 1.4`

returns the decoding of the URL encoded string `s` under the given character encoding scheme.

4.5 Sending E-Mail

In the past, it was simple to write a program that sends e-mail by making a socket connection to port 25, the SMTP port. The Simple Mail Transport Protocol (SMTP) describes the format for e-mail messages. Once you are connected to the server, send a mail header (in the SMTP format, which is easy to generate), followed by the mail message.

Here are the details:

1. Open a socket to your host.

```
Socket s = new Socket("mail.yourserver.com", 25); // 25 is SMTP
PrintWriter out = new PrintWriter(s.getOutputStream(), "UTF-8");
```

2. Send the following information to the print stream:

```
HELO sending host
MAIL FROM: sender e-mail address
RCPT TO: recipient e-mail address
DATA
```

```
Subject: subject
(blank line)
mail message (any number of lines)

.
QUIT
```

The SMTP specification (RFC 821) states that lines must be terminated with \r followed by \n.

It used to be that SMTP servers were often willing to route e-mail from anyone. However, in these days of spam floods, most servers have built-in checks and only accept requests from users or IP address ranges that they trust. Authentication usually happens over secure socket connections.

Implementing these authentication schemes manually would be very tedious. Instead, we will show you how to use the JavaMail API to send e-mail from a Java program.

Download JavaMail from www.oracle.com/technetwork/java/javamail and unzip it somewhere on your hard disk.

To use JavaMail, you need to set up some properties that depend on your mail server. For example, with GMail, you use

```
mail.transport.protocol=smtpls
mail.smtps.auth=true
mail.smtps.host=smtp.gmail.com
mail.smtps.user=cayhorstmann@gmail.com
```

Our sample program reads these from a property file.

For security reasons, we don't put the password into the property file but instead prompt for it.

Read in the property file, then get a mail session like this:

```
Session mailSession = Session.getDefaultInstance(props);
```

Make a message with the desired sender, recipient, subject, and message text:

```
MimeMessage message = new MimeMessage(mailSession);
message.setFrom(new InternetAddress(from));
message.addRecipient(RecipientType.TO, new InternetAddress(to));
message.setSubject(subject);
message.setText(builder.toString());
```

Then send it off:

```
Transport tr = mailSession.getTransport();
tr.connect(null, password);
tr.sendMessage(message, message.getAllRecipients());
tr.close();
```

The program in Listing 4.8 reads the message from a text file of the format

Sender Recipient Subject Message text (any number of lines)

To run the program, type

```
java -classpath .:path/to/mail.jar path/to/message.txt
```

Here, `mail.jar` is the JAR file that came with the JavaMail distribution. (Windows users: Remember to type a semicolon instead of a colon in the classpath.)

At the time of this writing, GMail does not check the veracity of the information—you can supply any sender you like. (Keep this in mind the next time you get an e-mail message from `president@whitehouse.gov` inviting you to a black-tie affair on the front lawn.)



TIP: If you can't figure out why your mail connection isn't working, call

```
mailSession.setDebug(true);
```

and check out the messages. Also, the JavaMail API FAQ has some useful hints.

Listing 4.8 mail/MailTest.java

```
1 package mail;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
6 import java.util.*;
7 import javax.mail.*;
8 import javax.mail.internet.*;
9 import javax.mail.internet.MimeMessage.RecipientType;
10
11 /**
12 * This program shows how to use JavaMail to send mail messages.
13 * @author Cay Horstmann
14 * @version 1.00 2012-06-04
15 */
16 public class MailTest
17 {
18     public static void main(String[] args) throws MessagingException, IOException
19     {
20         Properties props = new Properties();
21         try (InputStream in = Files.newInputStream(Paths.get("mail", "mail.properties")))
22         {
23             props.load(in);
24         }
```

(Continues)

Listing 4.8 (Continued)

```
25 List<String> lines = Files.readAllLines(Paths.get(args[0]), Charset.forName("UTF-8"));
26
27     String from = lines.get(0);
28     String to = lines.get(1);
29     String subject = lines.get(2);
30
31     StringBuilder builder = new StringBuilder();
32     for (int i = 3; i < lines.size(); i++)
33     {
34         builder.append(lines.get(i));
35         builder.append("\n");
36     }
37
38     Console console = System.console();
39     String password = new String(console.readPassword("Password: "));
40
41     Session mailSession = Session.getDefaultInstance(props);
42     // mailSession.setDebug(true);
43     MimeMessage message = new MimeMessage(mailSession);
44     message.setFrom(new InternetAddress(from));
45     message.addRecipient(RecipientType.TO, new InternetAddress(to));
46     message.setSubject(subject);
47     message.setText(builder.toString());
48     Transport tr = mailSession.getTransport();
49     try
50     {
51         tr.connect(null, password);
52         tr.sendMessage(message, message.getAllRecipients());
53     }
54     finally
55     {
56         tr.close();
57     }
58 }
59 }
```

In this chapter, you have seen how to write network clients and servers in Java and how to harvest information from web servers. The next chapter covers database connectivity. You will learn how to work with relational databases in Java, using the JDBC API.

CHAPTER

5

Database Programming

In this chapter

- 5.1 The Design of JDBC, page 282
- 5.2 The Structured Query Language, page 285
- 5.3 JDBC Configuration, page 291
- 5.4 Working with JDBC Statements, page 297
- 5.5 Query Execution, page 309
- 5.6 Scrollable and Updatable Result Sets, page 321
- 5.7 Row Sets, page 328
- 5.8 Metadata, page 333
- 5.9 Transactions, page 344
- 5.10 Advanced SQL Types, page 347
- 5.11 Connection Management in Web and Enterprise Applications, page 349

In 1996, Sun released the first version of the JDBC API. This API lets programmers connect to a database to query or update it using the Structured Query Language (SQL). (SQL, usually pronounced “sequel,” is an industry standard for relational database access.) JDBC has since become one of the most commonly used APIs in the Java library.

JDBC has been updated several times. As part of the Java SE 1.2 release in 1998, a second version of JDBC was issued. JDBC 3 is included with Java SE 1.4 and

5.0. As this book is published, JDBC 4.2, the version included with Java SE 8, is the most current version.

In this chapter, we will explain the key ideas behind JDBC. We will introduce you to (or refresh your memory of) SQL, the industry-standard Structured Query Language for relational databases. We will then provide enough details and examples to let you start using JDBC for common programming situations.

NOTE: According to Oracle, JDBC is a trademarked term and not an acronym for Java Database Connectivity. It was named to be reminiscent of ODBC, a standard database API pioneered by Microsoft and since incorporated into the SQL standard.

5.1 The Design of JDBC

From the start, the developers of the Java technology were aware of the potential that Java showed for working with databases. In 1995, they began working on extending the standard Java library to deal with SQL access to databases. What they first hoped to do was to extend Java so that a program could talk to any random database using only “pure” Java. It didn’t take them long to realize that this is an impossible task: There are simply too many databases out there, using too many protocols. Moreover, although database vendors were all in favor of Java providing a standard network protocol for database access, they were only in favor of it if Java used *their* network protocol.

What all the database vendors and tool vendors *did* agree on was that it would be useful for Java to provide a pure Java API for SQL access along with a driver manager to allow third-party drivers to connect to specific databases. Database vendors could provide their own drivers to plug in to the driver manager. There would then be a simple mechanism for registering third-party drivers with the driver manager.

This organization follows the very successful model of Microsoft’s ODBC which provided a C programming language interface for database access. Both JDBC and ODBC are based on the same idea: Programs written according to the API talk to the driver manager, which, in turn, uses a driver to talk to the actual database.

This means the JDBC API is all that most programmers will ever have to deal with.

5.1.1 JDBC Driver Types

The JDBC specification classifies drivers into the following *types*:

- A *type 1 driver* translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Early versions of Java included one such driver, the *JDBC/ODBC bridge*. However, the bridge requires deployment and proper configuration of an ODBC driver. When JDBC was first released, the bridge was handy for testing, but it was never intended for production use. At this point, many better drivers are available, and Java 8 no longer provides the JDBC/ODBC bridge.
- A *type 2 driver* is written partly in Java and partly in native code; it communicates with the client API of a database. When using such a driver, you must install some platform-specific code onto the client in addition to a Java library.
- A *type 3 driver* is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This simplifies deployment because the platform-specific code is located only on the server.
- A *type 4 driver* is a pure Java library that translates JDBC requests directly to a database-specific protocol.

NOTE: The JDBC specification is available at http://download.oracle.com/otndocs/jcp/jdbc-4_2-mrel2-spec/.

Most database vendors supply either a type 3 or type 4 driver with their database. Furthermore, a number of third-party companies specialize in producing drivers with better standards conformance, support for more platforms, better performance, or, in some cases, simply better reliability than the drivers provided by the database vendors.

In summary, the ultimate goal of JDBC is to make possible the following:

- Programmers can write applications in the Java programming language to access any database using standard SQL statements (or even specialized extensions of SQL) while still following Java language conventions.
- Database vendors and database tool vendors can supply the low-level drivers. Thus, they can optimize their drivers for their specific products.

NOTE: If you are curious as to why Java just didn't adopt the ODBC model, the reason, as given at the JavaOne conference in 1996, was this:

- ODBC is hard to learn.
 - ODBC has a few commands with lots of complex options. The preferred style in the Java programming language is to have simple and intuitive methods, but to have lots of them.
 - ODBC relies on the use of `void*` pointers and other C features that are not natural in the Java programming language.
 - An ODBC-based solution is inherently less safe and harder to deploy than a pure Java solution.
-

5.1.2 Typical Uses of JDBC

The traditional client/server model has a rich GUI on the client and a database on the server (see Figure 5.1). In this model, a JDBC driver is deployed on the client.

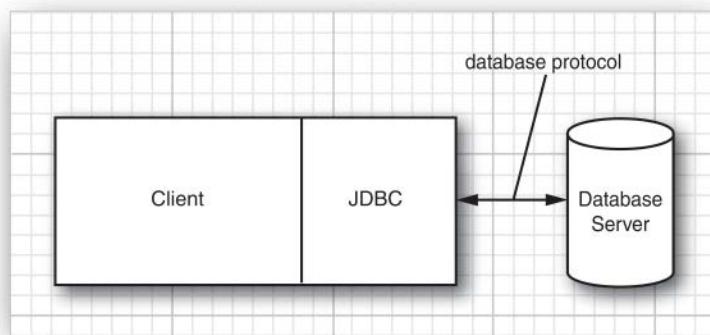


Figure 5.1 A traditional client/server application

However, nowadays it is far more common to have a three-tier model where the client application does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates *visual presentation* (on the client) from the

business logic (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java desktop application, a web browser, or a mobile app.

Communication between the client and the middle tier typically occurs through HTTP. JDBC manages the communication between the middle tier and the back-end database. Figure 5.2 shows the basic architecture.

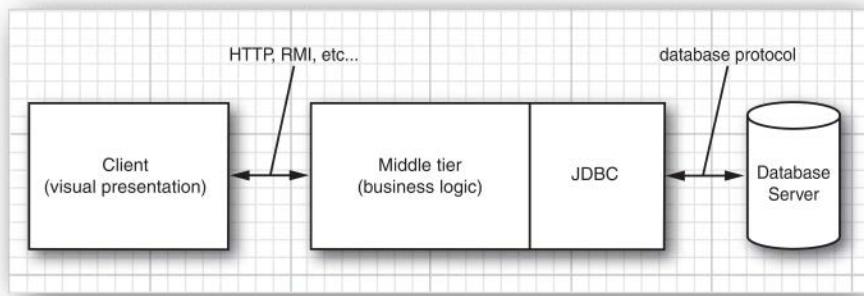


Figure 5.2 A three-tier application

5.2 The Structured Query Language

JDBC lets you communicate with databases using SQL, which is the command language for essentially all modern relational databases. Desktop databases usually have a GUI that lets users manipulate the data directly, but server-based databases are accessed purely through SQL.

The JDBC package can be thought of as nothing more than an API for communicating SQL statements to databases. We will briefly introduce SQL in this section. If you have never seen SQL before, you might not find this material sufficient. If so, turn to one of the many learning resources on the topic; we recommend *Learning SQL* by Alan Beaulieu (O'Reilly, 2009) or the online book *Learn SQL The Hard Way* at <http://sql.learncodethehardway.org/>.

You can think of a database as a bunch of named tables with rows and columns. Each column has a *column name*. Each row contains a set of related data.

As an example database for this book, we use a set of database tables that describe a collection of classic computer science books (see Tables 5.1 through 5.4).

Table 5.1 The Authors Table

Author_ID	Name	Fname
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
...

Table 5.2 The Books Table

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
...

Table 5.3 The BooksAuthors Table

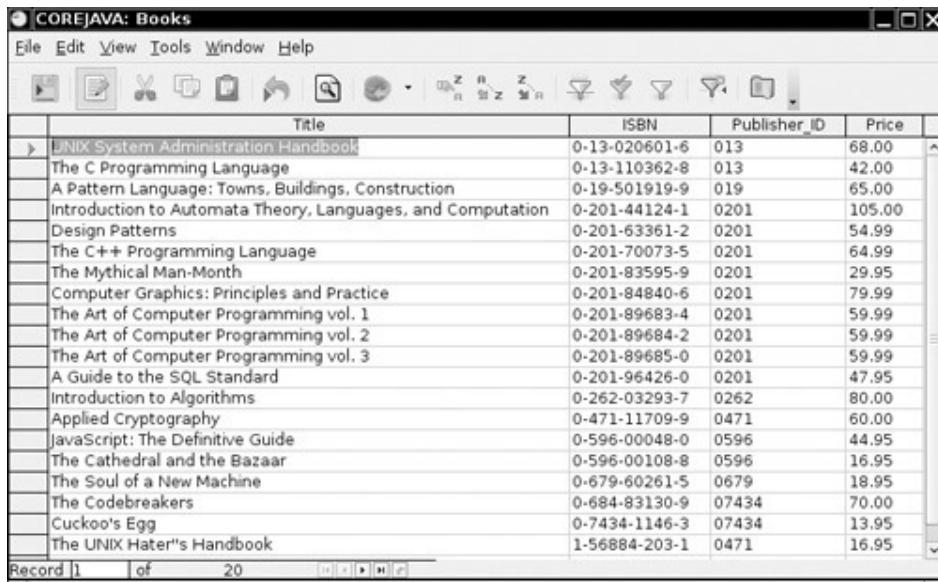
ISBN	Author_ID	Seq_No
0-201-96426-0	DATE	1
0-201-96426-0	DARW	2
0-19-501919-9	ALEX	1
...

Table 5.4 The Publishers Table

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com
...

Figure 5.3 shows a view of the Books table. Figure 5.4 shows the result of *joining* this table with the Publishers table. The Books and the Publishers tables each contain an identifier for the publisher. When we join both tables on the publisher code, we obtain a *query result* made up of values from the joined tables. Each row in the result contains the information about a book, together with the publisher

name and web page URL. Note that the publisher names and URLs are duplicated across several rows because we have several rows with the same publisher.



The screenshot shows a graphical interface for managing a database table named 'Books'. The window title is 'COREJAVA: Books'. The menu bar includes File, Edit, View, Tools, Window, and Help. The toolbar contains various icons for database operations like insert, update, delete, and search. The table has four columns: Title, ISBN, Publisher_ID, and Price. The data consists of 20 rows of book information, many of which share the same publisher ID (0201). The last row shows a publisher ID of 0471, indicating multiple publishers for some books. The bottom of the window shows a record navigation bar with 'Record 1 of 20' and standard navigation buttons.

	Title	ISBN	Publisher_ID	Price
>	UNIX System Administration Handbook	0-13-020601-6	013	68.00
	The C Programming Language	0-13-110362-8	013	42.00
	A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
	Introduction to Automata Theory, Languages, and Computation	0-201-44124-1	0201	105.00
	Design Patterns	0-201-63361-2	0201	54.99
	The C++ Programming Language	0-201-70073-5	0201	64.99
	The Mythical Man-Month	0-201-83595-9	0201	29.95
	Computer Graphics: Principles and Practice	0-201-84840-6	0201	79.99
	The Art of Computer Programming vol. 1	0-201-89683-4	0201	59.99
	The Art of Computer Programming vol. 2	0-201-89684-2	0201	59.99
	The Art of Computer Programming vol. 3	0-201-89685-0	0201	59.99
	A Guide to the SQL Standard	0-201-96426-0	0201	47.95
	Introduction to Algorithms	0-262-03293-7	0262	80.00
	Applied Cryptography	0-471-11709-9	0471	60.00
	JavaScript: The Definitive Guide	0-596-00048-0	0596	44.95
	The Cathedral and the Bazaar	0-596-00108-8	0596	16.95
	The Soul of a New Machine	0-679-60261-5	0679	18.95
	The Codebreakers	0-684-83130-9	07434	70.00
	Cuckoo's Egg	0-7434-1146-3	07434	13.95
	The UNIX Hater's Handbook	1-56884-203-1	0471	16.95

Figure 5.3 Sample table containing books

The benefit of joining tables is avoiding unnecessary duplication of data in the database tables. For example, a naive database design might have had columns for the publisher name and URL right in the Books table. But then the database itself, and not just the query result, would have many duplicates of these entries. If a publisher's web address changed, *all* entries would need to be updated. Clearly, this is somewhat error-prone. In the relational model, we distribute data into multiple tables so that no information is unnecessarily duplicated. For example, each publisher's URL is contained only once in the publisher table. If the information needs to be combined, the tables are joined.

In the figures, you can see a graphical tool to inspect and link the tables. Many vendors have tools to express queries in a simple form by connecting column names and filling information into forms. Such tools are often called *query by example* (QBE) tools. In contrast, a query that uses SQL is written out in text, using SQL syntax, for example:

```
SELECT Books.Title, Books.Publisher_Id, Books.Price, Publishers.Name, Publishers.URL  
FROM Books, Publishers  
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

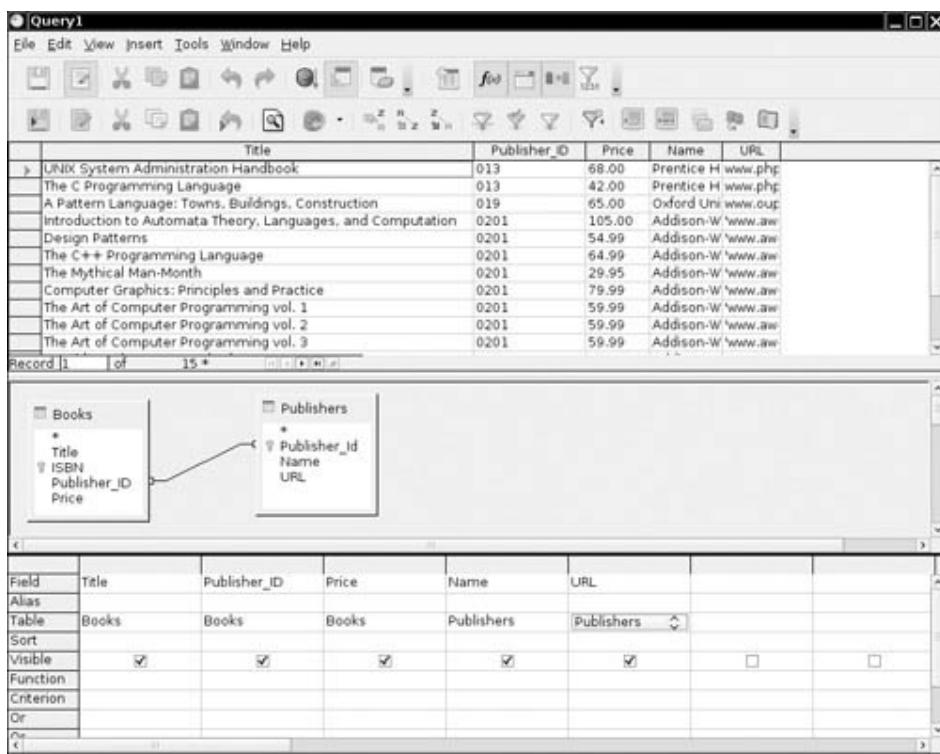


Figure 5.4 Two tables joined together

In the remainder of this section, you will learn how to write such queries. If you are already familiar with SQL, just skip this section.

By convention, SQL keywords are written in capital letters, although this is not necessary.

The SELECT statement is quite flexible. You can simply select all rows in the Books table with the following query:

```
SELECT * FROM Books
```

The FROM clause is required in every SQL SELECT statement. It tells the database which tables to examine to find the data.

You can choose the columns that you want:

```
SELECT ISBN, Price, Title  
FROM Books
```

You can restrict the rows in the answer with the `WHERE` clause:

```
SELECT ISBN, Price, Title  
FROM Books  
WHERE Price <= 29.95
```

Be careful with the “equals” comparison. SQL uses `=` and `<>`, rather than `==` or `!=` as in the Java programming language, for equality testing.

NOTE: Some database vendors support the use of `!=` for inequality testing. This is not standard SQL, so we recommend against such use.

The `WHERE` clause can also use pattern matching by means of the `LIKE` operator. The wildcard characters are not the usual `*` and `?`, however. Use a `%` for zero or more characters and an underscore for a single character. For example,

```
SELECT ISBN, Price, Title  
FROM Books  
WHERE Title NOT LIKE '%n_x%'
```

excludes books with titles that contain words such as Unix or Linux.

Note that strings are enclosed in single quotes, not double quotes. A single quote inside a string is represented by a pair of single quotes. For example,

```
SELECT Title  
FROM Books  
WHERE Title LIKE '%''%'
```

reports all titles that contain a single quote.

You can select data from multiple tables:

```
SELECT * FROM Books, Publishers
```

Without a `WHERE` clause, this query is not very interesting. It lists *all combinations* of rows from both tables. In our case, where `Books` has 20 rows and `Publishers` has 8 rows, the result is a set of rows with 20×8 entries and lots of duplications. We really want to constrain the query to say that we are only interested in *matching* books with their publishers:

```
SELECT * FROM Books, Publishers  
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

This query result has 20 rows, one for each book, because each book has one publisher in the `Publisher` table.

Whenever you have multiple tables in a query, the same column name can occur in two different places. That happened in our example. There is a column called `Publisher_Id` in both the `Books` and the `Publishers` tables. When an ambiguity would otherwise result, you must prefix each column name with the name of the table to which it belongs, such as `Books.Publisher_Id`.

You can use SQL to change the data inside a database as well. For example, suppose you want to reduce by \$5.00 the current price of all books that have “C++” in their title:

```
UPDATE Books  
SET Price = Price - 5.00  
WHERE Title LIKE '%C++%'
```

Similarly, to delete all C++ books, use a `DELETE` query:

```
DELETE FROM Books  
WHERE Title LIKE '%C++%'
```

SQL comes with built-in functions for taking averages, finding maximums and minimums in a column, and so on, which we do not discuss here.

Typically, to insert values into a table, you can use the `INSERT` statement:

```
INSERT INTO Books  
VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201', 47.95)
```

You need a separate `INSERT` statement for every row being inserted in the table.

Of course, before you can query, modify, and insert data, you must have a place to store data. Use the `CREATE TABLE` statement to make a new table. Specify the name and data type for each column. For example,

```
CREATE TABLE Books  
(  
    Title CHAR(60),  
    ISBN CHAR(13),  
    Publisher_Id CHAR(6),  
    Price DECIMAL(10,2)  
)
```

Table 5.5 shows the most common SQL data types.

In this book, we do not discuss the additional clauses, such as keys and constraints, that you can use with the `CREATE TABLE` statement.

Table 5.5 Common SQL Data Types

Data Types	Description
INTEGER or INT	Typically, a 32-bit integer
SMALLINT	Typically, a 16-bit integer
NUMERIC(m, n), DECIMAL(m, n) or DEC(m, n)	Fixed-point decimal number with m total digits and n digits after the decimal point
FLOAT(n)	A floating-point number with n binary digits of precision
REAL	Typically, a 32-bit floating-point number
DOUBLE	Typically, a 64-bit floating-point number
CHARACTER(n) or CHAR(n)	Fixed-length string of length n
VARCHAR(n)	Variable-length strings of maximum length n
BOOLEAN	A Boolean value
DATE	Calendar date, implementation-dependent
TIME	Time of day, implementation-dependent
TIMESTAMP	Date and time of day, implementation-dependent
BLOB	A binary large object
CLOB	A character large object

5.3 JDBC Configuration

Of course, you need a database program for which a JDBC driver is available. There are many excellent choices, such as IBM DB2, Microsoft SQL Server, MySQL, Oracle, and PostgreSQL.

You must also create a database for your experimental use. We assume you name it `COREJAVA`. Create a new database, or have your database administrator create one with the appropriate permissions. You need to be able to create, update, and drop tables in the database.

If you have never installed a client/server database before, you might find that setting up the database is somewhat complex and that diagnosing the cause for failure can be difficult. It might be best to seek expert help if your setup is not working correctly.

If this is your first experience with databases, we recommend that you use the Apache Derby database, which is available from <http://db.apache.org/derby> and also included with some versions of the JDK.

NOTE: The version of Apache Derby that is included with the JDK is officially called JavaDB. We don't think that's particularly helpful, and we will call it Derby in this chapter.

You need to gather a number of items before you can write your first database program. The following sections cover these items.

5.3.1 Database URLs

When connecting to a database, you must use various database-specific parameters such as host names, port numbers, and database names.

JDBC uses a syntax similar to that of ordinary URLs to describe data sources. Here are examples of the syntax:

```
jdbc:derby://localhost:1527/COREJAVA;create=true  
jdbc:postgresql:CoreJAVA
```

These JDBC URLs specify a Derby database and a PostgreSQL database named COREJAVA.

The general syntax is

```
jdbc:subprotocol:other stuff
```

where a subprotocol selects the specific driver for connecting to the database.

The format for the *other stuff* parameter depends on the subprotocol used. You will need to look up your vendor's documentation for the specific format.

5.3.2 Driver JAR Files

You need to obtain the JAR file in which the driver for your database is located. If you use Derby, you need the file `derbyclient.jar`. With another database, you need to locate the appropriate driver. For example, the PostgreSQL drivers are available at <http://jdbc.postgresql.org>.

Include the driver JAR file on the class path when running a program that accesses the database. (You don't need the JAR file for compiling.)

When you launch programs from the command line, simply use the command

```
java -classpath driverPath::. ProgramName
```

On Windows, use a semicolon to separate the current directory (denoted by the `.` character) from the driver JAR location.

5.3.3 Starting the Database

The database server needs to be started before you can connect to it. The details depend on your database.

With the Derby database, follow these steps:

1. Open a command shell and change to a directory that will hold the database files.
2. Locate the file `derbyrun.jar`. With some versions of the JDK, it is contained in the `jdk/db/lib` directory. If it's not there, install Apache Derby and locate the JAR file in the installation directory. We will denote the directory containing `lib/derbyrun.jar` with `derby`.
3. Run the command

```
java -jar derby/lib/derbyrun.jar server start
```

4. Double-check that the database is working correctly. Create a file `ij.properties` that contains these lines:

```
ij.driver=org.apache.derby.jdbc.ClientDriver  
ij.protocol=jdbc:derby://localhost:1527/  
ij.database=COREJAVA;create=true
```

From another command shell, run Derby's interactive scripting tool (called `ij`) by executing

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

Now you can issue SQL commands such as

```
CREATE TABLE Greetings (Message CHAR(20));  
INSERT INTO Greetings VALUES ('Hello, World!');  
SELECT * FROM Greetings;  
DROP TABLE Greetings;
```

Note that each command must be terminated by a semicolon. To exit, type

```
EXIT;
```

5. When you are done using the database, stop the server with the command

```
java -jar derby/lib/derbyrun.jar server shutdown
```

If you use another database, you need to consult the documentation to find out how to start and stop your database server, and how to connect to it and issue SQL commands.

5.3.4 Registering the Driver Class

Many JDBC JAR files (such as the Derby driver included with Java SE 8) automatically register the driver class. In that case, you can skip the manual registration step that we describe in this section. A JAR file can automatically register the driver class if it contains a file `META-INF/services/java.sql.Driver`. You can simply unzip your driver's JAR file to check.

NOTE: This registration mechanism uses a little-known part of the JAR specification; see <http://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Service%20Provider>. Automatic registration is a requirement for a JDBC4-compliant driver.

If your driver's JAR file doesn't support automatic registration, you need to find out the name of the JDBC driver classes used by your vendor. Typical driver names are

```
org.apache.derby.jdbc.ClientDriver  
org.postgresql.Driver
```

There are two ways to register the driver with the `DriverManager`. One way is to load the driver class in your Java program. For example,

```
Class.forName("org.postgresql.Driver"); // force loading of driver class
```

This statement causes the driver class to be loaded, thereby executing a static initializer that registers the driver.

Alternatively, you can set the `jdbc.drivers` property. You can specify the property with a command-line argument, such as

```
java -Djdbc.drivers=org.postgresql.Driver ProgramName
```

Or, your application can set the system property with a call such as

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

You can also supply multiple drivers; separate them with colons, for example

```
org.postgresql.Driver:org.apache.derby.jdbc.ClientDriver
```

5.3.5 Connecting to the Database

In your Java program, open a database connection like this:

```
String url = "jdbc:postgresql:COREJAVA";  
String username = "dbuser";
```

```
String password = "secret";
Connection conn = DriverManager.getConnection(url, username, password);
```

The driver manager iterates through the registered drivers to find a driver that can use the subprotocol specified in the database URL.

The `getconnection` method returns a `Connection` object. In the following sections, you will see how to use the `Connection` object to execute SQL statements.

To connect to the database, you will need to have a user name and password for your database.

NOTE: By default, Derby lets you connect with any user name, and it does not check passwords. A separate set of tables is generated for each user. The default user name is `app`.

The test program in Listing 5.1 puts these steps to work. It loads connection parameters from a file named `database.properties` and connects to the database. The `database.properties` file supplied with the sample code contains connection information for the Derby database. If you use a different database, put your database-specific connection information into that file. Here is an example for connecting to a PostgreSQL database:

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

After connecting to the database, the test program executes the following SQL statements:

```
CREATE TABLE Greetings (Message CHAR(20))
INSERT INTO Greetings VALUES ('Hello, World!')
SELECT * FROM Greetings
```

The result of the `SELECT` statement is printed, and you should see an output of

Hello, World!

Then the table is removed by executing the statement

```
DROP TABLE Greetings
```

To run this test, start your database, as described previously, and launch the program as

```
java -classpath .:driverJAR test.TestDB
```

(As always, Windows users need to use ; instead of : to separate the path elements.)



TIP: One way to debug JDBC-related problems is to enable JDBC tracing. Call the `DriverManager.setLogWriter` method to send trace messages to a `PrintWriter`. The trace output contains a detailed listing of the JDBC activity. Most JDBC driver implementations provide additional mechanisms for tracing. For example, with Derby, you can add a `traceFile` option to the JDBC URL:
`jdbc:derby://localhost:1527/COREJAVA;create=true;traceFile=trace.out.`

Listing 5.1 test/TestDB.java

```
1 package test;
2
3 import java.nio.file.*;
4 import java.sql.*;
5 import java.io.*;
6 import java.util.*;
7
8 /**
9  * This program tests that the database and the JDBC driver are correctly configured.
10 * @version 1.02 2012-06-05
11 * @author Cay Horstmann
12 */
13 public class TestDB
14 {
15     public static void main(String args[]) throws IOException
16     {
17         try
18         {
19             runTest();
20         }
21         catch (SQLException ex)
22         {
23             for (Throwable t : ex)
24                 t.printStackTrace();
25         }
26     }
27
28 /**
29  * Runs a test by creating a table, adding a value, showing the table contents, and removing
30  * the table.
31  */
32     public static void runTest() throws SQLException, IOException
33     {
```

```
34     try (Connection conn = getConnection();
35           Statement stat = conn.createStatement())
36     {
37         stat.executeUpdate("CREATE TABLE Greetings (Message CHAR(20))");
38         stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, World!')");
39
40         try (ResultSet result = stat.executeQuery("SELECT * FROM Greetings"))
41         {
42             if (result.next())
43                 System.out.println(result.getString(1));
44         }
45         stat.executeUpdate("DROP TABLE Greetings");
46     }
47 }
48
49 /**
50 * Gets a connection from the properties specified in the file database.properties.
51 * @return the database connection
52 */
53 public static Connection getConnection() throws SQLException, IOException
54 {
55     Properties props = new Properties();
56     try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
57     {
58         props.load(in);
59     }
60     String drivers = props.getProperty("jdbc.drivers");
61     if (drivers != null) System.setProperty("jdbc.drivers", drivers);
62     String url = props.getProperty("jdbc.url");
63     String username = props.getProperty("jdbc.username");
64     String password = props.getProperty("jdbc.password");
65
66     return DriverManager.getConnection(url, username, password);
67 }
68 }
```

java.sql.DriverManager 1.1

- static Connection getConnection(String url, String user, String password)
establishes a connection to the given database and returns a Connection object.

5.4 Working with JDBC Statements

In the following sections, you will see how to use the JDBC Statement to execute SQL statements, obtain results, and deal with errors. Then we show you a simple program for populating a database.

5.4.1 Executing SQL Statements

To execute a SQL statement, you first create a `Statement` object. To create `Statement` objects, use the `Connection` object that you obtained from the call to `DriverManager.getConnection`.

```
Statement stat = conn.createStatement();
```

Next, place the statement that you want to execute into a string, for example

```
String command = "UPDATE Books"
    + " SET Price = Price - 5.00"
    + " WHERE Title NOT LIKE '%Introduction%'";
```

Then call the `executeUpdate` method of the `Statement` interface:

```
stat.executeUpdate(command);
```

The `executeUpdate` method returns a count of the rows that were affected by the SQL statement, or zero for statements that do not return a row count. For example, the call to `executeUpdate` in the preceding example returns the number of rows where the price was lowered by \$5.00.

The `executeUpdate` method can execute actions such as `INSERT`, `UPDATE`, and `DELETE`, as well as data definition statements such as `CREATE TABLE` and `DROP TABLE`. However, you need to use the `executeQuery` method to execute `SELECT` queries. There is also a catch-all `execute` statement to execute arbitrary SQL statements. It's commonly used only for queries that a user supplies interactively.

When you execute a query, you are interested in the result. The `executeQuery` object returns an object of type `ResultSet` that you can use to walk through the result one row at a time.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

The basic loop for analyzing a result set looks like this:

```
while (rs.next())
{
    look at a row of the result set
}
```



CAUTION: The iteration protocol of the `ResultSet` interface is subtly different from the protocol of the `java.util.Iterator` interface. Here, the iterator is initialized to a position *before* the first row. You must call the `next` method once to move the iterator to the first row. Also, there is no `hasNext` method; keep calling `next` until it returns `false`.

The order of the rows in a result set is completely arbitrary. Unless you specifically ordered the result with an ORDER BY clause, you should not attach any significance to the row order.

When inspecting an individual row, you will want to know the contents of the fields. A large number of accessor methods give you this information.

```
String isbn = rs.getString(1);  
double price = rs.getDouble("Price");
```

There are accessors for various *types*, such as `getString` and `getDouble`. Each accessor has two forms: One takes a numeric argument and the other takes a string argument. When you supply a numeric argument, you refer to the column with that number. For example, `rs.getString(1)` returns the value of the first column in the current row.



CAUTION: Unlike array indexes, database column numbers start at 1.

When you supply a string argument, you refer to the column in the result set with that name. For example, `rs.getDouble("Price")` returns the value of the column with label `Price`. Using the numeric argument is a bit more efficient, but the string arguments make the code easier to read and maintain.

Each `get` method makes reasonable type conversions when the type of the method doesn't match the type of the column. For example, the call `rs.getString("Price")` converts the floating-point value of the `Price` column to a string.

java.sql.Connection 1.1

- `Statement createStatement()`

creates a `Statement` object that can be used to execute SQL queries and updates without parameters.

- `void close()`

immediately closes the current connection and the JDBC resources that it created.

java.sql.Statement 1.1

- `ResultSet executeQuery(String sqlQuery)`
executes the SQL statement given in the string and returns a `ResultSet` object to view the query result.
- `int executeUpdate(String sqlStatement)`
- `long executeLargeUpdate(String sqlStatement) 8`
executes the SQL `INSERT`, `UPDATE`, or `DELETE` statement specified by the string. Also executes Data Definition Language (DDL) statements such as `CREATE TABLE`. Returns the number of rows affected, or 0 for a statement without an update count.
- `boolean execute(String sqlStatement)`
executes the SQL statement specified by the string. Multiple result sets and update counts may be produced. Returns `true` if the first result is a result set, `false` otherwise. Call `getResultSet` or `getUpdateCount` to retrieve the first result. See Section 5.5.4, “Multiple Results,” on p. 319 for details on processing multiple results.
- `ResultSet getResultSet()`
returns the result set of the preceding query statement, or `null` if the preceding statement did not have a result set. Call this method only once per executed statement.
- `int getUpdateCount()`
- `long getLargeUpdateCount() 8`
returns the number of rows affected by the preceding update statement, or -1 if the preceding statement was a statement without an update count. Call this method only once per executed statement.
- `void close()`
closes this statement object and its associated result set.
- `boolean isClosed() 6`
returns `true` if this statement is closed.
- `void closeOnCompletion() 7`
causes this statement to be closed once all of its result sets have been closed.

java.sql.ResultSet 1.1

- `boolean next()`
makes the current row in the result set move forward by one. Returns `false` after the last row. Note that you must call this method to advance to the first row.

(Continues)

java.sql.ResultSet 1.1 (Continued)

- `Xxx getXxx(int columnNumber)`
- `Xxx getXxx(String columnLabel)`
(`Xxx` is a type such as `int`, `double`, `String`, `Date`, etc.)
- `<T> T getObject(int columnIndex, Class<T> type)` 7
- `<T> T getObject(String columnLabel, Class<T> type)` 7
- `void updateObject(int columnIndex, Object x, SQLType targetSqlType)` 8
- `void updateObject(String columnLabel, Object x, SQLType targetSqlType)` 8
returns or updates the value of the column with the given column index or label, converted to the specified type. The column label is the label specified in the SQL AS clause or the column name if AS is not used.
- `int findColumn(String columnName)`
gives the column index associated with a column name.
- `void close()`
immediately closes the current result set.
- `boolean isClosed()` 6
returns true if this statement is closed.

5.4.2 Managing Connections, Statements, and Result Sets

Every `Connection` object can create one or more `Statement` objects. You can use the same `Statement` object for multiple unrelated commands and queries. However, a statement has *at most one* open result set. If you issue multiple queries whose results you analyze concurrently, you need multiple `Statement` objects.

Be forewarned, though, that at least one commonly used database (Microsoft SQL Server) has a JDBC driver that allows only one active statement at a time. Use the `getMaxStatements` method of the `DatabaseMetaData` interface to find out the number of concurrently open statements that your JDBC driver supports.

This sounds restrictive, but in practice, you should probably not fuss with multiple concurrent result sets. If the result sets are related, you should be able to issue a combined query and analyze a single result. It is much more efficient to let the database combine queries than it is for a Java program to iterate through multiple result sets.

When you are done using a `ResultSet`, `Statement`, or `Connection`, you should call the `close` method immediately. These objects use large data structures that draw on the finite resources of the database server.

The `close` method of a `Statement` object automatically closes the associated result set if the statement has an open result set. Similarly, the `close` method of the `Connection` class closes all statements of the connection.

Conversely, as of Java SE 7, you can call the `closeOnCompletion` method on a `Statement`, and it will close automatically as soon as all its result sets have closed.

If your connections are short-lived, you don't have to worry about closing statements and result sets. To make absolutely sure that a connection object cannot possibly remain open, use a try-with-resources statement:

```
try (Connection conn = . . .)
{
    Statement stat = conn.createStatement();
    ResultSet result = stat.executeQuery(queryString);
    process query result
}
```



TIP: Use the try-with-resources block just to close the connection, and use a separate try/catch block to handle exceptions. Separating the try blocks makes your code easier to read and maintain.

5.4.3 Analyzing SQL Exceptions

Each `SQLException` has a chain of `SQLException` objects that are retrieved with the `getNextException` method. This exception chain is in addition to the “cause” chain of `Throwable` objects that every exception has. (See Volume I, Chapter 11 for details about Java exceptions.) One would need two nested loops to fully enumerate all these exceptions. Fortunately, Java SE 6 enhanced the `SQLException` class to implement the `Iterable<Throwable>` interface. The `iterator()` method yields an `Iterator<Throwable>` that iterates through both chains: starts by going through the cause chain of the first `SQLException`, then moves on to the next `SQLException`, and so on. You can simply use an enhanced for loop:

```
for (Throwable t : sqlException)
{
    do something with t
}
```

You can call `getSQLState` and `getErrorCode` on a `SQLException` to analyze it further. The first method yields a string that is standardized by either X/Open or SQL:2003. (Call the `getSQLStateType` method of the `DatabaseMetaData` interface to find out which standard is used by your driver.) The error code is vendor-specific.

The SQL exceptions are organized into an inheritance tree (shown in Figure 5.5). This allows you to catch specific error types in a vendor-independent way.

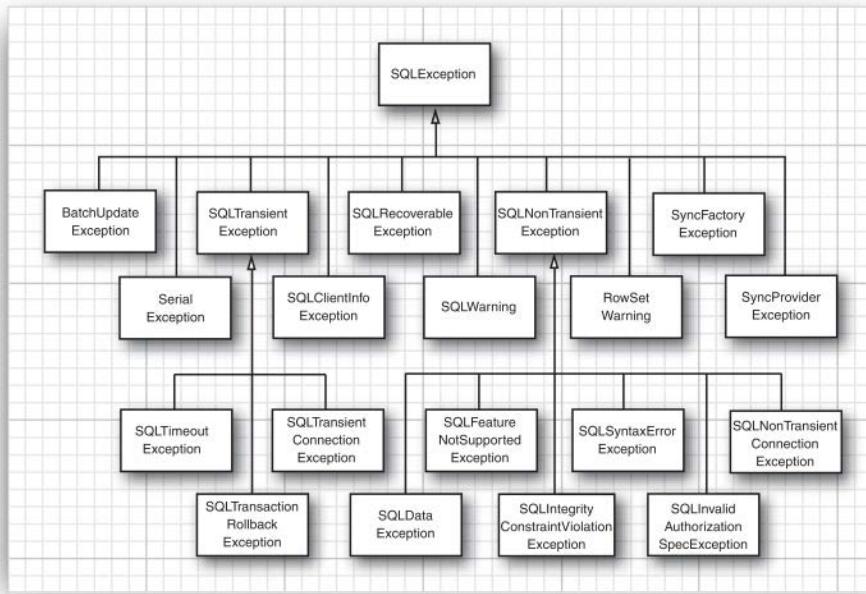


Figure 5.5 SQL exception types

In addition, the database driver can report nonfatal conditions as warnings. You can retrieve warnings from connections, statements, and result sets. The `SQLWarning` class is a subclass of `SQLException` (even though a `SQLWarning` is not thrown as an exception). Call `getSQLState` and `getErrorCode` to get further information about the warnings. Similar to SQL exceptions, warnings are chained. To retrieve all warnings, use this loop:

```

SQLWarning w = stat.getWarning();
while (w != null)
{
    do something with w
    w = w.nextWarning();
}
    
```

The `DataTruncation` subclass of `SQLWarning` is used when data are read from the database and unexpectedly truncated. If data truncation happens in an update statement, a `DataTruncation` is thrown as an exception.

java.sql.SQLException 1.1

- `SQLException getNextException()`
gets the next SQL exception chained to this one, or `null` at the end of the chain.
- `Iterator<Throwable> iterator() 6`
gets an iterator that yields the chained SQL exceptions and their causes.
- `String getSQLState()`
gets the “SQL state”—a standardized error code.
- `int getErrorCode()`
gets the vendor-specific error code.

java.sql.SQLWarning 1.1

- `SQLWarning getNextWarning()`
returns the next warning chained to this one, or `null` at the end of the chain.

java.sql.Connection 1.1***java.sql.Statement 1.1******java.sql.ResultSet 1.1***

- `SQLWarning getWarnings()`
returns the first of the pending warnings, or `null` if no warnings are pending.

java.sql.DataTruncation 1.1

- `boolean getParameter()`
returns `true` if the data truncation applies to a parameter, `false` if it applies to a column.
- `int getIndex()`
returns the index of the truncated parameter or column.
- `int getDataSize()`
returns the number of bytes that should have been transferred, or `-1` if the value is unknown.

(Continues)

java.sql.DataTruncation 1.1 (Continued)

- int getTransferSize()

returns the number of bytes that were actually transferred, or -1 if the value is unknown.

5.4.4 Populating a Database

We are now ready to write our first real JDBC program. Of course it would be nice to try some of the fancy queries that we discussed earlier, but we have a problem: Right now, there are no data in the database. We need to populate the database, and there is a simple way of doing that with a set of SQL instructions to create tables and insert data into them. Most database programs can process a set of SQL instructions from a text file, but there are pesky differences about statement terminators and other syntactical issues.

For that reason, we will use JDBC to create a simple program that reads a file with SQL instructions, one instruction per line, and executes them.

Specifically, the program reads data from a text file in a format such as

```
CREATE TABLE Publishers (Publisher_Id CHAR(6), Name CHAR(30), URL CHAR(80));
INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley', 'www.aw-bc.com');
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons', 'www.wiley.com');
...

```

Listing 5.2 contains the code for the program that reads the SQL statement file and executes the statements. You don't have to read through the code; we merely provide the program so that you can populate your database and run the examples in the remainder of this chapter.

Make sure that your database server is running, and run the program as follows:

```
java -classpath driverPath:. exec.ExecSQL Books.sql
java -classpath driverPath:. exec.ExecSQL Authors.sql
java -classpath driverPath:. exec.ExecSQL Publishers.sql
java -classpath driverPath:. exec.ExecSQL BooksAuthors.sql
```

Before running the program, check that the file `database.properties` is set up properly for your environment (see Section 5.3.5, “Connecting to the Database,” on p. 294).

NOTE: Your database may also have a utility to read SQL files directly. For example, with Derby, you can run

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties Books.sql
```

(The `ij.properties` file is described in Section 5.3.3, “Starting the Database,” on p. 293.)

In the data format for the `ExecSQL` command, we allow an optional semicolon at the end of each line because most database utilities expect this format.

The following steps briefly describe the `ExecSQL` program:

1. Connect to the database. The `getConnection` method reads the properties in the file `database.properties` and adds the `jdbc.drivers` property to the system properties. The driver manager uses the `jdbc.drivers` property to load the appropriate database driver. The `getConnection` method uses the `jdbc.url`, `jdbc.username`, and `jdbc.password` properties to open the database connection.
2. Open the file with the SQL statements. If no file name was supplied, prompt the user to enter the statements on the console.
3. Execute each statement with the generic `execute` method. If it returns `true`, the statement had a result set. The four SQL files that we provide for the book database all end in a `SELECT *` statement so that you can see that the data were successfully inserted.
4. If there was a result set, print out the result. Since this is a generic result set, we need to use metadata to find out how many columns the result has. For more information, see Section 5.8, “Metadata,” on p. 333.
5. If there is any SQL exception, print the exception and any chained exceptions that may be contained in it.
6. Close the connection to the database.

Listing 5.2 shows the code for the program.

Listing 5.2 exec/ExecSQL.java

```
1 package exec;  
2  
3 import java.io.*;  
4 import java.nio.file.*;  
5 import java.util.*;  
6 import java.sql.*;
```

```
7
8 /**
9  * Executes all SQL statements in a file. Call this program as <br>
10 * java -classpath driverPath:. ExecSQL commandFile
11 *
12 * @version 1.32 2016-04-27
13 * @author Cay Horstmann
14 */
15 class ExecSQL
16 {
17     public static void main(String args[]) throws IOException
18     {
19         try (Scanner in = args.length == 0 ? new Scanner(System.in)
20              : new Scanner(Paths.get(args[0]), "UTF-8"))
21         {
22             try (Connection conn = getConnection();
23                  Statement stat = conn.createStatement())
24             {
25                 while (true)
26                 {
27                     if (args.length == 0) System.out.println("Enter command or EXIT to exit:");
28
29                     if (!in.hasNextLine()) return;
30
31                     String line = in.nextLine().trim();
32                     if (line.equalsIgnoreCase("EXIT")) return;
33                     if (line.endsWith(";")) // remove trailing semicolon
34                     {
35                         line = line.substring(0, line.length() - 1);
36                     }
37                     try
38                     {
39                         boolean isResult = stat.execute(line);
40                         if (isResult)
41                         {
42                             try (ResultSet rs = stat.getResultSet())
43                             {
44                                 showResultSet(rs);
45                             }
46                         }
47                         else
48                         {
49                             int updateCount = stat.getUpdateCount();
50                             System.out.println(updateCount + " rows updated");
51                         }
52                     }
53                 }
54             }
55         }
56     }
57 }
```

(Continues)

Listing 5.2 (Continued)

```
53         catch (SQLException ex)
54     {
55         for (Throwable e : ex)
56             e.printStackTrace();
57     }
58 }
59 }
60 }
61 catch (SQLException e)
62 {
63     for (Throwable t : e)
64         t.printStackTrace();
65 }
66 }
67 /**
68 * Gets a connection from the properties specified in the file database.properties.
69 * @return the database connection
70 */
71 public static Connection getConnection() throws SQLException, IOException
72 {
73     Properties props = new Properties();
74     try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
75     {
76         props.load(in);
77     }
78
79     String drivers = props.getProperty("jdbc.drivers");
80     if (drivers != null) System.setProperty("jdbc.drivers", drivers);
81
82     String url = props.getProperty("jdbc.url");
83     String username = props.getProperty("jdbc.username");
84     String password = props.getProperty("jdbc.password");
85
86     return DriverManager.getConnection(url, username, password);
87 }
88
89 /**
90 * Prints a result set.
91 * @param result the result set to be printed
92 */
93 public static void showResultSet(ResultSet result) throws SQLException
94 {
95     ResultSetMetaData metaData = result.getMetaData();
96     int columnCount = metaData.getColumnCount();
```

```
99     for (int i = 1; i <= columnCount; i++)
100    {
101        if (i > 1) System.out.print(", ");
102        System.out.print(metaData.getColumnLabel(i));
103    }
104    System.out.println();
105
106    while (result.next())
107    {
108        for (int i = 1; i <= columnCount; i++)
109        {
110            if (i > 1) System.out.print(", ");
111            System.out.print(result.getString(i));
112        }
113        System.out.println();
114    }
115}
116}
```

5.5 Query Execution

In this section, we write a program that executes queries against the COREJAVA database. For this program to work, you must have populated the COREJAVA database with tables, as described in the preceding section.

When querying the database, you can select the author and the publisher or leave either of them as Any.

You can also change the data in the database. Select a publisher and type an amount. All prices of that publisher are adjusted by the amount you entered, and the program displays how many rows were changed. After a price change, you might want to run a query to verify the new prices.

5.5.1 Prepared Statements

In this program, we use one new feature, *prepared statements*. Consider the query for all books by a particular publisher, independent of the author. The SQL query is

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = the name from the list box
```

Instead of building a separate query statement every time the user launches such a query, we can *prepare* a query with a host variable and use it many times, each time filling in a different string for the variable. That technique benefits performance. Whenever the database executes a query, it first computes a strategy

of how to do it efficiently. By preparing the query and reusing it, you ensure that the planning step is done only once.

Each host variable in a prepared query is indicated with a ?. If there is more than one variable, you must keep track of the positions of the ? when setting the values. For example, our prepared query becomes

```
String publisherQuery =  
    "SELECT Books.Price, Books.Title" +  
    " FROM Books, Publishers" +  
    " WHERE Books.Publisher_Id = Publishers.Publisher_Id AND Publishers.Name = ?";  
PreparedStatement stat = conn.prepareStatement(publisherQuery);
```

Before executing the prepared statement, you must bind the host variables to actual values with a set method. As with the get methods of the ResultSet interface, there are different set methods for the various types. Here, we want to set a string to a publisher name.

```
stat.setString(1, publisher);
```

The first argument is the position number of the host variable that we want to set. The position 1 denotes the first ?. The second argument is the value that we want to assign to the host variable.

If you reuse a prepared query that you have already executed, all host variables stay bound unless you change them with a set method or call the clearParameters method. That means you only need to call a setXxx method on those host variables that change from one query to the next.

Once all variables have been bound to values, you can execute the prepared statement:

```
ResultSet rs = stat.executeQuery();
```



TIP: Building a query manually, by concatenating strings, is tedious and potentially dangerous. You have to worry about special characters such as quotes, and, if your query involves user input, you have to guard against injection attacks. Therefore, you should use prepared statements whenever your query involves variables.

The price update feature is implemented as an UPDATE statement. Note that we call executeUpdate, not executeQuery, because the UPDATE statement does not return a result set. The return value of executeUpdate is the count of changed rows.

```
int r = stat.executeUpdate();  
System.out.println(r + " rows updated");
```

NOTE: A `PreparedStatement` object becomes invalid after the associated `Connection` object is closed. However, many databases automatically *cache* prepared statements. If the same query is prepared twice, the database simply reuses the query strategy. Therefore, don't worry about the overhead of calling `prepareStatement`.

The following list briefly describes the structure of the example program.

- The author and publisher array lists are populated by running two queries that return all author and publisher names in the database.
- The queries involving authors are complex. A book can have multiple authors, so the `BooksAuthors` table stores the correspondence between authors and books. For example, the book with ISBN 0-201-96426-0 has two authors with codes `DATE` and `DARW`. The `BooksAuthors` table has the rows

```
0-201-96426-0, DATE, 1  
0-201-96426-0, DARW, 2
```

to indicate this fact. The third column lists the order of the authors. (We can't just use the position of the rows in the table. There is no fixed row ordering in a relational table.) Thus, the query has to join the `Books`, `BooksAuthors`, and `Authors` tables to compare the author name with the one selected by the user.

```
SELECT Books.Price, Books.Title FROM Books, BooksAuthors, Authors, Publishers  
WHERE Authors.Author_Id = BooksAuthors.Author_Id AND BooksAuthors.ISBN = Books.ISBN  
AND Books.Publisher_Id = Publishers.Publisher_Id AND Authors.Name = ? AND Publishers.Name = ?
```



TIP: Some Java programmers avoid complex SQL statements such as this one. A surprisingly common, but very inefficient, workaround is to write lots of Java code that iterates through multiple result sets. But the database is a *lot* better at executing query code than a Java program can be—that's the core competency of a database. A rule of thumb: If you can do it in SQL, don't do it in Java.

- The `changePrices` method executes an `UPDATE` statement. Note that the `WHERE` clause of the `UPDATE` statement needs the publisher *code* and we know only the publisher *name*. This problem is solved with a nested subquery:

```
UPDATE Books  
SET Price = Price + ?  
WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishers WHERE Name = ?)
```

Listing 5.3 is the complete program code.

Listing 5.3 query/QueryTest.java

```
1 package query;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.sql.*;
6 import java.util.*;
7
8 /**
9  * This program demonstrates several complex database queries.
10 * @version 1.30 2012-06-05
11 * @author Cay Horstmann
12 */
13 public class QueryTest
14 {
15     private static final String allQuery = "SELECT Books.Price, Books.Title FROM Books";
16
17     private static final String authorPublisherQuery = "SELECT Books.Price, Books.Title"
18         + " FROM Books, BooksAuthors, Authors, Publishers"
19         + " WHERE Authors.Author_Id = BooksAuthors.Author_Id AND BooksAuthors.ISBN = Books.ISBN"
20         + " AND Books.Publisher_Id = Publishers.Publisher_Id AND Authors.Name = ?"
21         + " AND Publishers.Name = ?";
22
23     private static final String authorQuery
24         = "SELECT Books.Price, Books.Title FROM Books, BooksAuthors, Authors"
25         + " WHERE Authors.Author_Id = BooksAuthors.Author_Id AND BooksAuthors.ISBN = Books.ISBN"
26         + " AND Authors.Name = ?";
27
28     private static final String publisherQuery
29         = "SELECT Books.Price, Books.Title FROM Books, Publishers"
30         + " WHERE Books.Publisher_Id = Publishers.Publisher_Id AND Publishers.Name = ?";
31
32
33     private static final String priceUpdate = "UPDATE Books " + "SET Price = Price + ? "
34         + " WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishers WHERE Name = ?)";
35
36     private static Scanner in;
37     private static ArrayList<String> authors = new ArrayList<>();
38     private static ArrayList<String> publishers = new ArrayList<>();
39
40     public static void main(String[] args) throws IOException
41     {
42         try (Connection conn = getConnection())
43         {
44             in = new Scanner(System.in);
45             authors.add("Any");
```

```
46     publishers.add("Any");
47     try (Statement stat = conn.createStatement())
48     {
49         // Fill the authors array list
50         String query = "SELECT Name FROM Authors";
51         try (ResultSet rs = stat.executeQuery(query))
52         {
53             while (rs.next())
54                 authors.add(rs.getString(1));
55         }
56
57         // Fill the publishers array list
58         query = "SELECT Name FROM Publishers";
59         try (ResultSet rs = stat.executeQuery(query))
60         {
61             while (rs.next())
62                 publishers.add(rs.getString(1));
63         }
64     }
65     boolean done = false;
66     while (!done)
67     {
68         System.out.print("Q)uery C)hange prices E)xit: ");
69         String input = in.nextLine().toUpperCase();
70         if (input.equals("Q"))
71             executeQuery(conn);
72         else if (input.equals("C"))
73             changePrices(conn);
74         else
75             done = true;
76     }
77 }
78 catch (SQLException e)
79 {
80     for (Throwable t : e)
81         System.out.println(t.getMessage());
82 }
83
84 /**
85 * Executes the selected query.
86 * @param conn the database connection
87 */
88 private static void executeQuery(Connection conn) throws SQLException
89 {
90     String author = select("Authors:", authors);
91     String publisher = select("Publishers:", publishers);
```

(Continues)

Listing 5.3 (Continued)

```
93     PreparedStatement stat;
94     if (!author.equals("Any") && !publisher.equals("Any"))
95     {
96         stat = conn.prepareStatement(authorPublisherQuery);
97         stat.setString(1, author);
98         stat.setString(2, publisher);
99     }
100    else if (!author.equals("Any") && publisher.equals("Any"))
101    {
102        stat = conn.prepareStatement(authorQuery);
103        stat.setString(1, author);
104    }
105    else if (author.equals("Any") && !publisher.equals("Any"))
106    {
107        stat = conn.prepareStatement(publisherQuery);
108        stat.setString(1, publisher);
109    }
110    else
111        stat = conn.prepareStatement(allQuery);
112
113    try (ResultSet rs = stat.executeQuery())
114    {
115        while (rs.next())
116            System.out.println(rs.getString(1) + ", " + rs.getString(2));
117    }
118 }
119 /**
120 * Executes an update statement to change prices.
121 * @param conn the database connection
122 */
123 public static void changePrices(Connection conn) throws SQLException
124 {
125     String publisher = select("Publishers:", publishers.subList(1, publishers.size()));
126     System.out.print("Change prices by: ");
127     double priceChange = in.nextDouble();
128     PreparedStatement stat = conn.prepareStatement(priceUpdate);
129     stat.setDouble(1, priceChange);
130     stat.setString(2, publisher);
131     int r = stat.executeUpdate();
132     System.out.println(r + " records updated.");
133 }
134 /**
135 * Asks the user to select a string.
```

```
138     * @param prompt the prompt to display
139     * @param options the options from which the user can choose
140     * @return the option that the user chose
141     */
142    public static String select(String prompt, List<String> options)
143    {
144        while (true)
145        {
146            System.out.println(prompt);
147            for (int i = 0; i < options.size(); i++)
148                System.out.printf("%2d) %s%n", i + 1, options.get(i));
149            int sel = in.nextInt();
150            if (sel > 0 && sel <= options.size())
151                return options.get(sel - 1);
152        }
153    }
154 /**
155 * Gets a connection from the properties specified in the file database.properties.
156 * @return the database connection
157 */
158 public static Connection getConnection() throws SQLException, IOException
159 {
160     Properties props = new Properties();
161     try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
162     {
163         props.load(in);
164     }
165
166     String drivers = props.getProperty("jdbc.drivers");
167     if (drivers != null) System.setProperty("jdbc.drivers", drivers);
168     String url = props.getProperty("jdbc.url");
169     String username = props.getProperty("jdbc.username");
170     String password = props.getProperty("jdbc.password");
171
172     return DriverManager.getConnection(url, username, password);
173 }
174 }
175 }
```

java.sql.Connection 1.1

- `PreparedStatement prepareStatement(String sql)`

returns a `PreparedStatement` object containing the precompiled statement. The string `sql` contains a SQL statement with one or more parameter placeholders denoted by `?` characters.

java.sql.PreparedStatement 1.1

- `void setXxx(int n, Xxx x)`
(`Xxx` is a type such as `int`, `double`, `String`, `Date`, etc.)
sets the value of the `n`th parameter to `x`.
- `void clearParameters()`
clears all current parameters in the prepared statement.
- `ResultSet executeQuery()`
executes a prepared SQL query and returns a `ResultSet` object.
- `int executeUpdate()`
executes the prepared SQL `INSERT`, `UPDATE`, or `DELETE` statement represented by the `PreparedStatement` object. Returns the number of rows affected, or 0 for DDL statements such as `CREATE TABLE`.

5.5.2 Reading and Writing LOBs

In addition to numbers, strings, and dates, many databases can store *large objects* (LOBs) such as images or other data. In SQL, binary large objects are called BLOBs, and character large objects are called CLOBs.

To read a LOB, execute a `SELECT` statement and call the `getBlob` or `getBlob` method on the `ResultSet`. You will get an object of type `Blob` or `Clob`. To get the binary data from a `Blob`, call the `getBytes` or `getBinaryStream`. For example, if you have a table with book cover images, you can retrieve an image like this:

```
PreparedStatement stat = conn.prepareStatement("SELECT Cover FROM BookCovers WHERE ISBN=?");  
.  
.  
.  
stat.set(1, isbn);  
try (ResultSet result = stat.executeQuery())  
{  
    if (result.next())  
    {  
        Blob coverBlob = result.getBlob(1);  
        Image coverImage = ImageIO.read(coverBlob.getBinaryStream());  
    }  
}
```

Similarly, if you retrieve a `Clob` object, you can get character data by calling the `getSubString` or `getCharacterStream` method.

To place a LOB into a database, call `createBlob` or `createClob` on your `Connection` object, get an output stream or writer to the LOB, write the data, and store the object in the database. For example, here is how you store an image:

```
Blob coverBlob = connection.createBlob();
int offset = 0;
OutputStream out = coverBlob.setBinaryStream(offset);
ImageIO.write(coverImage, "PNG", out);
PreparedStatement stat = conn.prepareStatement("INSERT INTO Cover VALUES (?, ?)");
stat.set(1, isbn);
stat.set(2, coverBlob);
stat.executeUpdate();
```

java.sql.ResultSet 1.1

- `Blob getBlob(int columnIndex)` 1.2
 - `Blob getBlob(String columnLabel)` 1.2
 - `Clob getClob(int columnIndex)` 1.2
 - `Clob getClob(String columnLabel)` 1.2
- gets the BLOB or CLOB at the given column.

java.sql.Blob 1.2

- `long length()`
gets the length of this BLOB.
- `byte[] getBytes(long startPosition, long length)`
gets the data in the given range from this BLOB.
- `InputStream getBinaryStream()`
- `InputStream getBinaryStream(long startPosition, long length)`
returns a stream to read the data from this BLOB or from the given range.
- `OutputStream setBinaryStream(long startPosition)` 1.4
returns an output stream for writing into this BLOB, starting at the given position.

java.sql.Clob 1.4

- `long length()`
gets the number of characters of this CLOB.
- `String getSubString(long startPosition, long length)`
gets the characters in the given range from this CLOB.

(Continues)

java.sql.Clob 1.4 (Continued)

- Reader getCharacterStream()
- Reader getCharacterStream(long startPosition, long length)
returns a reader (not a stream) to read the characters from this CLOB or from the given range.
- Writer setCharacterStream(long startPosition) **1.4**
returns a writer (not a stream) for writing into this CLOB, starting at the given position.

java.sql.Connection 1.1

- Blob createBlob() **6**
- Clob createClob() **6**
creates an empty BLOB or CLOB.

5.5.3 SQL Escapes

The “escape” syntax features are commonly supported by databases but use database-specific syntax variations. It is the job of the JDBC driver to translate the escape syntax to the syntax of a particular database.

Escapes are provided for the following features:

- Date and time literals
- Calling scalar functions
- Calling stored procedures
- Outer joins
- The escape character in LIKE clauses

Date and time literals vary widely among databases. To embed a date or time literal, specify the value in ISO 8601 format (www.cl.cam.ac.uk/~mgk25/iso-time.html). The driver will then translate it into the native format. Use d, t, ts for DATE, TIME, or TIMESTAMP values:

```
{d '2008-01-24'}  
{t '23:59:59'}  
{ts '2008-01-24 23:59:59.999'}
```

A *scalar function* is a function that returns a single value. Many functions are widely available in databases, but with varying names. The JDBC specification provides standard names and translates them into the database-specific names. To call a function, embed the standard function name and arguments like this:

```
{fn left(?, 20)}  
{fn user()}
```

You can find a complete list of supported function names in the JDBC specification.

A *stored procedure* is a procedure that executes in the database, written in a database-specific language. To call a stored procedure, use the `call` escape. You need not supply parentheses if the procedure has no parameters. Use `=` to capture a return value:

```
{call PROC1(?, ?)}  
{call PROC2}  
{call ? = PROC3(?)}
```

An *outer join* of two tables does not require that the rows of each table match according to the join condition. For example, the query

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers ON Books.Publisher_Id = Publisher.Publisher_Id}
```

contains books for which `Publisher_Id` has no match in the `Publishers` table, with `NULL` values to indicate that no match exists. You would need a `RIGHT OUTER JOIN` to include publishers without matching books, or a `FULL OUTER JOIN` to return both. The escape syntax is needed because not all databases use a standard notation for these joins.

Finally, the `_` and `%` characters have special meanings in a `LIKE` clause—to match a single character or a sequence of characters. There is no standard way to use them literally. If you want to match all strings containing a `_`, use this construct:

```
... WHERE ? LIKE %!_% {escape '!'}
```

Here we define `!` as the escape character. The combination `!_` denotes a literal underscore.

5.5.4 Multiple Results

It is possible for a query to return multiple results. This can happen when executing a stored procedure, or with databases that also allow submission of multiple `SELECT` statements in a single query. Here is how you retrieve all result sets:

1. Use the `execute` method to execute the SQL statement.
2. Retrieve the first result or update count.

3. Repeatedly call the `getMoreResults` method to move on to the next result set.
4. Finish when there are no more result sets or update counts.

The `execute` and `getMoreResults` methods return `true` if the next item in the chain is a result set. The `getUpdateCount` method returns `-1` if the next item in the chain is not an update count.

The following loop traverses all results:

```
boolean isResult = stat.execute(command);
boolean done = false;
while (!done)
{
    if (isResult)
    {
        ResultSet result = stat.getResultSet();
        do something with result
    }
    else
    {
        int updateCount = stat.getUpdateCount();
        if (updateCount >= 0)
            do something with updateCount
        else
            done = true;
    }
    if (!done) isResult = stat.getMoreResults();
}
```

`java.sql.Statement` 1.1

- `boolean getMoreResults()`
- `boolean getMoreResults(int current)` **6**

gets the next result for this statement. The `current` parameter is one of `CLOSE_CURRENT_RESULT` (default), `KEEP_CURRENT_RESULT`, or `CLOSE_ALL_RESULTS`. Returns true if the next result exists and is a result set.

5.5.5 Retrieving Autogenerated Keys

Most databases support some mechanism for autonumbering rows in a database. Unfortunately, the mechanisms differ widely among vendors. These automatic numbers are often used as primary keys. Although JDBC doesn't offer a

vendor-independent solution for generating keys, it does provide an efficient way of retrieving them. When you insert a new row into a table and a key is automatically generated, you can retrieve it with the following code:

```
stat.executeUpdate(insertStatement, Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stat.getGeneratedKeys();
if (rs.next())
{
    int key = rs.getInt(1);
    ...
}
```

java.sql.Statement 1.1

- `boolean execute(String statement, int autogenerated)` 1.4
- `int executeUpdate(String statement, int autogenerated)` 1.4

executes the given SQL statement, as previously described. If `autogenerated` is set to `Statement.RETURN_GENERATED_KEYS` and the statement is an `INSERT` statement, the first column contains the autogenerated key.

5.6 Scrollable and Updatable Result Sets

As you have seen, the `next` method of the `ResultSet` interface iterates over the rows in a result set. That is certainly adequate for a program that needs to analyze the data. However, consider a visual data display that shows a table or query results (such as Figure 5.4 on p. 288). You usually want the user to be able to move both forward and backward in the result set. In a *scrollable* result, you can move forward and backward through a result set and even jump to any position.

Furthermore, once users see the contents of a result set displayed, they may be tempted to edit it. In an *updatable* result set, you can programmatically update entries so that the database is automatically updated. We discuss these capabilities in the following sections.

5.6.1 Scrollable Result Sets

By default, result sets are not scrollable or updatable. To obtain scrollable result sets from your queries, you must obtain a different `Statement` object with the method

```
Statement stat = conn.createStatement(type, concurrency);
```

For a prepared statement, use the call

```
PreparedStatement stat = conn.prepareStatement(command, type, concurrency);
```

The possible values of `type` and `concurrency` are listed in Tables 5.6 and 5.7. You have the following choices:

- Do you want the result set to be scrollable? If not, use `ResultSet.TYPE_FORWARD_ONLY`.
- If the result set is scrollable, do you want it to reflect changes in the database that occurred after the query that yielded it? (In our discussion, we assume the `ResultSet.TYPE_SCROLL_INSENSITIVE` setting for scrollable result sets. This assumes that the result set does not “sense” database changes that occurred after execution of the query.)
- Do you want to be able to update the database by editing the result set? (See the next section for details.)

Table 5.6 ResultSet Type Values

Value	Explanation
<code>TYPE_FORWARD_ONLY</code>	The result set is not scrollable (default).
<code>TYPE_SCROLL_INSENSITIVE</code>	The result set is scrollable but not sensitive to database changes.
<code>TYPE_SCROLL_SENSITIVE</code>	The result set is scrollable and sensitive to database changes.

Table 5.7 ResultSet Concurrency Values

Value	Explanation
<code>CONCUR_READ_ONLY</code>	The result set cannot be used to update the database (default).
<code>CONCUR_UPDATABLE</code>	The result set can be used to update the database.

For example, if you simply want to be able to scroll through a result set but don’t want to edit its data, use

```
Statement stat = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

All result sets that are returned by method calls

```
ResultSet rs = stat.executeQuery(query);
```

are now scrollable. A scrollable result set has a *cursor* that indicates the current position.

NOTE: Not all database drivers support scrollable or updatable result sets. (The `supportsResultSetType` and `supportsResultSetConcurrency` methods of the `DatabaseMetaData` interface will tell you which types and concurrency modes are supported by a particular database using a particular driver.) Even if a database supports all result set modes, a particular query might not be able to yield a result set with all the properties that you requested. (For example, the result set of a complex query might not be updatable.) In that case, the `executeQuery` method returns a `ResultSet` of lesser capabilities and adds a `SQLWarning` to the connection object. (Section 5.4.3, “Analyzing SQL Exceptions,” on p. 302 shows how to retrieve the warning.) Alternatively, you can use the `getType` and `getConcurrency` methods of the `ResultSet` interface to find out what mode a result set actually has. If you do not check the result set capabilities and issue an unsupported operation, such as `previous` on a result set that is not scrollable, the operation will throw a `SQLException`.

Scrolling is very simple. Use

```
if (rs.previous()) . . .
```

to scroll backward. The method returns `true` if the cursor is positioned on an actual row, or `false` if it is now positioned before the first row.

You can move the cursor backward or forward by a number of rows with the call

```
rs.relative(n);
```

If `n` is positive, the cursor moves forward. If `n` is negative, it moves backward. If `n` is zero, the call has no effect. If you attempt to move the cursor outside the current set of rows, it is set to point either after the last row or before the first row, depending on the sign of `n`. Then, the method returns `false` and the cursor does not move. The method returns `true` if the cursor is positioned on an actual row.

Alternatively, you can set the cursor to a particular row number:

```
rs.absolute(n);
```

To get the current row number, call

```
int currentRow = rs.getRow();
```

The first row in the result set has number 1. If the return value is 0, the cursor is not currently on a row—it is either before the first row or after the last row.

The convenience methods `first`, `last`, `beforeFirst`, and `afterLast` move the cursor to the first, to the last, before the first, or after the last position.

Finally, the methods `isFirst`, `isLast`, `isBeforeFirst`, and `isAfterLast` test whether the cursor is at one of these special positions.

Using a scrollable result set is very simple. The hard work of caching the query data is carried out behind the scenes by the database driver.

5.6.2 Updatable Result Sets

If you want to edit the data in the result set and have the changes automatically reflected in the database, create an updatable result set. Updatable result sets don't have to be scrollable, but if you present data to a user for editing, you usually want to allow scrolling as well.

To obtain updatable result sets, create a statement as follows:

```
Statement stat = conn.createStatement()  
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

The result sets returned by a call to `executeQuery` are then updatable.

NOTE: Not all queries return updatable result sets. If your query is a join that involves multiple tables, the result might not be updatable. However, if your query involves only a single table or if it joins multiple tables by their primary keys, you should expect the result set to be updatable. Call the `getConcurrency` method of the `ResultSet` interface to find out for sure.

For example, suppose you want to raise the prices of some books, but you don't have a simple criterion for issuing an `UPDATE` statement. Then, you can iterate through all books and update prices based on arbitrary conditions.

```
String query = "SELECT * FROM Books";  
ResultSet rs = stat.executeQuery(query);  
while (rs.next())  
{  
    if (...)  
    {  
        double increase = ...;  
        double price = rs.getDouble("Price");  
        rs.updateDouble("Price", price + increase);  
        rs.updateRow(); // make sure to call updateRow after updating fields  
    }  
}
```

There are `updateXxx` methods for all data types that correspond to SQL types, such as `updateDouble`, `updateString`, and so on; specify the name or the number of the column (as with the `getXxx` methods), then the new value for the field.

NOTE: If you use the `updateXxx` method whose first parameter is the column number, be aware that this is the column number in the *result set*. It could well be different from the column number in the database.

The `updateXxx` method changes only the row values, not the database. When you are done with the field updates in a row, you must call the `updateRow` method. That method sends all updates in the current row to the database. If you move the cursor to another row without calling `updateRow`, this row's updates are discarded from the row set and never communicated to the database. You can also call the `cancelRowUpdates` method to cancel the updates to the current row.

The preceding example shows how to modify an existing row. If you want to add a new row to the database, first use the `moveToInsertRow` method to move the cursor to a special position, called the *insert row*. Then, build up a new row in the insert row position by issuing `updateXxx` instructions. When you are done, call the `insertRow` method to deliver the new row to the database. When you are done inserting, call `moveToCurrentRow` to move the cursor back to the position before the call to `moveToInsertRow`. Here is an example:

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

Note that you cannot influence *where* the new data is added in the result set or the database.

If you don't specify a column value in the insert row, it is set to a SQL NULL. However, if the column has a NOT NULL constraint, an exception is thrown and the row is not inserted.

Finally, you can delete the row under the cursor:

```
rs.deleteRow();
```

The `deleteRow` method immediately removes the row from both the result set and the database.

The `updateRow`, `insertRow`, and `deleteRow` methods of the `ResultSet` interface give you the same power as executing UPDATE, INSERT, and DELETE SQL statements. However, Java programmers might find it more natural to manipulate the database contents through result sets than by constructing SQL statements.



CAUTION: If you are not careful, you can write staggeringly inefficient code with updatable result sets. It is *much* more efficient to execute an UPDATE statement than to make a query and iterate through the result, changing data along the way. Updatable result sets make sense for interactive programs in which a user can make arbitrary changes, but for most programmatic changes, a SQL UPDATE is more appropriate.

NOTE: JDBC 2 delivered further enhancements to result sets, such as the capability to update a result set with the most recent data if the data have been modified by another concurrent database connection. JDBC 3 added yet another refinement, specifying the behavior of result sets when a transaction is committed. However, these advanced features are outside the scope of this introductory chapter. We refer you to the *JDBC™ API Tutorial and Reference, Third Edition*, by Maydene Fisher, Jon Ellis, and Jonathan Bruce (Addison-Wesley, 2003) and the JDBC specification for more information.

java.sql.Connection 1.1

- Statement createStatement(int type, int concurrency) 1.2
- PreparedStatement prepareStatement(String command, int type, int concurrency) 1.2

creates a statement or prepared statement that yields result sets with the given type and concurrency.

<i>Parameters:</i>	command	The command to prepare
	type	One of the constants TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE of the ResultSet interface
	concurrency	One of the constants CONCUR_READ_ONLY or CONCUR_UPDATABLE of the ResultSet interface

java.sql.ResultSet 1.1

- int getType() 1.2

returns the type of this result set—one of TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE.

(Continues)

java.sql.ResultSet 1.1 (Continued)

- **int getConcurrency() 1.2**

returns the concurrency setting of this result set—one of CONCUR_READ_ONLY or CONCUR_UPDATABLE.

- **boolean previous() 1.2**

moves the cursor to the preceding row. Returns true if the cursor is positioned on a row, or false if the cursor is positioned before the first row.

- **int getRow() 1.2**

gets the number of the current row. Rows are numbered starting with 1.

- **boolean absolute(int r) 1.2**

moves the cursor to row r. Returns true if the cursor is positioned on a row.

- **boolean relative(int d) 1.2**

moves the cursor by d rows. If d is negative, the cursor is moved backward. Returns true if the cursor is positioned on a row.

- **boolean first() 1.2**

- **boolean last() 1.2**

moves the cursor to the first or last row. Returns true if the cursor is positioned on a row.

- **void beforeFirst() 1.2**

- **void afterLast() 1.2**

moves the cursor before the first or after the last row.

- **boolean isFirst() 1.2**

- **boolean isLast() 1.2**

tests whether the cursor is at the first or last row.

- **boolean isBeforeFirst() 1.2**

- **boolean isAfterLast() 1.2**

tests whether the cursor is before the first or after the last row.

- **void moveToInsertRow() 1.2**

moves the cursor to the insert row. The insert row is a special row for inserting new data with the updateXxx and insertRow methods.

- **void moveToCurrentRow() 1.2**

moves the cursor back from the insert row to the row that it occupied when the moveToInsertRow method was called.

(Continues)

java.sql.ResultSet 1.1 (Continued)

- **void insertRow() 1.2**
inserts the contents of the insert row into the database and the result set.
- **void deleteRow() 1.2**
deletes the current row from the database and the result set.
- **void updateXxx(int column, Xxx data) 1.2**
- **void updateXxx(String columnName, Xxx data) 1.2**
(*Xxx* is a type such as *int*, *double*, *String*, *Date*, etc.)
updates a field in the current row of the result set.
- **void updateRow() 1.2**
sends the current row updates to the database.
- **void cancelRowUpdates() 1.2**
cancels the current row updates.

java.sql.DatabaseMetaData 1.1

- **boolean supportsResultSetType(int type) 1.2**
returns true if the database can support result sets of the given type; type is one of the constants *TYPE_FORWARD_ONLY*, *TYPE_SCROLL_INSENSITIVE*, or *TYPE_SCROLL_SENSITIVE* of the *ResultSet* interface.
- **boolean supportsResultSetConcurrency(int type, int concurrency) 1.2**
returns true if the database can support result sets of the given combination of type and concurrency.

<i>Parameters:</i>	<i>type</i>	One of the constants <i>TYPE_FORWARD_ONLY</i> , <i>TYPE_SCROLL_INSENSITIVE</i> , or <i>TYPE_SCROLL_SENSITIVE</i> of the <i>ResultSet</i> interface
	<i>concurrency</i>	One of the constants <i>CONCUR_READ_ONLY</i> or <i>CONCUR_UPDATABLE</i> of the <i>ResultSet</i> interface

5.7 Row Sets

Scrollable result sets are powerful, but they have a major drawback. You need to keep the database connection open during the entire user interaction. However, a user can walk away from the computer for a long time, leaving the connection occupied. That is not good—database connections are scarce resources. In this

situation, use a *row set*. The `RowSet` interface extends the `ResultSet` interface, but row sets don't have to be tied to a database connection.

Row sets are also suitable if you need to move a query result to a different tier of a complex application, or to another device such as a cell phone. You would never want to move a result set—its data structures can be huge, and it is tethered to the database connection.

5.7.1 Constructing Row Sets

The `javax.sql.rowset` package provides the following interfaces that extend the `RowSet` interface:

- A `CachedRowSet` allows disconnected operation. We will discuss cached row sets in the following section.
- A `WebRowSet` is a cached row set that can be saved to an XML file. The XML file can be moved to another tier of a web application where it is opened by another `WebRowSet` object.
- The `FilteredRowSet` and `JoinRowSet` interfaces support lightweight operations on row sets that are equivalent to SQL `SELECT` and `JOIN` operations. These operations are carried out on the data stored in row sets, without having to make a database connection.
- A `JdbcRowSet` is a thin wrapper around a `ResultSet`. It adds useful methods from the `RowSet` interface.

As of Java SE 7, there is a standard way for obtaining a row set:

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

There are similar methods for obtaining the other row set types.

Before Java SE 7, there were vendor-specific methods for creating row sets. In addition, the JDK supplies reference implementations in the package `com.sun.rowset`. The class names end in `Impl`, for example, `CachedRowSetImpl`. If you can't use the `RowSetProvider`, you can instead use those classes:

```
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
```

5.7.2 Cached Row Sets

A cached row set contains all data from a result set. Since `CachedRowSet` is a subinterface of the `ResultSet` interface, you can use a cached row set exactly as you would use a result set. Cached row sets confer an important benefit: You can close the connection and still use the row set. As you will see in our sample program in

Listing 5.4, this greatly simplifies the implementation of interactive applications. Each user command simply opens the database connection, issues a query, puts the result in a cached row set, and then closes the database connection.

It is even possible to modify the data in a cached row set. Of course, the modifications are not immediately reflected in the database; you need to make an explicit request to accept the accumulated changes. The `CachedRowSet` then reconnects to the database and issues SQL statements to write the accumulated changes.

You can populate a `CachedRowSet` from a result set:

```
ResultSet result = . . .;
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
crs.populate(result);
conn.close(); // now OK to close the database connection
```

Alternatively, you can let the `CachedRowSet` object establish a connection automatically. Set up the database parameters:

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

Then set the query statement and any parameters.

```
crs.setCommand("SELECT * FROM Books WHERE Publisher_ID = ?");
crs.setString(1, publisherId);
```

Finally, populate the row set with the query result:

```
crs.execute();
```

This call establishes a database connection, issues the query, populates the row set, and disconnects.

If your query result is very large, you would not want to put it into the row set in its entirety. After all, your users will probably only look at a few rows. In that case, specify a page size:

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20);
. . .
crs.execute();
```

Now you will only get 20 rows. To get the next batch of rows, call

```
crs.nextPage();
```

You can inspect and modify the row set with the same methods you use for result sets. If you modified the row set contents, you must write it back to the database by calling

```
crs.acceptChanges(conn);
```

or

```
crs.acceptChanges();
```

The second call works only if you configured the row set with the information required to connect to a database (such as URL, user name, and password).

In Section 5.6.2, “Updatable Result Sets,” on p. 324, you saw that not all result sets are updatable. Similarly, a row set that contains the result of a complex query will not be able to write its changes back to the database. You should be safe if your row set contains data from a single table.



CAUTION: If you populated the row set from a result set, the row set does not know the name of the table to update. You need to call `setTableName` to set the table name.

Another complexity arises if the data in the database have changed after you populated the row set. This is clearly a sign of trouble that could lead to inconsistent data. The reference implementation checks whether the original row set values (that is, the values before editing) are identical to the current values in the database. If so, they are replaced with the edited values; otherwise, a `SyncProviderException` is thrown and none of the changes are written. Other implementations may use other strategies for synchronization.

javax.sql.RowSet 1.4

- `String getURL()`
- `void setURL(String url)`

gets or sets the database URL.

- `String getUsername()`
- `void setUsername(String username)`

gets or sets the user name for connecting to the database.

- `String getPassword()`
- `void setPassword(String password)`

gets or sets the password for connecting to the database.

(Continues)

javax.sql.RowSet 1.4 (Continued)

- `String getCommand()`
- `void setCommand(String command)`
gets or sets the command that is executed to populate this row set.
- `void execute()`
populates this row set by issuing the statement set with `setCommand`. For the driver manager to obtain a connection, the URL, user name, and password must be set.

javax.sql.rowset.CachedRowSet 5.0

- `void execute(Connection conn)`
populates this row set by issuing the statement set with `setCommand`. This method uses the given connection *and closes it*.
- `void populate(ResultSet result)`
populates this cached row set with the data from the given result set.
- `String getTableName()`
- `void setTableName(String tableName)`
gets or sets the name of the table from which this cached row set was populated.
- `int getPageSize()`
- `void setPageSize(int size)`
gets or sets the page size.
- `boolean nextPage()`
- `boolean previousPage()`
loads the next or previous page of rows. Returns `true` if there is a next or previous page.
- `void acceptChanges()`
- `void acceptChanges(Connection conn)`
reconnects to the database and writes the changes that are the result of editing the row set. May throw a `SyncProviderException` if the data cannot be written back because the database data have changed.

javax.sql.rowset.RowSetProvider 7

- `static RowSetFactory newFactory()`
creates a row set factory.

javax.sql.rowset.RowSetFactory 7

- `CachedRowSet createCachedRowSet()`
- `FilteredRowSet createFilteredRowSet()`
- `JdbcRowSet createJdbcRowSet()`
- `JoinRowSet createJoinRowSet()`
- `WebRowSet createWebRowSet()`

creates a row set of the specified type.

5.8 Metadata

In the preceding sections, you saw how to populate, query, and update database tables. However, JDBC can give you additional information about the *structure* of a database and its tables. For example, you can get a list of the tables in a particular database or the column names and types of a table. This information is not useful when you are implementing a business application with a predefined database. After all, if you design the tables, you know their structure. Structural information is, however, extremely useful for programmers who write tools that work with any database.

In SQL, data that describe the database or one of its parts are called *metadata* (to distinguish them from the actual data stored in the database). You can get three kinds of metadata: about a database, about a result set, and about parameters of prepared statements.

To find out more about the database, request an object of type `DatabaseMetaData` from the database connection.

```
DatabaseMetaData meta = conn.getMetaData();
```

Now you are ready to get some metadata. For example, the call

```
ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
```

returns a result set that contains information about all tables in the database. (See the API note at the end of this section for other parameters to this method.)

Each row in the result set contains information about a table in the database. The third column is the name of the table. (Again, see the API note for the other columns.) The following loop gathers all table names:

```
while (mrs.next())
    tableNames.addItem(mrs.getString(3));
```

There is a second important use for database metadata. Databases are complex, and the SQL standard leaves plenty of room for variability. Well over a hundred methods in the `DatabaseMetaData` interface can inquire about the database, including calls with such exotic names as

```
meta.supportsCatalogsInPrivilegeDefinitions()
```

and

```
meta.nullPlusNonNullIsNull()
```

Clearly, these are geared toward advanced users with special needs—in particular, those who need to write highly portable code that works with multiple databases.

The `DatabaseMetaData` interface gives data about the database. A second metadata interface, `ResultSetMetaData`, reports information about a result set. Whenever you have a result set from a query, you can inquire about the number of columns and each column's name, type, and field width. Here is a typical loop:

```
ResultSet rs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData meta = rs.getMetaData();
for (int i = 1; i <= meta.getColumnCount(); i++)
{
    String columnName = meta.getColumnName(i);
    int columnWidth = meta.getColumnDisplaySize(i);
    ...
}
```

In this section, we will show you how to write such a simple tool. The program in Listing 5.4 uses metadata to let you browse all tables in a database. The program also illustrates the use of a cached row set.

The combo box on top displays all tables in the database. Select one of them, and the center of the frame is filled with the field names of that table and the values of the first row, as shown in Figure 5.6. Click Next and Previous to scroll through the rows in the table. You can also delete a row and edit the row values. Click the Save button to save the changes to the database.

NOTE: Many databases come with much more sophisticated tools for viewing and editing tables. If your database doesn't, check out iSQL-Viewer (<http://isql.sourceforge.net>) or SQuirreL (<http://squirrel-sql.sourceforge.net>). These programs can view the tables in any JDBC database. Our example program is not intended as a replacement for these tools, but it shows you how to implement a tool for working with arbitrary tables.

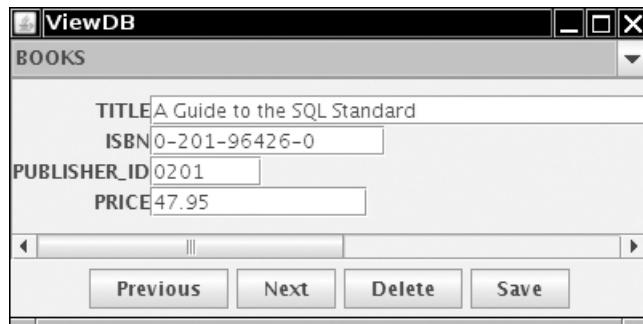


Figure 5.6 The ViewDB application

Listing 5.4 view/ViewDB.java

```
1 package view;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.nio.file.*;
7 import java.sql.*;
8 import java.util.*;
9
10 import javax.sql.*;
11 import javax.sql.rowset.*;
12 import javax.swing.*;
13
14 /**
15 * This program uses metadata to display arbitrary tables in a database.
16 * @version 1.33 2016-04-27
17 * @author Cay Horstmann
18 */
19 public class ViewDB
20 {
21     public static void main(String[] args)
22     {
23         EventQueue.invokeLater(() ->
24         {
25             JFrame frame = new ViewDBFrame();
26             frame.setTitle("ViewDB");
27             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28             frame.setVisible(true);
29         });
30     }
31 }
```

(Continues)

Listing 5.4 (Continued)

```
32 /**
33 * The frame that holds the data panel and the navigation buttons.
34 */
35
36 class ViewDBFrame extends JFrame
37 {
38     private JButton previousButton;
39     private JButton nextButton;
40     private JButton deleteButton;
41     private JButton saveButton;
42     private DataPanel dataPanel;
43     private Component scrollPane;
44     private JComboBox<String> tableNames;
45     private Properties props;
46     private CachedRowSet crs;
47     private Connection conn;
48
49     public ViewDBFrame()
50     {
51         tableNames = new JComboBox<String>();
52
53         try
54         {
55             readDatabaseProperties();
56             conn = getConnection();
57             DatabaseMetaData meta = conn.getMetaData();
58             try (ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" })) {
59                 {
60                     while (mrs.next())
61                         tableNames.addItem(mrs.getString(3));
62                 }
63             }
64             catch (SQLException ex)
65             {
66                 for (Throwable t : ex)
67                     t.printStackTrace();
68             }
69             catch (IOException ex)
70             {
71                 ex.printStackTrace();
72             }
73             tableNames.addActionListener(
74                 event -> showTable((String) tableNames.getSelectedItem(), conn));
75             add(tableNames, BorderLayout.NORTH);
76             addWindowListener(new WindowAdapter()
77             {
78
```

```
79         public void windowClosing(WindowEvent event)
80     {
81         try
82     {
83         if (conn != null) conn.close();
84     }
85     catch (SQLException ex)
86     {
87         for (Throwable t : ex)
88             t.printStackTrace();
89     }
90 }
91 );
92
93 JPanel buttonPanel = new JPanel();
94 add(buttonPanel, BorderLayout.SOUTH);
95
96 previousButton = new JButton("Previous");
97 previousButton.addActionListener(event -> showPreviousRow());
98 buttonPanel.add(previousButton);
99
100 nextButton = new JButton("Next");
101 nextButton.addActionListener(event -> showNextRow());
102 buttonPanel.add(nextButton);
103
104 deleteButton = new JButton("Delete");
105 deleteButton.addActionListener(event -> deleteRow());
106 buttonPanel.add(deleteButton);
107
108 saveButton = new JButton("Save");
109 saveButton.addActionListener(event -> saveChanges());
110 buttonPanel.add(saveButton);
111 if (tableNames.getItemCount() > 0)
112     showTable(tableNames.getItemAt(0), conn);
113 }
114
115 /**
116 * Prepares the text fields for showing a new table, and shows the first row.
117 * @param tableName the name of the table to display
118 * @param conn the database connection
119 */
120 public void showTable(String tableName, Connection conn)
121 {
122     try (Statement stat = conn.createStatement();
123          ResultSet result = stat.executeQuery("SELECT * FROM " + tableName))
124     {
125         // get result set
126
127         // copy into cached row set
```

(Continues)

Listing 5.4 (Continued)

```
128     RowSetFactory factory = RowSetProvider.newFactory();
129     crs = factory.createCachedRowSet();
130     crs.setTableName(tableName);
131     crs.populate(result);
132
133     if (scrollPane != null) remove(scrollPane);
134     dataPanel = new DataPanel(crs);
135     scrollPane = new JScrollPane(dataPanel);
136     add(scrollPane, BorderLayout.CENTER);
137     pack();
138     showNextRow();
139 }
140 catch (SQLException ex)
141 {
142     for (Throwable t : ex)
143         t.printStackTrace();
144 }
145 }
146 /**
147 * Moves to the previous table row.
148 */
149 public void showPreviousRow()
150 {
151     try
152     {
153         if (crs == null || crs.isFirst()) return;
154         crs.previous();
155         dataPanel.showRow(crs);
156     }
157     catch (SQLException ex)
158     {
159         for (Throwable t : ex)
160             t.printStackTrace();
161     }
162 }
163 }
164 /**
165 * Moves to the next table row.
166 */
167 public void showNextRow()
168 {
169     try
170     {
171         if (crs == null || crs.isLast()) return;
172         crs.next();
173         dataPanel.showRow(crs);
174     }
```

```
176     catch (SQLException ex)
177     {
178         for (Throwable t : ex)
179             t.printStackTrace();
180     }
181 }
182
183 /**
184 * Deletes current table row.
185 */
186 public void deleteRow()
187 {
188     if (crs == null) return;
189     new SwingWorker<Void, Void>()
190     {
191         public Void doInBackground() throws SQLException
192         {
193             crs.deleteRow();
194             crs.acceptChanges(conn);
195             if (crs.isAfterLast())
196                 if (!crs.last()) crs = null;
197             return null;
198         }
199         public void done()
200         {
201             dataPanel.showRow(crs);
202         }
203     }.execute();
204 }
205
206 /**
207 * Saves all changes.
208 */
209 public void saveChanges()
210 {
211     if (crs == null) return;
212     new SwingWorker<Void, Void>()
213     {
214         public Void doInBackground() throws SQLException
215         {
216             dataPanel.setRow(crs);
217             crs.acceptChanges(conn);
218             return null;
219         }
220     }.execute();
221 }
222
223 private void readDatabaseProperties() throws IOException
224 {
```

(Continues)

Listing 5.4 (Continued)

```
225     props = new Properties();
226     try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
227     {
228         props.load(in);
229     }
230     String drivers = props.getProperty("jdbc.drivers");
231     if (drivers != null) System.setProperty("jdbc.drivers", drivers);
232 }
233 /**
234 * Gets a connection from the properties specified in the file database.properties.
235 * @return the database connection
236 */
237 private Connection getConnection() throws SQLException
238 {
239     String url = props.getProperty("jdbc.url");
240     String username = props.getProperty("jdbc.username");
241     String password = props.getProperty("jdbc.password");
242
243     return DriverManager.getConnection(url, username, password);
244 }
245 }
246 /**
247 * This panel displays the contents of a result set.
248 */
249 class DataPanel extends JPanel
250 {
251     private java.util.List<JTextField> fields;
252
253     /**
254      * Constructs the data panel.
255      * @param rs the result set whose contents this panel displays
256      */
257     public DataPanel(ResultSet rs) throws SQLException
258     {
259         fields = new ArrayList<>();
260         setLayout(new GridBagLayout());
261         GridBagConstraints gbc = new GridBagConstraints();
262         gbc.gridwidth = 1;
263         gbc.gridheight = 1;
264
265         ResultSetMetaData rsmd = rs.getMetaData();
266         for (int i = 1; i <= rsmd.getColumnCount(); i++)
267         {
268             JTextField field = new JTextField(10);
269             fields.add(field);
270             gbc.gridx = i;
271             add(field, gbc);
272         }
273     }
274 }
```

```
270     gbc.gridx = i - 1;
271
272     String columnName = rsmd.getColumnLabel(i);
273     gbc.gridx = 0;
274     gbc.anchor = GridBagConstraints.EAST;
275     add(new JLabel(columnName), gbc);
276
277     int columnWidth = rsmd getColumnDisplaySize(i);
278     JTextField tb = new JTextField(columnWidth);
279     if (!rsmd.getColumnClassName(i).equals("java.lang.String"))
280         tb.setEditable(false);
281
282     fields.add(tb);
283
284     gbc.gridx = 1;
285     gbc.anchor = GridBagConstraints.WEST;
286     add(tb, gbc);
287 }
288 }
289
290 /**
291 * Shows a database row by populating all text fields with the column values.
292 */
293 public void showRow(ResultSet rs)
294 {
295     try
296     {
297         if (rs == null) return;
298         for (int i = 1; i <= fields.size(); i++)
299         {
300             String field = rs == null ? "" : rs.getString(i);
301             JTextField tb = fields.get(i - 1);
302             tb.setText(field);
303         }
304     }
305     catch (SQLException ex)
306     {
307         for (Throwable t : ex)
308             t.printStackTrace();
309     }
310 }
311
312 /**
313 * Updates changed data into the current row of the row set.
314 */
315 public void setRow(ResultSet rs) throws SQLException
316 {
```

(Continues)

Listing 5.4 (Continued)

```

317     for (int i = 1; i <= fields.size(); i++)
318     {
319         String field = rs.getString(i);
320         JTextField tb = fields.get(i - 1);
321         if (!field.equals(tb.getText()))
322             rs.updateString(i, tb.getText());
323     }
324     rs.updateRow();
325 }
326 }
```

***java.sql.Connection* 1.1**

- `DatabaseMetaData getMetaData()`

returns the metadata for the connection as a `DatabaseMetaData` object.

***java.sql.DatabaseMetaData* 1.1**

- `ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])`

returns a description of all tables in a catalog that match the schema and table name patterns and the type criteria. (*A schema* describes a group of related tables and access permissions. A *catalog* describes a related group of schemas. These concepts are important for structuring large databases.)

The `catalog` and `schemaPattern` parameters can be `""` to retrieve those tables without a catalog or schema, or `null` to return tables regardless of catalog or schema.

The `types` array contains the names of the table types to include. Typical types are `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS`, and `SYNONYM`. If `types` is `null`, tables of all types are returned.

The result set has five columns, all of which are of type `String`.

Column	Name	Explanation
1	TABLE_CAT	Table catalog (may be <code>null</code>)
2	TABLE_SCHEM	Table schema (may be <code>null</code>)
3	TABLE_NAME	Table name
4	TABLE_TYPE	Table type
5	REMARKS	Comment on the table

(Continues)

java.sql.DatabaseMetaData 1.1 (Continued)

- `int getJDBCMajorVersion() 1.4`
- `int getJDBCMinorVersion() 1.4`

returns the major or minor JDBC version numbers of the driver that established the database connection. For example, a JDBC 3.0 driver has major version number 3 and minor version number 0.

- `int getMaxConnections()`
returns the maximum number of concurrent connections allowed to this database.
- `int getMaxStatements()`
returns the maximum number of concurrently open statements allowed per database connection, or 0 if the number is unlimited or unknown.

java.sql.ResultSet 1.1

- `ResultSetMetaData getMetaData()`
returns the metadata associated with the current ResultSet columns.

java.sql.ResultSetMetaData 1.1

- `int getColumnCount()`

returns the number of columns in the current ResultSet object.

- `int getColumnDisplaySize(int column)`

returns the maximum width of the column specified by the index parameter.

Parameters: `column` The column number

- `String getColumnLabel(int column)`

returns the suggested title for the column.

Parameters: `column` The column number

- `String getColumnName(int column)`

returns the column name associated with the column index specified.

Parameters: `column` The column number

5.9 Transactions

You can group a set of statements to form a *transaction*. The transaction can be *committed* when all has gone well. Or, if an error has occurred in one of them, it can be *rolled back* as if none of the statements had been issued.

The major reason for grouping statements into transactions is *database integrity*. For example, suppose we want to transfer money from one bank account to another. Then, it is important that we simultaneously debit one account and credit another. If the system fails after debiting the first account but before crediting the other account, the debit needs to be undone.

If you group update statements into a transaction, the transaction either succeeds in its entirety and can be *committed*, or it fails somewhere in the middle. In that case, you can carry out a *rollback* and the database automatically undoes the effect of all updates that occurred since the last committed transaction.

5.9.1 Programming Transactions with JDBC

By default, a database connection is in *autocommit mode*, and each SQL statement is committed to the database as soon as it is executed. Once a statement is committed, you cannot roll it back. Turn off this default so you can use transactions:

```
conn.setAutoCommit(false);
```

Create a statement object in the normal way:

```
Statement stat = conn.createStatement();
```

Call `executeUpdate` any number of times:

```
stat.executeUpdate(command1);
stat.executeUpdate(command2);
stat.executeUpdate(command3);
. . .
```

If all statements have been executed without error, call the `commit` method:

```
conn.commit();
```

However, if an error occurred, call

```
conn.rollback();
```

Then, all statements since the last commit are automatically reversed. You typically issue a rollback when your transaction was interrupted by a `SQLException`.

5.9.2 Save Points

With some databases and drivers, you can gain finer-grained control over the rollback process by using *save points*. Creating a save point marks a point to which you can later return without having to abandon the entire transaction. For example,

```
Statement stat = conn.createStatement(); // start transaction; rollback() goes here  
stat.executeUpdate(command1);  
Savepoint svpt = conn.setSavepoint(); // set savepoint; rollback(svpt) goes here  
stat.executeUpdate(command2);  
  
if (...) conn.rollback(svpt); // undo effect of command2  
...  
conn.commit();
```

When you no longer need a save point, you should release it:

```
conn.releaseSavepoint(svpt);
```

5.9.3 Batch Updates

Suppose a program needs to execute many `INSERT` statements to populate a database table. You can improve the performance of the program by using a *batch update*. In a batch update, a sequence of statements is collected and submitted as a batch.

NOTE: Use the `supportsBatchUpdates` method of the `DatabaseMetaData` interface to find out if your database supports this feature.

The statements in a batch can be actions such as `INSERT`, `UPDATE`, or `DELETE` as well as data definition statements such as `CREATE TABLE` or `DROP TABLE`. An exception is thrown if you add a `SELECT` statement to a batch. (Conceptually, a `SELECT` statement makes no sense in a batch because it returns a result set without updating the database.)

To execute a batch, first create a `Statement` object in the usual way:

```
Statement stat = conn.createStatement();
```

Now, instead of calling `executeUpdate`, call the `addBatch` method:

```
String command = "CREATE TABLE . . ."  
stat.addBatch(command);  
  
while (...)  
{  
    command = "INSERT INTO . . . VALUES (" + . . . + ")";  
    stat.addBatch(command);  
}
```

Finally, submit the entire batch:

```
int[] counts = stat.executeBatch();
```

The call to `executeBatch` returns an array of the row counts for all submitted statements.

For proper error handling in batch mode, treat the batch execution as a single transaction. If a batch fails in the middle, you want to roll back to the state before the beginning of the batch.

First, turn the autocommit mode off, then collect the batch, execute it, commit it, and finally restore the original autocommit mode:

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.createStatement();
. . .
// keep calling stat.addBatch(. . .);
. . .
stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```

java.sql.Connection 1.1

- `boolean getAutoCommit()`
- `void setAutoCommit(boolean b)`
gets or sets the autocommit mode of this connection to `b`. If autocommit is `true`, all statements are committed as soon as their execution is completed.
- `void commit()`
commits all statements that were issued since the last commit.
- `void rollback()`
undoes the effect of all statements that were issued since the last commit.
- `Savepoint setSavepoint() 1.4`
- `Savepoint setSavepoint(String name) 1.4`
sets an unnamed or named save point.
- `void rollback(Savepoint svpt) 1.4`
rolls back until the given save point.
- `void releaseSavepoint(Savepoint svpt) 1.4`
releases the given save point.

java.sql.Savepoint 1.4

- **int getSavepointId()**
gets the ID of this unnamed save point, or throws a `SQLException` if this is a named save point.
- **String getSavepointName()**
gets the name of this save point, or throws a `SQLException` if this is an unnamed save point.

java.sql.Statement 1.1

- **void addBatch(String command) 1.2**
adds the command to the current batch of commands for this statement.
- **int[] executeBatch() 1.2**
- **long[] executeLargeBatch() 8**
executes all commands in the current batch. Each value in the returned array corresponds to one of the batch statements. If it is non-negative, it is a row count. If it is the value `SUCCESS_NO_INFO`, the statement succeeded, but no row count is available. If it is `EXECUTE_FAILED`, the statement failed.

java.sql.DatabaseMetaData 1.1

- **boolean supportsBatchUpdates() 1.2**
returns true if the driver supports batch updates.

5.10 Advanced SQL Types

Table 5.8 lists the SQL data types supported by JDBC and their equivalents in the Java programming language.

A SQL `ARRAY` is a sequence of values. For example, in a `Student` table, you can have a `Scores` column that is an `ARRAY OF INTEGER`. The `getArray` method returns an object of the interface type `java.sql.Array`. That interface has methods to fetch the array values.

When you get a LOB or an array from a database, the actual contents are fetched from the database only when you request individual values. This is a useful performance enhancement, as the data can be quite voluminous.

Table 5.8 SQL Data Types and Their Corresponding Java Types

SQL Data Type	Java Data Type
INTEGER or INT	int
SMALLINT	short
NUMERIC(m,n), DECIMAL(m,n) or DEC(m,n)	java.math.BigDecimal
FLOAT(n)	double
REAL	float
DOUBLE	double
CHARACTER(n) or CHAR(n)	String
VARCHAR(n), LONG VARCHAR	String
BOOLEAN	boolean
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
ROWID	java.sql.RowId
NCHAR(n), NVARCHAR(n), LONG NVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

Some databases support ROWID values that describe the location of a row so that it can be retrieved very rapidly. JDBC 4 introduced an interface `java.sql.RowId` and the methods to supply the row ID in queries and retrieve it from results.

A *national character string* (NCHAR and its variants) stores strings in a local character encoding and sorts them using a local sorting convention. JDBC 4 provided methods for converting between Java `String` objects and national character strings in queries and results.

Some databases can store user-defined structured types. JDBC 3 provided a mechanism for automatically mapping structured SQL types to Java objects.

Some databases provide native storage for XML data. JDBC 4 introduced a `SQLXML` interface that can mediate between the internal XML representation and the DOM Source/Result interfaces, as well as binary streams. See the API documentation for the `SQLXML` class for details.

We do not discuss these advanced SQL types any further. You can find more information on these topics in the *JDBC API Tutorial and Reference* and the JDBC specification.

5.11 Connection Management in Web and Enterprise Applications

The simplistic database connection setup with a `database.properties` file, as described in the preceding sections, is suitable for small test programs but won't scale for larger applications.

When a JDBC application is deployed in a web or enterprise environment, the management of database connections is integrated with the JNDI. The properties of data sources across the enterprise can be stored in a directory. Using a directory allows for centralized management of user names, passwords, database names, and JDBC URLs.

In such an environment, you can use the following code to establish a database connection:

```
Context jndiContext = new InitialContext();
DataSource source = (DataSource) jndiContext.lookup("java:comp/env/jdbc/corejava");
Connection conn = source.getConnection();
```

Note that the `DriverManager` is no longer involved. Instead, the JNDI service locates a *data source*. A data source is an interface that allows for simple JDBC connections as well as more advanced services, such as executing distributed transactions that involve multiple databases. The `DataSource` interface is defined in the `javax.sql` standard extension package.

NOTE: In a Java EE container, you don't even have to program the JNDI lookup. Simply use the `Resource` annotation on a `DataSource` field, and the data source reference will be set when your application is loaded:

```
@Resource(name="jdbc/corejava")
private DataSource source;
```

Of course, the data source needs to be configured somewhere. If you write database programs that execute in a servlet container such as Apache Tomcat or in an application server such as GlassFish, place the database configuration

(including the JNDI name, JDBC URL, user name, and password) in a configuration file, or set it in an admin GUI.

Management of user names and logins is just one of the issues that require special attention. Another issue involves the cost of establishing database connections. Our sample database programs used two strategies for obtaining a database connection. The `QueryDB` program in Listing 5.3 established a single database connection at the start of the program and closed it at the end of the program. The `ViewDB` program in Listing 5.4 opened a new connection whenever one was needed.

However, neither of these approaches is satisfactory. Database connections are a finite resource. If a user walks away from an application for some time, the connection should not be left open. Conversely, obtaining a connection for each query and closing it afterward is very costly.

The solution is to *pool* connections. This means that database connections are not physically closed but are kept in a queue and reused. Connection pooling is an important service, and the JDBC specification provides hooks for implementors to supply it. However, the JDK itself does not provide any implementation, and database vendors don't usually include one with their JDBC drivers either. Instead, vendors of web containers and application servers supply connection pool implementations.

Using a connection pool is completely transparent to the programmer. Acquire a connection from a source of pooled connections by obtaining a data source and calling `getConnection`. When you are done using the connection, call `close`. That doesn't close the physical connection but tells the pool that you are done using it. The connection pool typically makes an effort to pool prepared statements as well.

You have now learned about the JDBC fundamentals and know enough to implement simple database applications. However, as we mentioned at the beginning of this chapter, databases are complex and quite a few advanced topics are beyond the scope of this introductory chapter. For an overview of advanced JDBC capabilities, refer to the *JDBC API Tutorial and Reference* or the JDBC specification.

In this chapter, you have learned how to work with relational databases in Java. The next chapter covers the Java 8 date and time library.

CHAPTER

6

The Date and Time API

In this chapter

- 6.1 The Time Line, page 352
- 6.2 Local Dates, page 355
- 6.3 Date Adjusters, page 358
- 6.4 Local Time, page 360
- 6.5 Zoned Time, page 361
- 6.6 Formatting and Parsing, page 365
- 6.7 Interoperating with Legacy Code, page 369

Time flies like an arrow, and we can easily set a starting point and count forward and backward in seconds. So why is it so hard to deal with time? The problem is humans. All would be easy if we could just tell each other: “Meet me at 1523793600, and don’t be late!” But we want time to relate to daylight and the seasons. That’s where things get complicated. Java 1.0 had a `Date` class that was, in hindsight, naïve, and had most of its methods deprecated in Java 1.1 when a `Calendar` class was introduced. Its API wasn’t stellar, its instances were mutable, and it didn’t deal with issues such as leap seconds. The third time is a charm, and the `java.time` API introduced in Java SE 8 has remedied the flaws of the past and should serve us for quite some time. In this chapter, you will learn what makes time computations so vexing, and how the Date and Time API solves these issues.

6.1 The Time Line

Historically, the fundamental time unit—the second—was derived from Earth’s rotation around its axis. There are 24 hours or $24 \times 60 \times 60 = 86400$ seconds in a full revolution, so it seems just a question of astronomical measurements to precisely define a second. Unfortunately, Earth wobbles slightly, and a more precise definition was needed. In 1967, a new precise definition of a second, matching the historical definition, was derived from an intrinsic property of atoms of caesium-133. Since then, a network of atomic clocks keeps the official time.

Ever so often, the official time keepers synchronize the absolute time with the rotation of Earth. At first, the official seconds were slightly adjusted, but starting in 1972, “leap seconds” were occasionally inserted. (In theory, a second might need to be removed once in a while, but that has not yet happened.) There is talk of changing the system again. Clearly, leap seconds are a pain, and many computer systems instead use “smoothing” where time is artificially slowed down or sped up just before the leap second, keeping 86,400 seconds per day. This works because the local time on a computer isn’t all that precise, and computers are used to synchronizing themselves with an external time service.

The Java Date and Time API specification requires that Java uses a time scale that:

- Has 86,400 seconds per day.
- Exactly matches the official time at noon each day.
- Closely matches it elsewhere, in a precisely defined way.

That gives Java the flexibility to adjust to future changes in the official time.

In Java, an `Instant` represents a point on the time line. An origin, called the *epoch*, is arbitrarily set at midnight of January 1, 1970 at the prime meridian that passes through the Greenwich Royal Observatory in London. This is the same convention used in the UNIX/POSIX time. Starting from that origin, time is measured in 86,400 seconds per day, forward and backward, to nanosecond precision. The `Instant` values go back as far as a billion years (`Instant.MIN`). That’s not quite enough to express the age of the universe (around 13.5 billion years), but it should be enough for all practical purposes. After all, a billion years ago, the earth was covered in ice and populated by microscopic ancestors of today’s plants and animals. The largest value, `Instant.MAX`, is December 31 of the year 1,000,000,000.

The static method call `Instant.now()` gives the current instant. You can compare two instants with the `equals` and `compareTo` methods in the usual way, so you can use instants as timestamps.

To find out the difference between two instants, use the static method `Duration.between`. For example, here is how you can measure the running time of an algorithm:

```
Instant start = Instant.now();
runAlgorithm();
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long millis = timeElapsed.toMillis();
```

A Duration is the amount of time between two instants. You can get the length of a Duration in conventional units by calling `toNanos`, `toMillis`, `getSeconds`, `toMinutes`, `toHours`, or `toDays`.

Durations require more than a `long` value for their internal storage. The number of seconds is stored in a `long`, and the number of nanoseconds in an additional `int`. If you want to make computations to nanosecond accuracy, and you actually need the entire range of a Duration, you can use one of the methods in Table 6.1. Otherwise, you can just call `toNanos` and do your calculations with `long` values.



NOTE: It takes almost 300 years of nanoseconds to overflow a `long`.

For example, if you want to check whether an algorithm is at least ten times faster than another, you can compute

```
Duration timeElapsed2 = Duration.between(start2, end2);
boolean overTenTimesFaster =
    timeElapsed.multipliedBy(10).minus(timeElapsed2).isNegative();
// Or timeElapsed.toNanos() * 10 < timeElapsed2.toNanos()
```

Table 6.1 Arithmetic Operations for Time Instants and Durations

Method	Description
<code>plus</code> , <code>minus</code>	Adds a duration to, or subtracts a duration from, this Instant or Duration.
<code>plusNanos</code> , <code>plusMillis</code> , <code>plusSeconds</code> , <code>minusNanos</code> , <code>minusMillis</code> , <code>minusSeconds</code>	Adds or subtracts a number of the given time units to this Instant or Duration.
<code>plusMinutes</code> , <code>plusHours</code> , <code>plusDays</code> , <code>minusMinutes</code> , <code>minusHours</code> , <code>minusDays</code>	Adds or subtracts a number of the given time units to this Duration.
<code>multipliedBy</code> , <code>dividedBy</code> , <code>negated</code>	Returns a duration obtained by multiplying or dividing this Duration by a given <code>long</code> , or by <code>-1</code> . Note that you can scale only durations, not instants.
<code>isZero</code> , <code>isNegative</code>	Checks whether this Duration is zero or negative.



NOTE: The Instant and Duration classes are immutable, and all methods, such as multipliedBy or minus, return a new instance.

In the example program in Listing 6.1, you can see how to use the Instant and Duration classes for timing two algorithms.

Listing 6.1 timeline/TimeLine.java

```
1 package timeline;
2
3 import java.time.*;
4 import java.util.*;
5 import java.util.stream.*;
6
7 public class Timeline
8 {
9     public static void main(String[] args)
10    {
11         Instant start = Instant.now();
12         runAlgorithm();
13         Instant end = Instant.now();
14         Duration timeElapsed = Duration.between(start, end);
15         long millis = timeElapsed.toMillis();
16         System.out.printf("%d milliseconds\n", millis);
17
18         Instant start2 = Instant.now();
19         runAlgorithm2();
20         Instant end2 = Instant.now();
21         Duration timeElapsed2 = Duration.between(start2, end2);
22         System.out.printf("%d milliseconds\n", timeElapsed2.toMillis());
23         boolean overTenTimesFaster = timeElapsed.multipliedBy(10)
24             .minus(timeElapsed2).isNegative();
25         System.out.printf("The first algorithm is %smore than ten times faster",
26             overTenTimesFaster ? "" : "not ");
27    }
28
29    public static void runAlgorithm()
30    {
31        int size = 10;
32        List<Integer> list = new Random().ints().map(i -> i % 100).limit(size)
33            .boxed().collect(Collectors.toList());
34        Collections.sort(list);
35        System.out.println(list);
36    }
37
```

```
38  public static void runAlgorithm2()
39  {
40      int size = 10;
41      List<Integer> list = new Random().ints().map(i -> i % 100).limit(size)
42          .boxed().collect(Collectors.toList());
43      while (!IntStream.range(1, list.size()).allMatch(
44          i -> list.get(i - 1).compareTo(list.get(i)) <= 0))
45          Collections.shuffle(list);
46      System.out.println(list);
47  }
48 }
```

6.2 Local Dates

Now let us turn from absolute time to human time. There are two kinds of human time in the Java API, *local date/time* and *zoned time*. Local date/time has a date and/or time of day, but no associated time zone information. An example of a local date is June 14, 1903 (the day on which Alonzo Church, inventor of the lambda calculus, was born). Since that date has neither a time of day nor time zone information, it does not correspond to a precise instant of time. In contrast, July 16, 1969, 09:32:00 EDT (the launch of Apollo 11) is a zoned date/time, representing a precise instant on the time line.

There are many calculations where time zones are not required, and in some cases they can even be a hindrance. Suppose you schedule a meeting every week at 10:00. If you add 7 days (that is, $7 \times 24 \times 60 \times 60$ seconds) to the last zoned time, and you happen to cross the daylight savings time boundary, the meeting will be an hour too early or too late!

For that reason, the API designers recommend that you do not use zoned time unless you really want to represent absolute time instances. Birthdays, holidays, schedule times, and so on are usually best represented as local dates or times.

A `LocalDate` is a date with a year, month, and day of the month. To construct one, you can use the `now` or of static methods:

```
LocalDate today = LocalDate.now(); // Today's date
LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
alonzoBirthday = LocalDate.of(1903, Month.JUNE, 14);
// Uses the Month enumeration
```

Unlike the irregular conventions in UNIX and `java.util.Date`, where months are zero-based and years are counted from 1900, you supply the usual numbers for the month of year. Alternatively, you can use the `Month` enumeration.

Table 6.2 shows the most useful methods for working with `LocalDate` objects.

Table 6.2 LocalDate Methods

Method	Description
now, of	These static methods construct a LocalDate, either from the current time or from a given year, month, and day.
plusDays, plusWeeks, plusMonths, plusYears	Adds a number of days, weeks, months, or years to this LocalDate.
minusDays, minusWeeks, minusMonths, minusYears	Subtracts a number of days, weeks, months, or years from this LocalDate.
plus, minus	Adds or subtracts a Duration or Period.
withDayOfMonth, withDayOfYear, withMonth, withYear	Returns a new LocalDate with the day of month, day of year, month, or year changed to the given value.
getDayOfMonth	Gets the day of the month (between 1 and 31).
getDayOfYear	Gets the day of the year (between 1 and 366).
getDayOfWeek	Gets the day of the week, returning a value of the DayOfWeek enumeration.
getMonth, getMonthValue	Gets the month as a value of the Month enumeration, or as a number between 1 and 12.
getYear	Gets the year, between -999,999,999 and 999,999,999.
until	Gets the Period, or the number of the given ChronoUnits, between two dates.
isBefore, isAfter	Compares this LocalDate with another.
isLeapYear	Returns true if the year is a leap year—that is, if it is divisible by 4 but not by 100, or divisible by 400. The algorithm is applied for all past years, even though that is historically inaccurate. (Leap years were invented in the year -46, and the rules involving divisibility by 100 and 400 were introduced in the Gregorian calendar reform of 1582. The reform took over 300 years to become universal.)

For example, *Programmer’s Day* is the 256th day of the year. Here is how you can easily compute it:

```
LocalDate programmersDay = LocalDate.of(2014, 1, 1).plusDays(255);
// September 13, but in a leap year it would be September 12
```

Recall that the difference between two time instants is a Duration. The equivalent for local dates is a Period, which expresses a number of elapsed years, months, or

days. You can call `birthday.plus(Period.ofYears(1))` to get the birthday next year. Of course, you can also just call `birthday.plusYears(1)`. But `birthday.plus(Duration.ofDays(365))` won't produce the correct result in a leap year.

The `until` method yields the difference between two local dates. For example,

```
independenceDay.until(christmas)
```

yields a period of 5 months and 21 days. That is actually not terribly useful because the number of days per month varies. To find the number of days, use

```
independenceDay.until(christmas, ChronoUnit.DAYS) // 174 days
```



CAUTION: Some methods in Table 6.2 could potentially create nonexistent dates. For example, adding one month to January 31 should not yield February 31. Instead of throwing an exception, these methods return the last valid day of the month. For example,

```
LocalDate.of(2016, 1, 31).plusMonths(1)
```

and

```
LocalDate.of(2016, 3, 31).minusMonths(1)
```

yield February 29, 2016.

The `getDayOfWeek` yields the weekday, as a value of the `DayOfWeek` enumeration. `DayOfWeek.MONDAY` has the numerical value 1, and `DayOfWeek.SUNDAY` has the value 7. For example,

```
LocalDate.of(1900, 1, 1).getDayOfWeek().getValue()
```

yields 1. The `DayOfWeek` enumeration has convenience methods `plus` and `minus` to compute weekdays modulo 7. For example, `DayOfWeek.SATURDAY.plus(3)` yields `DayOfWeek.TUESDAY`.



NOTE: The weekend days actually come at the end of the week. This is different from `java.util.Calendar` where Sunday has value 1 and Saturday value 7.

In addition to `LocalDate`, there are also classes `MonthDay`, `YearMonth`, and `Year` to describe partial dates. For example, December 25 (with the year unspecified) can be represented as a `MonthDay`.

The example program in Listing 6.2 shows how to work with the `LocalDate` class.

Listing 6.2 localdates/LocalDates.java

```
1 package localdates;
2
3 import java.time.*;
4 import java.time.temporal.*;
5
6 public class LocalDates
7 {
8     public static void main(String[] args)
9     {
10         LocalDate today = LocalDate.now(); // Today's date
11         System.out.println("today: " + today);
12
13         LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
14         alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);
15         // Uses the Month enumeration
16         System.out.println("alonzosBirthday: " + alonzosBirthday);
17
18         LocalDate programmersDay = LocalDate.of(2018, 1, 1).plusDays(255);
19         // September 13, but in a leap year it would be September 12
20         System.out.println("programmersDay: " + programmersDay);
21
22         LocalDate independenceDay = LocalDate.of(2018, Month.JULY, 4);
23         LocalDate christmas = LocalDate.of(2018, Month.DECEMBER, 25);
24
25         System.out.println("Until christmas: " + independenceDay.until(christmas));
26         System.out.println("Until christmas: "
27             + independenceDay.until(christmas, ChronoUnit.DAYS));
28
29         System.out.println(LocalDate.of(2016, 1, 31).plusMonths(1));
30         System.out.println(LocalDate.of(2016, 3, 31).minusMonths(1));
31
32         DayOfWeek startOfLastMillennium = LocalDate.of(1900, 1, 1).getDayOfWeek();
33         System.out.println("startOfLastMillennium: " + startOfLastMillennium);
34         System.out.println(startOfLastMillennium.getValue());
35         System.out.println(DayOfWeek.SATURDAY.plus(3));
36     }
37 }
```

6.3 Date Adjusters

For scheduling applications, you often need to compute dates such as “the first Tuesday of every month.” The `TemporalAdjusters` class provides a number of static methods for common adjustments. You pass the result of an adjustment method to the `with` method. For example, the first Tuesday of a month can be computed like this:

```
LocalDate firstTuesday = LocalDate.of(year, month, 1).with(
    TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

As always, the `with` method returns a new `LocalDate` object without modifying the original. Table 6.3 shows the available adjusters.

Table 6.3 Date Adjusters in the `TemporalAdjusters` Class

Method	Description
<code>next(weekday)</code> , <code>previous(weekday)</code>	Next or previous date that falls on the given weekday
<code>nextOrSame(weekday)</code> , <code>previousOrSame(weekday)</code>	Next or previous date that falls on the given weekday, starting from the given date
<code>dayOfWeekInMonth(n, weekday)</code>	The <code>n</code> th weekday in the month
<code>lastInMonth(weekday)</code>	The last weekday in the month
<code>firstDayOfMonth()</code> , <code>firstDayOfNextMonth()</code> , <code>firstDayOfNextYear()</code> , <code>lastDayOfMonth()</code> , <code>lastDayOfYear()</code>	The date described in the method name

You can also make your own adjuster by implementing the `TemporalAdjuster` interface. Here is an adjuster for computing the next weekday.

```
TemporalAdjuster NEXT_WORKDAY = w ->
{
    LocalDate result = (LocalDate) w;
    do
    {
        result = result.plusDays(1);
    }
    while (result.getDayOfWeek().getValue() >= 6);
    return result;
};

LocalDate backToWork = today.with(NEXT_WORKDAY);
```

Note that the parameter of the lambda expression has type `Temporal`, and it must be cast to `LocalDate`. You can avoid this cast with the `ofDateAdjuster` method that expects a lambda of type `UnaryOperator<LocalDate>`.

```
TemporalAdjuster NEXT_WORKDAY = TemporalAdjusters.ofDateAdjuster(w ->
{
    LocalDate result = w; // No cast
    do
    {
        result = result.plusDays(1);
```

```

        }
        while (result.getDayOfWeek().getValue() >= 6);
        return result;
    });
}

```

6.4 Local Time

A `LocalTime` represents a time of day, such as 15:30:00. You can create an instance with the `now` or `of` methods:

```

LocalTime rightNow = LocalTime.now();
LocalTime bedtime = LocalTime.of(22, 30); // or LocalTime.of(22, 30, 0)

```

Table 6.4 shows common operations with local times. The `plus` and `minus` operations wrap around a 24-hour day. For example,

```
LocalTime wakeup = bedtime.plusHours(8); // wakeup is 6:30:00
```



NOTE: `LocalTime` doesn't concern itself with AM/PM. That silliness is left to a formatter—see Section 6.6, “Formatting and Parsing,” on p. 365.

Table 6.4 LocalTime Methods

Method	Description
<code>now, of</code>	These static methods construct a <code>LocalTime</code> , either from the current time or from the given hours, minutes, and, optionally, seconds and nanoseconds.
<code>plusHours, plusMinutes, plusSeconds, plusNanos</code>	Adds a number of hours, minutes, seconds, or nanoseconds to this <code>LocalTime</code> .
<code>minusHours, minusMinutes, minusSeconds, minusNanos</code>	Subtracts a number of hours, minutes, seconds, or nanoseconds from this <code>LocalTime</code> .
<code>plus, minus</code>	Adds or subtracts a <code>Duration</code> .
<code>withHour, withMinute, withSecond, withNano</code>	Returns a new <code>LocalTime</code> with the hour, minute, second, or nanosecond changed to the given value.
<code>getHour, getMinute, getSecond, getNano</code>	Gets the hour, minute, second, or nanosecond of this <code>LocalTime</code> .
<code>toSecondOfDay, toNanoOfDay</code>	Returns the number of seconds or nanoseconds between midnight and this <code>LocalTime</code> .
<code>isBefore, isAfter</code>	Compares this <code>LocalTime</code> with another.

There is a `LocalDateTime` class representing a date and time. That class is suitable for storing points in time in a fixed time zone—for example, for a schedule of classes or events. However, if you need to make calculations that span the daylight savings time, or if you need to deal with users in different time zones, you should use the `ZonedDateTime` class that we discuss next.

6.5 Zoned Time

Time zones, perhaps because they are an entirely human creation, are even messier than the complications caused by the earth's irregular rotation. In a rational world, we'd all follow the clock in Greenwich, and some of us would eat our lunch at 02:00, others at 22:00. Our stomachs would figure it out. This is actually done in China, which spans four conventional time zones. Elsewhere, we have time zones with irregular and shifting boundaries and, to make matters worse, the daylight savings time.

As capricious as the time zones may appear to the enlightened, they are a fact of life. When you implement a calendar application, it needs to work for people who fly from one country to another. When you have a conference call at 10:00 in New York, but happen to be in Berlin, you expect to be alerted at the correct local time.

The Internet Assigned Numbers Authority (IANA) keeps a database of all known time zones around the world (www.iana.org/time-zones), which is updated several times per year. The bulk of the updates deals with the changing rules for daylight savings time. Java uses the IANA database.

Each time zone has an ID, such as `America/New_York` or `Europe/Berlin`. To find out all available time zones, call `ZoneId.getAvailableZoneIds`. At the time of this writing, there were almost 600 IDs.

Given a time zone ID, the static method `ZoneId.of(id)` yields a `ZoneId` object. You can use that object to turn a `LocalDateTime` object into a `ZonedDateTime` object by calling `local.atZone(zoneId)`, or you can construct a `ZonedDateTime` by calling the static method `ZonedDateTime.of(year, month, day, hour, minute, second, nano, zoneId)`. For example,

```
ZonedDateTime apollo11Launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,  
ZoneId.of("America/New_York"));  
// 1969-07-16T09:32-04:00[America/New_York]
```

This is a specific instant in time. Call `apollo11Launch.toInstant` to get the `Instant`. Conversely, if you have an instant in time, call `instant.atZone(ZoneId.of("UTC"))` to get the `ZonedDateTime` at the Greenwich Royal Observatory, or use another `ZoneId` to get it elsewhere on the planet.



NOTE: UTC stands for “Coordinated Universal Time,” and the acronym is a compromise between the aforementioned English and the French “Temps Universel Coordiné,” having the distinction of being incorrect in either language. UTC is the time at the Greenwich Royal Observatory, without daylight savings time.

Many of the methods of `ZonedDateTime` are the same as those of `LocalDateTime` (see Table 6.5). Most are straightforward, but daylight savings time introduces some complications.

When daylight savings time starts, clocks advance by an hour. What happens when you construct a time that falls into the skipped hour? For example, in 2013, Central Europe switched to daylight savings time on March 31 at 2:00. If you try to construct nonexistent time March 31 2:30, you actually get 3:30.

```
ZonedDateTime skipped = ZonedDateTime.of(  
    LocalDate.of(2013, 3, 31),  
    LocalTime.of(2, 30),  
    ZoneId.of("Europe/Berlin"));  
// Constructs March 31 3:30
```

Conversely, when daylight time ends, clocks are set back by an hour, and there are two instants with the same local time! When you construct a time within that span, you get the earlier of the two.

```
ZonedDateTime ambiguous = ZonedDateTime.of(  
    LocalDate.of(2013, 10, 27), // End of daylight savings time  
    LocalTime.of(2, 30),  
    ZoneId.of("Europe/Berlin"));  
// 2013-10-27T02:30+02:00[Europe/Berlin]  
ZonedDateTime anHourLater = ambiguous.plusHours(1);  
// 2013-10-27T02:30+01:00[Europe/Berlin]
```

An hour later, the time has the same hours and minutes, but the zone offset has changed.

You also need to pay attention when adjusting a date across daylight savings time boundaries. For example, if you set a meeting for next week, don’t add a duration of seven days:

```
ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));  
// Caution! Won't work with daylight savings time
```

Instead, use the `Period` class.

```
ZonedDateTime nextMeeting = meeting.plus(Period.ofDays(7)); // OK
```

Table 6.5 ZonedDateTime Methods

Method	Description
<code>now, of, ofInstant</code>	Construct a ZonedDateTime from the current time, or from a LocalDateTime, or LocalDate / LocalTime, or year/month/day/hour/minute/second/nanosecond, together with a ZoneId, or from an Instant and ZoneId. These are static methods.
<code>plusDays, plusWeeks, plusMonths, plusYears, plusHours, plusMinutes, plusSeconds, plusNanos</code>	Adds a number of temporal units to this ZonedDateTime.
<code>minusDays, minusWeeks, minusMonths, minusYears, minusHours, minusMinutes, minusSeconds, minusNanos</code>	Subtracts a number of temporal units from this ZonedDateTime.
<code>plus, minus</code>	Adds or subtracts a Duration or Period.
<code>withDayOfMonth, withDayOfYear, withMonth, withYear, withHour, withMinute, withSecond, withNano</code>	Returns a new ZonedDateTime, with one temporal unit changed to the given value.
<code>withZoneSameInstant, withZoneSameLocal</code>	Returns a new ZonedDateTime in the given time zone, either representing the same instant or the same local time.
<code>getDayOfMonth</code>	Gets the day of the month (between 1 and 31).
<code>getDayOfYear</code>	Gets the day of the year (between 1 and 366).
<code>getDayOfWeek</code>	Gets the day of the week, returning a value of the DayOfWeek enumeration.
<code>getMonth, getMonthValue</code>	Gets the month as a value of the Month enumeration, or as a number between 1 and 12.
<code>getYear</code>	Gets the year, between -999,999,999 and 999,999,999.
<code>getHour, getMinute, getSecond, getNano</code>	Gets the hour, minute, second, or nanosecond of this ZonedDateTime.
<code>getOffset</code>	Gets the offset from UTC, as a ZoneOffset instance. Offsets can vary from -12:00 to +14:00. Some time zones have fractional offsets. Offsets change with daylight savings time.
<code>toLocalDate, toLocalTime, toInstant</code>	Yields the local date or local time, or the corresponding instant.
<code>isBefore, isAfter</code>	Compares this ZonedDateTime with another.



CAUTION: There is also an `OffsetDateTime` class that represents times with an offset from UTC, but without time zone rules. That class is intended for specialized applications that specifically require the absence of those rules, such as certain network protocols. For human time, use `ZonedDateTime`.

The example program in Listing 6.3 demonstrates the `ZonedDateTime` class.

Listing 6.3 zonedtimes/ZonedDateTime.java

```
1 package zonedtimes;
2
3 import java.time.*;
4
5 public class ZonedDateTime
6 {
7     public static void main(String[] args)
8     {
9         ZonedDateTime apollo11Launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,
10             ZoneId.of("America/New_York"));
11         // 1969-07-16T09:32-04:00[America/New_York]
12         System.out.println("apollo11Launch: " + apollo11Launch);
13
14         Instant instant = apollo11Launch.toInstant();
15         System.out.println("instant: " + instant);
16
17         ZonedDateTime zonedDateTime = instant.atZone(ZoneId.of("UTC"));
18         System.out.println("zonedDateTime: " + zonedDateTime);
19
20         ZonedDateTime skipped = ZonedDateTime.of(LocalDate.of(2013, 3, 31),
21             LocalTime.of(2, 30), ZoneId.of("Europe/Berlin"));
22         // Constructs March 31 3:30
23         System.out.println("skipped: " + skipped);
24
25         ZonedDateTime ambiguous = ZonedDateTime.of(LocalDate.of(2013, 10, 27),
26             // End of daylight savings time
27             LocalTime.of(2, 30), ZoneId.of("Europe/Berlin"));
28         // 2013-10-27T02:30+02:00[Europe/Berlin]
29         ZonedDateTime anHourLater = ambiguous.plusHours(1);
30         // 2013-10-27T02:30+01:00[Europe/Berlin]
31         System.out.println("ambiguous: " + ambiguous);
32         System.out.println("anHourLater: " + anHourLater);
33
34         ZonedDateTime meeting = ZonedDateTime.of(LocalDate.of(2013, 10, 31),
35             LocalTime.of(14, 30), ZoneId.of("America/Los_Angeles"));
36         System.out.println("meeting: " + meeting);
37         ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));
```

```
38     // Caution! Won't work with daylight savings time
39     System.out.println("nextMeeting: " + nextMeeting);
40     nextMeeting = meeting.plus(Period.ofDays(7)); // OK
41     System.out.println("nextMeeting: " + nextMeeting);
42 }
43 }
```

6.6 Formatting and Parsing

The `DateTimeFormatter` class provides three kinds of formatters to print a date/time value:

- Predefined standard formatters (see Table 6.6)
- Locale-specific formatters
- Formatters with custom patterns

To use one of the standard formatters, simply call its `format` method:

```
String formatted = DateTimeFormatter.ISO_OFFSET_DATE_TIME.format(apollo11Launch);
// 1969-07-16T09:32:00-04:00"
```

The standard formatters are mostly intended for machine-readable timestamps. To present dates and times to human readers, use a locale-specific formatter. There are four styles, `SHORT`, `MEDIUM`, `LONG`, and `FULL`, for both date and time—see Table 6.7.

The static methods `ofLocalizedDate`, `ofLocalizedTime`, and `ofLocalDateTime` create such a formatter. For example:

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalDateTime(FormatStyle.LONG);
String formatted = formatter.format(apollo11Launch);
// July 16, 1969 9:32:00 AM EDT
```

These methods use the default locale. To change to a different locale, simply use the `withLocale` method.

```
formatted = formatter.withLocale(Locale.FRENCH).format(apollo11Launch);
// 16 juillet 1969 09:32:00 EDT
```

The `DayOfWeek` and `Month` enumerations have methods `getDisplayName` for giving the names of weekdays and months in different locales and formats.

```
for (DayOfWeek w : DayOfWeek.values())
    System.out.print(w.getDisplayName(TextStyle.SHORT, Locale.ENGLISH) + " ");
// Prints Mon Tue Wed Thu Fri Sat Sun
```

See Chapter 7 for more information about locales.

Table 6.6 Predefined Formatters

Formatter	Description	Example
BASIC_ISO_DATE	Year, month, day, zone offset without separators	19690716-0500
ISO_LOCAL_DATE, ISO_LOCAL_TIME, ISO_LOCAL_DATE_TIME	Sepators -, :, T	1969-07-16, 09:32:00, 1969-07-16T09:32:00
ISO_OFFSET_DATE, ISO_OFFSET_TIME, ISO_OFFSET_DATE_TIME	Like ISO_LOCAL_XXX, but with zone offset	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00
ISO_ZONED_DATE_TIME	With zone offset and zone ID	1969-07-16T09:32:00-05:00[America/New_York]
ISO_INSTANT	In UTC, denoted by the Z zone ID	1969-07-16T14:32:00Z
ISO_DATE, ISO_TIME, ISO_DATE_TIME	Like ISO_OFFSET_DATE, ISO_OFFSET_TIME, and ISO_ZONED_DATE_TIME, but the zone information is optional	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00[America/New_York]
ISO_ORDINAL_DATE	The year and day of year, for LocalDate	1969-197
ISO_WEEK_DATE	The year, week, and day of week, for LocalDate	1969-W29-3
RFC_1123_DATE_TIME	The standard for email timestamps, codified in RFC 822 and updated to four digits for the year in RFC 1123	Wed, 16 Jul 1969 09:32:00 -0500

Table 6.7 Locale-Specific Formatting Styles

Style	Date	Time
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT



NOTE: The `java.time.format.DateTimeFormatter` class is intended as a replacement for `java.util.DateFormat`. If you need an instance of the latter for backward compatibility, call `formatter.toFormat()`.

Finally, you can roll your own date format by specifying a pattern. For example,

```
formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
```

formats a date in the form `Wed 1969-07-16 09:32`. Each letter denotes a different time field, and the number of times the letter is repeated selects a particular format, according to rules that are arcane and seem to have organically grown over time. Table 6.8 shows the most useful pattern elements.

Table 6.8 Commonly Used Formatting Symbols for Date/Time Formats

ChronoField or Purpose	Examples
ERA	G: AD, GGGG: Anno Domini, GGGGG: A
YEAR_OF_ERA	yy: 69, yyyy: 1969
MONTH_OF_YEAR	M: 7, MM: 07, MMM: Jul, MMMM: July, MMMMM: J
DAY_OF_MONTH	d: 6, dd: 06
DAY_OF_WEEK	e: 3, E: Wed, EEEE: Wednesday, EEEEE: W
HOUR_OF_DAY	H: 9, HH: 09
CLOCK_HOUR_OF_AM_PM	K: 9, KK: 09
AMPM_OF_DAY	a: AM
MINUTE_OF_HOUR	mm: 02
SECOND_OF_MINUTE	ss: 00
NANO_OF_SECOND	nnnnnn: 000000
Time zone ID	VV: America/New_York
Time zone name	z: EDT, zzzz: Eastern Daylight Time
Zone offset	x: -04, xx: -0400, xxx: -04:00, XXX: same, but use Z for zero
Localized zone offset	O: GMT-4, OOOOO: GMT-04:00

To parse a date/time value from a string, use one of the static `parse` methods. For example,

```
LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
ZonedDateTime apollo11Launch =
```

```
ZonedDateTime.parse("1969-07-16 03:32:00-0400",
    DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
```

The first call uses the standard ISO_LOCAL_DATE formatter, the second one a custom formatter.

The program in Listing 6.4 shows how to format and parse dates and times.

Listing 6.4 *formatting/Formatting.java*

```
1 package formatting;
2
3 import java.time.*;
4 import java.time.format.*;
5 import java.util.*;
6
7 public class Formatting
8 {
9     public static void main(String[] args)
10    {
11        ZonedDateTime apollo11Launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,
12            ZoneId.of("America/New_York"));
13
14        String formatted = DateTimeFormatter.ISO_OFFSET_DATE_TIME.format(apollo11Launch);
15        // 1969-07-16T09:32:00-04:00
16        System.out.println(formatted);
17
18        DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);
19        formatted = formatter.format(apollo11Launch);
20        // July 16, 1969 9:32:00 AM EDT
21        System.out.println(formatted);
22        formatted = formatter.withLocale(Locale.FRENCH).format(apollo11Launch);
23        // 16 juillet 1969 09:32:00 EDT
24        System.out.println(formatted);
25
26        formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
27        formatted = formatter.format(apollo11Launch);
28        System.out.println(formatted);
29
30        LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
31        System.out.println("churchsBirthday: " + churchsBirthday);
32        apollo11Launch = ZonedDateTime.parse("1969-07-16 03:32:00-0400",
33            DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
34        System.out.println("apollo11Launch: " + apollo11Launch);
35
36        for (DayOfWeek w : DayOfWeek.values())
37            System.out.print(w.getDisplayName(TextStyle.SHORT, Locale.ENGLISH)
38                + " ");
39    }
40 }
```

6.7 Interoperating with Legacy Code

As a brand-new creation, the Java Date and Time API will have to interoperate with existing classes—in particular, the ubiquitous `java.util.Date`, `java.util.GregorianCalendar`, and `java.sql.Date/Time/Timestamp`.

The `Instant` class is a close analog to `java.util.Date`. In Java SE 8, that class has two added methods: the `toInstant` method that converts a `Date` to an `Instant`, and the static `from` method that converts in the other direction.

Similarly, `ZonedDateTime` is a close analog to `java.util.GregorianCalendar`, and that class has gained conversion methods in Java SE 8. The `toZonedDateTime` method converts a `GregorianCalendar` to a `ZonedDateTime`, and the static `from` method does the opposite conversion.

Another set of conversions is available for the date and time classes in the `java.sql` package. You can also pass a `DateTimeFormatter` to legacy code that uses `java.text.Format`. Table 6.9 summarizes these conversions.

Table 6.9 Conversions between `java.time` Classes and Legacy Classes

Classes	To Legacy Class	From Legacy Class
Instant ↔ <code>java.util.Date</code>	<code>Date.from(instant)</code>	<code>date.toInstant()</code>
<code>ZonedDateTime</code> ↔ <code>java.util.GregorianCalendar</code>	<code>GregorianCalendar.from(zonedDateTime)</code>	<code>cal.toZonedDateTime()</code>
Instant ↔ <code>java.sql.Timestamp</code>	<code>TimeStamp.from(instant)</code>	<code>timestamp.toInstant()</code>
<code>LocalDateTime</code> ↔ <code>java.sql.Timestamp</code>	<code>Timestamp.valueOf(localDateTime)</code>	<code>timeStamp.toLocalDateTime()</code>
<code>LocalDate</code> ↔ <code>java.sql.Date</code>	<code>Date.valueOf(localDate)</code>	<code>date.toLocalDate()</code>
<code>LocalTime</code> ↔ <code>java.sql.Time</code>	<code>Time.valueOf(localTime)</code>	<code>time.toLocalTime()</code>
<code>DateTimeFormatter</code> → <code>java.text.DateFormat</code>	<code>formatter.toFormat()</code>	None
<code>java.util.TimeZone</code> → <code>ZoneId</code>	<code>Timezone.getTimeZone(id)</code>	<code>timeZone.toZoneId()</code>
<code>java.nio.file.attribute.FileTime</code> → <code>Instant</code>	<code>FileTime.from(instant)</code>	<code>fileTime.toInstant()</code>

You now know how to use the Java 8 date and time library to work with date and time values around the world. The next chapter takes the discussion of programming for an international audience further. You will see how to format program messages, numbers, and currencies in the way that makes sense for your customers, wherever they may be.

Internationalization

In this chapter

- 7.1 Locales, page 372
- 7.2 Number Formats, page 378
- 7.3 Currencies, page 384
- 7.4 Date and Time, page 385
- 7.5 Collation and Normalization, page 393
- 7.6 Message Formatting, page 400
- 7.7 Text Input and Output, page 404
- 7.8 Resource Bundles, page 408
- 7.9 A Complete Example, page 413

There's a big world out there; we hope that lots of its inhabitants will be interested in your software. The Internet, after all, effortlessly spans the barriers between countries. On the other hand, when you pay no attention to an international audience, *you* are putting up a barrier.

The Java programming language was the first language designed from the ground up to support internationalization. From the beginning, it had the one essential feature needed for effective internationalization: It used Unicode for all strings. Unicode support makes it easy to write Java programs that manipulate strings in any one of multiple languages.

Many programmers believe that all they need to do to internationalize their application is to support Unicode and to translate the messages in the user interface. However, as this chapter demonstrates, there is a lot more to internationalizing programs than just Unicode support. Dates, times, currencies, even numbers are formatted differently in different parts of the world. You need an easy way to configure menu and button names, message strings, and keyboard shortcuts for different languages.

In this chapter, we will show you how to write internationalized Java programs and how to localize dates, times, numbers, text, and GUIs. We will show you the tools that Java offers for writing internationalized programs. We will close this chapter with a complete example—a retirement calculator with a user interface in English, German, and Chinese.

7.1 Locales

When you look at an application that is adapted to an international market, the most obvious difference you notice is the language. This observation is actually a bit too limiting for true internationalization, since countries can share a common language, but you might still need to do some work to make computer users of both countries happy. As Oscar Wilde famously said: “We have really everything in common with America nowadays, except, of course, language.”

In all cases, menus, button labels, and program messages will need to be translated to the local language. They might also need to be rendered in a different script. There are many more subtle differences; for example, numbers are formatted quite differently in English and in German. The number

123,456.78

should be displayed as

123.456,78

for a German user—that is, the roles of the decimal point and the decimal comma separator are reversed. There are similar variations in the display of dates. In the United States, dates are somewhat irrationally displayed as month/day/year. Germany uses the more sensible order of day/month/year, whereas in China, the usage is year/month/day. Thus, the date

3/22/61

should be presented as

22.03.1961

to a German user. Of course, if the month names are written out explicitly, the difference in languages becomes apparent. The English

March 22, 1961

should be presented as

22. März 1961

in German, or

1961年3月22日

in Chinese.

There are several formatter classes that take these differences into account. To control the formatting, use the `Locale` class. A *locale* is made up of up to five components:

1. A language, specified by two or three lowercase letters, such as `en` (English), `de` (German), or `zh` (Chinese). Table 7.1 shows common codes.
2. Optionally, a script, specified by four letters with an initial uppercase, such as `Latn` (Latin), `Cyril` (Cyrillic), or `Hant` (traditional Chinese characters). This can be useful because some languages, such as Serbian, are written in Latin or Cyrillic, and some Chinese readers prefer the traditional over the simplified characters.
3. Optionally, a country or region, specified by two uppercase letters or three digits, such as `US` (United States) or `CH` (Switzerland). Table 7.2 shows common codes.
4. Optionally, a variant, specifying miscellaneous features such as dialects or spelling rules. Variants are rarely used nowadays. There used to be a “Nynorsk” variant of Norwegian, but it is now expressed with a different language code, `nn`. What used to be variants for the Japanese imperial calendar and Thai numerals are now expressed as extensions (see the next item).
5. Optionally, an extension. Extensions describe local preferences for calendars (such as the Japanese calendar), numbers (Thai instead of Western digits), and so on. The Unicode standard specifies some of these extensions. Extensions start with `u-` and a two-letter code specifying whether the extension deals with the calendar (`ca`), numbers (`nu`), and so on. For example, the extension `u-nu-thai` denotes the use of Thai numerals. Other extensions are entirely arbitrary and start with `x-`, such as `x-java`.

Rules for locales are formulated in the “Best Current Practices” memo BCP 47 of the Internet Engineering Task Force (<http://tools.ietf.org/html/bcp47>). You can find a more accessible summary at www.w3.org/International/articles/language-tags.

Table 7.1 Common ISO 639-1 Language Codes

Language	Code	Language	Code
Chinese	zh	Italian	it
Danish	da	Japanese	ja
Dutch	nl	Korean	ko
English	en	Norwegian	no
French	fr	Portuguese	pt
Finnish	fi	Spanish	es
German	de	Swedish	sv
Greek	el	Turkish	tr

Table 7.2 Common ISO 3166-1 Country Codes

Country	Code	Country	Code
Austria	AT	Japan	JP
Belgium	BE	Korea	KR
Canada	CA	The Netherlands	NL
China	CN	Norway	NO
Denmark	DK	Portugal	PT
Finland	FI	Spain	ES
Germany	DE	Sweden	SE
Great Britain	GB	Switzerland	CH
Greece	GR	Taiwan	TW
Ireland	IE	Turkey	TR
Italy	IT	United States	US

The codes for languages and countries seem a bit random because some of them are derived from local languages. German in German is Deutsch, Chinese in Chinese is zhongwen: hence de and zh. And Switzerland is CH, deriving from the Latin term *Confoederatio Helvetica* for the Swiss confederation.

Locales are described by tags—hyphenated strings of locale elements such as en-US.

In Germany, you would use a locale de-DE. Switzerland has four official languages (German, French, Italian, and Rhaeto-Romance). A German speaker in Switzerland would want to use a locale de-CH. This locale uses the rules for the German language, but currency values are expressed in Swiss francs, not euros.

If you only specify the language, say, de, then the locale cannot be used for country-specific issues such as currencies.

You can construct a `Locale` object from a tag string like this:

```
Locale usEnglish = Locale.forLanguageTag("en-US");
```

The `toLanguageTag` method yields the language tag for a given locale. For example, `Locale.US.toLanguageTag()` is the string "en-US"

For your convenience, there are predefined locale objects for various countries:

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
Locale.PRC
Locale.TAIWAN
Locale.UK
Locale.US
```

A number of predefined locales specify just a language without a location:

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

Finally, the static `getAvailableLocales` method returns an array of all locales known to the virtual machine.

Besides constructing a locale or using a predefined one, you have two other methods for obtaining a locale object.

The static `getDefault` method of the `Locale` class initially gets the default locale as stored by the local operating system. You can change the default Java locale by calling `setDefault`; however, that change only affects your program, not the operating system.

All locale-dependent utility classes can return an array of the locales they support. For example,

```
Locale[] supportedLocales = NumberFormat.getAvailableLocales();
```

returns all locales that the `NumberFormat` class can handle.



TIP: If you want to test a locale that just has language and country settings, you can supply them on the command line when you launch your program. For example, here we set the default locale to de-CH:

```
java -Duser.language=de -Duser.region=CH MyProgram
```

Once you have a locale, what can you do with it? Not much, as it turns out. The only useful methods in the `Locale` class are those for identifying the language and country codes. The most important one is `getDisplayName`. It returns a string describing the locale. This string does not contain the cryptic two-letter codes, but is in a form that can be presented to a user, such as

German (Switzerland)

Actually, there is a problem here. The display name is issued in the default locale. That might not be appropriate. If your user already selected German as the preferred language, you probably want to present the string in German. You can do just that by giving the German locale as a parameter. The code

```
Locale loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

prints

Deutsch (Schweiz)

This example shows why you need `Locale` objects. You feed them to locale-aware methods that produce text that is presented to users in different locations. You will see many examples of this in the following sections.

java.util.Locale 1.1

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`

constructs a locale with the given language, country, and variant. Don't use variants in new code—use the IETF BCP 47 language tags instead.
- `static Locale forLanguageTag(String languageTag) 7`

constructs a locale corresponding to the given language tag.
- `static Locale getDefault()`

returns the default locale.
- `static void setDefault(Locale loc)`

sets the default locale.
- `String getDisplayName()`

returns a name describing the locale, expressed in the current locale.
- `String getDisplayName(Locale loc)`

returns a name describing the locale, expressed in the given locale.
- `String getLanguage()`

returns the language code, a lowercase two-letter ISO 639 code.
- `String getDisplayLanguage()`

returns the name of the language, expressed in the current locale.
- `String getDisplayLanguage(Locale loc)`

returns the name of the language, expressed in the given locale.
- `String getCountry()`

returns the country code as an uppercase two-letter ISO 3166 code.
- `String getDisplayCountry()`

returns the name of the country, expressed in the current locale.
- `String getDisplayCountry(Locale loc)`

returns the name of the country, expressed in the given locale.
- `String toLanguageTag() 7`

returns the IETF BCP 47 language tag for this locale, e.g., "de-CH".
- `String toString()`

returns a description of the locale, with the language and country separated by underscores (e.g., "de_CH"). Use this method only for debugging.

7.2 Number Formats

We already mentioned how number and currency formatting is highly locale-dependent. The Java library supplies a collection of formatter objects that can format and parse numeric values in the `java.text` package. Go through the following steps to format a number for a particular locale:

1. Get the locale object, as described in the preceding section.
2. Use a “factory method” to obtain a formatter object.
3. Use the formatter object for formatting and parsing.

The factory methods are static methods of the `NumberFormat` class that take a `Locale` argument. There are three factory methods: `getNumberInstance`, `getCurrencyInstance`, and `getPercentInstance`. These methods return objects that can format and parse numbers, currency amounts, and percentages, respectively. For example, here is how you can format a currency value in German:

```
Locale loc = Locale.GERMAN;
NumberFormat currFmt = NumberFormat.getCurrencyInstance(loc);
double amt = 123456.78;
String result = currFmt.format(amt);
```

The result is

123.456,78 €

Note that the currency symbol is € and that it is placed at the end of the string. Also, note the reversal of decimal points and decimal commas.

Conversely, to read in a number that was entered or stored with the conventions of a certain locale, use the `parse` method. For example, the following code parses the value that the user typed into a text field. The `parse` method can deal with decimal points and commas, as well as digits in other languages.

```
TextField inputField;
.
.
.
NumberFormat fmt = NumberFormat.getNumberInstance();
// get the number formatter for default locale
Number input = fmt.parse(inputField.getText().trim());
double x = input.doubleValue();
```

The return type of `parse` is the abstract type `Number`. The returned object is either a `Double` or a `Long` wrapper object, depending on whether the parsed number was a floating-point number. If you don’t care about the distinction, you can simply use the `doubleValue` method of the `Number` class to retrieve the wrapped number.



CAUTION: Objects of type `Number` are not automatically unboxed—you cannot simply assign a `Number` object to a primitive type. Instead, use the `doubleValue` or `intValue` method.

If the text for the number is not in the correct form, the method throws a `ParseException`. For example, leading whitespace in the string is *not* allowed. (Call `trim` to remove it.) However, any characters that follow the number in the string are simply ignored, so no exception is thrown.

Note that the classes returned by the `getXXXInstance` factory methods are not actually of type `NumberFormat`. The `NumberFormat` type is an abstract class, and the actual formatters belong to one of its subclasses. The factory methods merely know how to locate the object that belongs to a particular locale.

You can get a list of the currently supported locales with the static `getAvailableLocales` method. That method returns an array of the locales for which number formatter objects can be obtained.

The sample program for this section lets you experiment with number formatters (see Figure 7.1). The combo box at the top of the figure contains all locales with number formatters. You can choose between number, currency, and percentage formatters. Each time you make another choice, the number in the text field is reformatted. If you go through a few locales, you can get a good impression of the many ways that a number or currency value can be formatted. You can also type a different number and click the Parse button to call the `parse` method, which tries to parse what you entered. If your input is successfully parsed, it is passed to `format` and the result is displayed. If parsing fails, then a “Parse error” message is displayed in the text field.



Figure 7.1 The `NumberFormatTest` program

The code, shown in Listing 7.1, is fairly straightforward. In the constructor, we call `NumberFormat.getAvailableLocales`. For each locale, we call `getDisplayName` and fill a combo box with the strings that the `getDisplayName` method returns. (The strings are not sorted; we tackle this issue in Section 7.5, “Collation and Normalization,” on p. 393.) Whenever the user selects another locale or clicks one of the radio buttons,

we create a new formatter object and update the text field. When the user clicks the Parse button, we call the parse method to do the actual parsing, based on the locale selected.

NOTE: You can use a Scanner to read localized integers and floating-point numbers. Call the useLocale method to set the locale.

Listing 7.1 numberFormat/NumberFormatTest.java

```
1 package numberFormat;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.text.*;
6 import java.util.*;
7
8 import javax.swing.*;
9
10 /**
11 * This program demonstrates formatting numbers under various locales.
12 * @version 1.14 2016-05-06
13 * @author Cay Horstmann
14 */
15 public class NumberFormatTest
16 {
17     public static void main(String[] args)
18     {
19         EventQueue.invokeLater(() ->
20             {
21                 JFrame frame = new NumberFormatFrame();
22                 frame.setTitle("NumberFormatTest");
23                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24                 frame.setVisible(true);
25             });
26     }
27 }
28
29 /**
30 * This frame contains radio buttons to select a number format, a combo box to pick a locale, a
31 * text field to display a formatted number, and a button to parse the text field contents.
32 */
33 class NumberFormatFrame extends JFrame
34 {
35     private Locale[] locales;
36     private double currentNumber;
37     private JComboBox<String> localeCombo = new JComboBox<>();
38     private JButton parseButton = new JButton("Parse");
```

```
39  private JTextField numberText = new JTextField(30);
40  private JRadioButton numberRadioButton = new JRadioButton("Number");
41  private JRadioButton currencyRadioButton = new JRadioButton("Currency");
42  private JRadioButton percentRadioButton = new JRadioButton("Percent");
43  private ButtonGroup rbGroup = new ButtonGroup();
44  private NumberFormat currentNumberFormat;
45
46  public NumberFormatFrame()
47  {
48      setLayout(new GridBagLayout());
49
50      ActionListener listener = event -> updateDisplay();
51
52      JPanel p = new JPanel();
53      addRadioButton(p, numberRadioButton, rbGroup, listener);
54      addRadioButton(p, currencyRadioButton, rbGroup, listener);
55      addRadioButton(p, percentRadioButton, rbGroup, listener);
56
57      add(new JLabel("Locale:"), new GBC(0, 0).setAnchor(GBC.EAST));
58      add(p, new GBC(1, 1));
59      add(parseButton, new GBC(0, 2).setInsets(2));
60      add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
61      add(numberText, new GBC(1, 2).setFill(GBC.HORIZONTAL));
62      locales = (Locale[]) NumberFormat.getAvailableLocales().clone();
63      Arrays.sort(locales, Comparator.comparing(Locale::getDisplayName));
64      for (Locale loc : locales)
65          localeCombo.addItem(loc.getDisplayName());
66      localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
67      currentNumber = 123456.78;
68      updateDisplay();
69
70      localeCombo.addActionListener(listener);
71
72      parseButton.addActionListener(event ->
73      {
74          String s = numberText.getText().trim();
75          try
76          {
77              Number n = currentNumberFormat.parse(s);
78              if (n != null)
79              {
80                  currentNumber = n.doubleValue();
81                  updateDisplay();
82              }
83              else
84              {
85                  numberText.setText("Parse error: " + s);
86              }
87      }
```

(Continues)

Listing 7.1 (*Continued*)

```
88         catch (ParseException e)
89         {
90             numberText.setText("Parse error: " + s);
91         }
92     });
93     pack();
94 }
95
96 /**
97 * Adds a radio button to a container.
98 * @param p the container into which to place the button
99 * @param b the button
100 * @param g the button group
101 * @param listener the button listener
102 */
103 public void addRadioButton(Container p, JRadioButton b, ButtonGroup g, ActionListener listener)
104 {
105     b.setSelected(g.getButtonCount() == 0);
106     b.addActionListener(listener);
107     g.add(b);
108     p.add(b);
109 }
110
111 /**
112 * Updates the display and formats the number according to the user settings.
113 */
114 public void updateDisplay()
115 {
116     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
117     currentNumberFormat = null;
118     if (numberRadioButton.isSelected())
119         currentNumberFormat = NumberFormat.getNumberInstance(currentLocale);
120     else if (currencyRadioButton.isSelected())
121         currentNumberFormat = NumberFormat.getCurrencyInstance(currentLocale);
122     else if (percentRadioButton.isSelected())
123         currentNumberFormat = NumberFormat.getPercentInstance(currentLocale);
124     String formatted = currentNumberFormat.format(currentNumber);
125     numberText.setText(formatted);
126 }
127 }
```

java.text.NumberFormat 1.1

- static Locale[] getAvailableLocales()
- static NumberFormat getInstance()
- static NumberFormat getInstance(Locale l)
- static NumberFormat getCurrencyInstance()
- static NumberFormat getCurrencyInstance(Locale l)
- static NumberFormat getPercentInstance()
- static NumberFormat getPercentInstance(Locale l)

returns a formatter for numbers, currency amounts, or percentage values for the current locale or for the given locale.

- String format(double x)
- String format(long x)

returns the string resulting from formatting the given floating-point number or integer.

- Number parse(String s)

parses the given string and returns the number value, as a Double if the input string describes a floating-point number and as a Long otherwise. The beginning of the string must contain a number; no leading whitespace is allowed. The number can be followed by other characters, which are ignored. Throws ParseException if parsing was not successful.

- void setParseIntegerOnly(boolean b)
- boolean isParseIntegerOnly()

sets or gets a flag to indicate whether this formatter should parse only integer values.

- void setGroupingUsed(boolean b)
- boolean isGroupingUsed()

sets or gets a flag to indicate whether this formatter emits and recognizes decimal separators (such as 100,000).

- void setMinimumIntegerDigits(int n)
- int getMinimumIntegerDigits()
- void setMaximumIntegerDigits(int n)
- int getMaximumIntegerDigits()
- void setMinimumFractionDigits(int n)
- int getMinimumFractionDigits()
- void setMaximumFractionDigits(int n)
- int getMaximumFractionDigits()

sets or gets the maximum or minimum number of digits allowed in the integer or fractional part of a number.

7.3 Currencies

To format a currency value, you can use the `NumberFormat.getCurrencyInstance` method. However, that method is not very flexible—it returns a formatter for a single currency. Suppose you prepare an invoice for an American customer in which some amounts are in dollars and others are in euros. You can't just use two formatters

```
NumberFormat dollarFormatter = NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance(Locale.GERMANY);
```

Your invoice would look very strange, with some values formatted like \$100,000 and others like 100.000 €. (Note that the euro value uses a decimal point, not a comma.)

Instead, use the `Currency` class to control the currency used by the formatters. You can get a `Currency` object by passing a currency identifier to the static `Currency.getInstance` method. Then call the `setCurrency` method for each formatter. Here is how you would set up the euro formatter for your American customer:

```
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance(Locale.US);
euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

The currency identifiers are defined by ISO 4217 (see www.currency-iso.org/iso_index/iso_tables/iso_tables_a1.htm). Table 7.3 provides a partial list.

Table 7.3 Currency Identifiers

Currency Value	Identifier
U.S. Dollar	USD
Euro	EUR
British Pound	GBP
Japanese Yen	JPY
Chinese Renminbi (Yuan)	CNY
Indian Rupee	INR
Russian Ruble	RUB

java.util.Currency 1.4

- static Currency getInstance(String currencyCode)
- static Currency getInstance(Locale locale)

returns the Currency instance for the given ISO 4217 currency code or the country of the given locale.

- String toString()
- String getCurrencyCode()

gets the ISO 4217 currency code of this currency.

- String getSymbol()
- String getSymbol(Locale locale)

gets the formatting symbol of this currency for the default locale or the given locale.

For example, the symbol for USD can be "\$" or "US\$", depending on the locale.

- int getDefaultFractionDigits()

gets the default number of fraction digits of this currency.

- static Set<Currency> getAvailableCurrencies() [7](#)

gets all available currencies.

7.4 Date and Time

When you are formatting date and time, you should be concerned with four locale-dependent issues:

- The names of months and weekdays should be presented in the local language.
- There will be local preferences for the order of year, month, and day.
- The Gregorian calendar might not be the local preference for expressing dates.
- The time zone of the location must be taken into account.

The `DateTimeFormatter` class from the `java.time` package handles these issues. Pick one of the formatting styles shown in Tables 7.4. Then get a formatter:

```
FormatStyle style = . . .; // One of FormatStyle.SHORT, FormatStyle.MEDIUM, . . .
DateTimeFormatter dateFormatter = DateTimeFormatter.ofLocalizedDate(style);
DateTimeFormatter timeFormatter = DateTimeFormatter.ofLocalizedTime(style);
DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofLocalizedDateTime(style);
// or DateTimeFormatter.ofLocalizedDateTime(style1, style2)
```

These formatters use the current locale. To use a different locale, use the `withLocale` method:

```
DateTimeFormatter dateFormatter =
    DateTimeFormatter.ofLocalizedDate(style).withLocale(locale);
```

Now you can format a `LocalDate`, `LocalDateTime`, `LocalTime`, or `ZonedDateTime`:

```
ZonedDateTime appointment = . . .;
String formatted = formatter.format(appointment);
```

Table 7.4 Date and Time Formatting Styles

Style	Date	Time
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT in en-US, 9:32:00 MSZ in de-DE (only for ZonedDateTime)
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT in en-US, 9:32 Uhr MSZ in de-DE (only for ZonedDateTime)

NOTE: Here we use the `DateTimeFormatter` class from the `java.time` package. There is also a legacy `java.text.SimpleDateFormat` class from Java 1.1 that works with `Date` and `Calendar` objects.

You can use one of the static `parse` methods of `LocalDate`, `LocalDateTime`, `LocalTime`, or `ZonedDateTime` to parse a date or time in a string:

```
LocalTime time = LocalTime.parse("9:32 AM", formatter);
```

These methods are not suitable for parsing human input, at least not without preprocessing. For example, the short time formatter for the United States will parse "9:32 AM" but not "9:32AM" or "9:32 am".



CAUTION: Date formatters parse nonexistent dates, such as November 31, and adjust them to the last date in the given month.

Sometimes, you need to display just the names of weekdays and months, for example, in a calendar application. Call the `getDisplayName` method of the `DayOfWeek` and `Month` enumerations.

```
for (Month m : Month.values())
    System.out.println(m.getDisplayName(textStyle, locale) + " ");
```

Tables 7.5 shows the text styles. The STANDALONE versions are for display outside a formatted date. For example, in Finnish, January is “tammikuuta” inside a date, but “tammikuu” standalone.

Table 7.5 Values of the `java.time.format.TextStyle` Enumeration

Style	Example
FULL / FULL_STANDALONE	January
SHORT / SHORT_STANDALONE	Jan
NARROW / NARROW_STANDALONE	J

NOTE: The first day of the week can be Saturday, Sunday, or Monday, depending on the locale. You can obtain it like this:

```
DayOfWeek first = WeekFields.of(locale).getFirstDayOfWeek();
```

Listing 7.2 shows the `DateFormat` class in action. You can select a locale and see how the date and time are formatted in different places around the world.

Figure 7.2 shows the program (after Chinese fonts were installed). As you can see, it correctly displays the output.

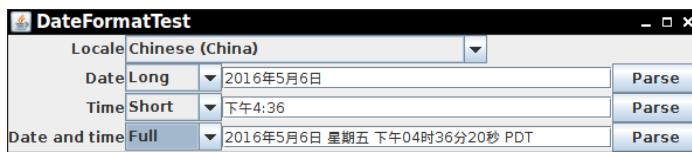


Figure 7.2 The `DateFormatTest` program

You can also experiment with parsing. Enter a date, time, or date/time and click the Parse button.

We use a helper class `EnumCombo` to solve a technical problem (see Listing 7.3). We wanted to fill a combo with values such as `Short`, `Medium`, and `Long` and then automatically convert the user’s selection to values `FormatStyle.SHORT`, `FormatStyle.MEDIUM`, and `FormatStyle.LONG`. Instead of writing repetitive code, we use reflection: We convert the user’s choice to upper case, replace all spaces with underscores, and then find the value of the static field with that name. (See Volume I, Chapter 5 for more details about reflection.)

Listing 7.2 dateFormat/DateTimeFormatterTest.java

```
1 package dateFormat;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.time.*;
6 import java.time.format.*;
7 import java.util.*;
8
9 import javax.swing.*;
10
11 /**
12 * This program demonstrates formatting dates under various locales.
13 * @version 1.00 2016-05-06
14 * @author Cay Horstmann
15 */
16 public class DateTimeFormatterTest
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(() ->
21         {
22             JFrame frame = new DateTimeFormatterFrame();
23             frame.setTitle("DateFormatTest");
24             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25             frame.setVisible(true);
26         });
27     }
28 }
29
30 /**
31 * This frame contains combo boxes to pick a locale, date and time formats, text fields to display
32 * formatted date and time, buttons to parse the text field contents, and a "lenient" check box.
33 */
34 class DateTimeFormatterFrame extends JFrame
35 {
36     private Locale[] locales;
37     private LocalDate currentDate;
38     private LocalTime currentTime;
39     private ZonedDateTime currentDateTime;
40     private DateTimeFormatter currentDateFormat;
41     private DateTimeFormatter currentTimeFormat;
42     private DateTimeFormatter currentDateTimeFormat;
43     private JComboBox<String> localeCombo = new JComboBox<>();
44     private JButton dateParseButton = new JButton("Parse");
45     private JButton timeParseButton = new JButton("Parse");
46     private JButton dateTimeParseButton = new JButton("Parse");
47     private JTextField dateText = new JTextField(30);
48     private JTextField timeText = new JTextField(30);
```

```
49 private JTextField dateText = new JTextField(30);
50 private EnumCombo<FormatStyle> dateStyleCombo = new EnumCombo<>(FormatStyle.class,
51     "Short", "Medium", "Long", "Full");
52 private EnumCombo<FormatStyle> timeStyleCombo = new EnumCombo<>(FormatStyle.class,
53     "Short", "Medium");
54 private EnumCombo<FormatStyle> dateTimeStyleCombo = new EnumCombo<>(FormatStyle.class,
55     "Short", "Medium", "Long", "Full");
56
57 public DateTimeFormatterFrame()
58 {
59     setLayout(new GridBagLayout());
60     add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
61     add(localeCombo, new GBC(1, 0, 2, 1).setAnchor(GBC.WEST));
62
63     add(new JLabel("Date"), new GBC(0, 1).setAnchor(GBC.EAST));
64     add(dateStyleCombo, new GBC(1, 1).setAnchor(GBC.WEST));
65     add(dateText, new GBC(2, 1, 2, 1).setFill(GBC.HORIZONTAL));
66     add(dateParseButton, new GBC(4, 1).setAnchor(GBC.WEST));
67
68     add(new JLabel("Time"), new GBC(0, 2).setAnchor(GBC.EAST));
69     add(timeStyleCombo, new GBC(1, 2).setAnchor(GBC.WEST));
70     add(timeText, new GBC(2, 2, 2, 1).setFill(GBC.HORIZONTAL));
71     add(timeParseButton, new GBC(4, 2).setAnchor(GBC.WEST));
72
73     add(new JLabel("Date and time"), new GBC(0, 3).setAnchor(GBC.EAST));
74     add(dateTimeStyleCombo, new GBC(1, 3).setAnchor(GBC.WEST));
75     add(dateTimeText, new GBC(2, 3, 2, 1).setFill(GBC.HORIZONTAL));
76     add(dateTimeParseButton, new GBC(4, 3).setAnchor(GBC.WEST));
77
78     locales = (Locale[]) Locale.getAvailableLocales().clone();
79     Arrays.sort(locales, Comparator.comparing(Locale::getDisplayName));
80     for (Locale loc : locales)
81         localeCombo.addItem(loc.getDisplayName());
82     localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
83     currentDate = LocalDate.now();
84     currentTime = LocalTime.now();
85     currentDateTime = ZonedDateTime.now();
86     updateDisplay();
87
88     ActionListener listener = event -> updateDisplay();
89
90     localeCombo.addActionListener(listener);
91     dateStyleCombo.addActionListener(listener);
92     timeStyleCombo.addActionListener(listener);
93     dateTimeStyleCombo.addActionListener(listener);
94
95     dateParseButton.addActionListener(event ->
96     {
97         String d = dateText.getText().trim();
```

(Continues)

Listing 7.2 (*Continued*)

```
98         try
99         {
100             currentDate = LocalDate.parse(d, currentDateFormat);
101             updateDisplay();
102         }
103         catch (Exception e)
104         {
105             dateText.setText(e.getMessage());
106         }
107     });
108
109     timeParseButton.addActionListener(event ->
110     {
111         String t = timeText.getText().trim();
112         try
113         {
114             currentTime = LocalTime.parse(t, currentTimeFormat);
115             updateDisplay();
116         }
117         catch (Exception e)
118         {
119             timeText.setText(e.getMessage());
120         }
121     });
122
123     dateTimeParseButton.addActionListener(event ->
124     {
125         String t = dateTimeText.getText().trim();
126         try
127         {
128             currentDateTime = ZonedDateTime.parse(t, currentDateTimeFormat);
129             updateDisplay();
130         }
131         catch (Exception e)
132         {
133             dateTimeText.setText(e.getMessage());
134         }
135     });
136
137     pack();
138 }
139 /**
140 * Updates the display and formats the date according to the user settings.
141 */
142
```

```
143 public void updateDisplay()
144 {
145     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
146     FormatStyle dateStyle = dateStyleCombo.getValue();
147     currentDateFormat = DateTimeFormatter.ofLocalizedDate(
148         dateStyle).withLocale(currentLocale);
149     dateText.setText(currentDateFormat.format(currentDate));
150     FormatStyle timeStyle = timeStyleCombo.getValue();
151     currentTimeFormat = DateTimeFormatter.ofLocalizedTime(
152         timeStyle).withLocale(currentLocale);
153     timeText.setText(currentTimeFormat.format(currentTime));
154     FormatStyle dateTimeStyle = dateTimeStyleCombo.getValue();
155     currentDateTimeFormat = DateTimeFormatter.ofLocalizedDateTime(
156         dateTimeStyle).withLocale(currentLocale);
157     dateTimeText.setText(currentDateTimeFormat.format(currentDateTime));
158 }
159 }
```

Listing 7.3 dateFormat/EnumCombo.java

```
1 package dateFormat;
2
3 import java.util.*;
4 import javax.swing.*;
5
6 /**
7  * A combo box that lets users choose from among static field
8  * values whose names are given in the constructor.
9  * @version 1.15 2016-05-06
10 * @author Cay Horstmann
11 */
12 public class EnumCombo<T> extends JComboBox<String>
13 {
14     private Map<String, T> table = new TreeMap<>();
15
16     /**
17      * Constructs an EnumCombo yielding values of type T.
18      * @param cl a class
19      * @param labels an array of strings describing static field names
20      * of cl that have type T
21      */
22     public EnumCombo(Class<?> cl, String... labels)
23     {
24         for (String label : labels)
25         {
```

(Continues)

Listing 7.3 (Continued)

```
26     String name = label.toUpperCase().replace(' ', '_');
27     try
28     {
29         java.lang.reflect.Field f = cl.getField(name);
30         @SuppressWarnings("unchecked") T value = (T) f.get(cl);
31         table.put(label, value);
32     }
33     catch (Exception e)
34     {
35         label = "(" + label + ")";
36         table.put(label, null);
37     }
38     addItem(label);
39 }
40 setSelectedItem(labels[0]);
41 }
42 /**
43 * Returns the value of the field that the user selected.
44 * @return the static field value
45 */
46 public T getValue()
47 {
48     return table.get(getSelectedItem());
49 }
50 }
```

java.time.format.DateTimeFormatter 8

- static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)
- static DateTimeFormatter ofLocalizedTime(FormatStyle dateStyle)
- static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTextStyle)
- static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle, FormatStyle timeStyle)
returns DateTimeFormatter instances that format dates, times, or dates and times with the specified styles.
- DateTimeFormatter withLocale(Locale locale)
returns a copy of this formatter with the given locale.
- String format(TemporalAccessor temporal)
returns the string resulting from formatting the given date/time.

```
java.time.LocalDate 8  
java.time.LocalDateTime 8  
java.time.LocalTime 8  
java.time.ZonedDateTime 8
```

- static *Xxx* parse(CharSequence text, DateTimeFormatter formatter)

parses the given string and returns the LocalDate, LocalTime, LocalDateTime, or ZonedDateTime described in it. Throws a DateTimeParseException if parsing was not successful.

7.5 Collation and Normalization

Most programmers know how to compare strings with the `compareTo` method of the `String` class. Unfortunately, when interacting with human users, this method is not very useful. The `compareTo` method uses the values of the UTF-16 encoding of the string, which leads to absurd results, even in English. For example, the following five strings are ordered according to the `compareTo` method:

```
America  
Zulu  
able  
zebra  
Ångström
```

For dictionary ordering, you would want to consider upper case and lower case to be equivalent. To an English speaker, the sample list of words would be ordered as

```
able  
America  
Ångström  
zebra  
Zulu
```

However, that order would not be acceptable to a Swedish user. In Swedish, the letter Å is different from the letter A, and it is collated *after* the letter Z! That is, a Swedish user would want the words to be sorted as

```
able  
America  
zebra  
Zulu  
Ångström
```

To obtain a locale-sensitive comparator, call the static `Collator.getInstance` method:

```
Collator coll = Collator.getInstance(locale);
words.sort(coll); // Collator implements Comparator<Object>
```

Since the `Collator` class implements the `Comparator` interface, you can pass a `Collator` object to the `List.sort(Comparator)` method to sort a list of strings.

There are a couple of advanced settings for collators. You can set a collator's *strength* to select how selective it should be. Character differences are classified as *primary*, *secondary*, or *tertiary*. For example, in English, the difference between "A" and "Z" is considered primary, the difference between "A" and "Å" is secondary, and between "A" and "a" is tertiary.

By setting the strength of the collator to `Collator.PRIMARY`, you tell it to pay attention only to primary differences. By setting the strength to `Collator.SECONDARY`, you instruct the collator to take secondary differences into account. That is, two strings will be more likely to be considered different when the strength is set to "secondary" or "tertiary," as shown in Table 7.6.

Table 7.6 Collations with Different Strengths (English Locale)

Primary	Secondary	Tertiary
Angstrom = Ångström	Angstrom ≠ Ångström	Angstrom ≠ Ångström
Able = able	Able = able	Able ≠ able

When the strength has been set to `Collator.IDENTICAL`, no differences are allowed. This setting is mainly useful in conjunction with a rather technical collator setting, the *decomposition mode*, which we take up next.

Occasionally, a character or sequence of characters can be described in more than one way in Unicode. For example, an "Å" can be Unicode character U+00C5, or it can be expressed as a plain A (U+0065) followed by a ° ("combining ring above"; U+030A). Perhaps more surprisingly, the letter sequence "ffi" can be described with a single character "Latin small ligature ffi" with code U+FB03. (One could argue that this is a presentation issue that should not have resulted in different Unicode characters, but we don't make the rules.)

The Unicode standard defines four *normalization forms* (D, KD, C, and KC) for strings. See www.unicode.org/unicode/reports/tr15/tr15-23.html for the details. In the normalization form C, accented characters are always composed. For example, a sequence of A and a combining ring above ° is combined into a single character Å. In form D, accented characters are always decomposed into their base letters and combining accents: Å is turned into A followed by °. Forms KC and KD also decompose characters such as ligatures or the trademark symbol.

You can choose the degree of normalization that you want a collator to use. The value `Collator.NO_DECOMPOSITION` does not normalize strings at all. This option is faster, but it might not be appropriate for text that expresses characters in multiple forms. The default, `Collator.CANONICAL_DECOMPOSITION`, uses the normalization form D. This is useful for text that contains accents but not ligatures. Finally, “full decomposition” uses normalization form KD. See Table 7.7 for examples.

Table 7.7 Differences between Decomposition Modes

No Decomposition	Canonical Decomposition	Full Decomposition
$\text{Å} \neq \text{A}^\circ$	$\text{Å} = \text{A}^\circ$	$\text{Å} = \text{A}^\circ$
$\text{™} \neq \text{TM}$	$\text{™} \neq \text{TM}$	$\text{™} = \text{TM}$

It is wasteful to have the collator decompose a string many times. If one string is compared many times against other strings, you can save the decomposition in a *collation key* object. The `getCollationKey` method returns a `CollationKey` object that you can use for further, faster comparisons. Here is an example:

```
String a = . . .;
CollationKey aKey = coll.getCollationKey(a);
if(aKey.compareTo(coll.getCollationKey(b)) == 0) // fast comparison
    . . .
```

Finally, you might want to convert strings into their normalized forms even when you don’t do collation—for example, when storing strings in a database or communicating with another program. The `java.text.Normalizer` class carries out the normalization process. For example,

```
String name = "Ångström";
String normalized = Normalizer.normalize(name, Normalizer.Form.NFD); // uses normalization form D
```

The normalized string contains ten characters. The “Å” and “ö” are replaced by “A°” and “o°” sequences.

However, that is not usually the best form for storage and transmission. Normalization form C first applies decomposition and then combines the accents back in a standardized order. According to the W3C, this is the recommended mode for transferring data over the Internet.

The program in Listing 7.4 lets you experiment with collation order. Type a word into the text field and click the Add button to add it to the list of words. Each time you add another word, or change the locale, strength, or decomposition mode, the list of words is sorted again. An = sign indicates words that are considered identical (see Figure 7.3).



Figure 7.3 The CollationTest program

The locale names in the combo box are displayed in sorted order, using the collator of the default locale. If you run this program with the US English locale, note that “Norwegian (Norway,Nynorsk)” comes before “Norwegian (Norway)”, even though the Unicode value of the comma character is greater than the Unicode value of the closing parenthesis.

Listing 7.4 `collation/CollationTest.java`

```
1 package collation;  
2  
3 import java.awt.*;  
4 import java.awt.event.*;  
5 import java.text.*;  
6 import java.util.*;  
7 import java.util.List;  
8  
9 import javax.swing.*;  
10  
11 /**  
12 * This program demonstrates collating strings under various locales.  
13 * @version 1.15 2016-05-06
```

```
14  * @author Cay Horstmann
15  */
16 public class CollationTest
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(() ->
21         {
22             JFrame frame = new CollationFrame();
23             frame.setTitle("CollationTest");
24             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25             frame.setVisible(true);
26         });
27     }
28 }
29 /**
30 * This frame contains combo boxes to pick a locale, collation strength and decomposition rules,
31 * a text field and button to add new strings, and a text area to list the collated strings.
32 */
33
34 class CollationFrame extends JFrame
35 {
36     private Collator collator = Collator.getInstance(Locale.getDefault());
37     private List<String> strings = new ArrayList<>();
38     private Collator currentCollator;
39     private Locale[] locales;
40     private JComboBox<String> localeCombo = new JComboBox<>();
41     private JTextField newWord = new JTextField(20);
42     private JTextArea sortedWords = new JTextArea(20, 20);
43     private JButton addButton = new JButton("Add");
44     private EnumCombo<Integer> strengthCombo = new EnumCombo<>(Collator.class, "Primary",
45             "Secondary", "Tertiary", "Identical");
46     private EnumCombo<Integer> decompositionCombo = new EnumCombo<>(Collator.class,
47             "Canonical Decomposition", "Full Decomposition", "No Decomposition");
48
49     public CollationFrame()
50     {
51         setLayout(new GridBagLayout());
52         add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
53         add(new JLabel("Strength"), new GBC(0, 1).setAnchor(GBC.EAST));
54         add(new JLabel("Decomposition"), new GBC(0, 2).setAnchor(GBC.EAST));
55         add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
56         add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
57         add(strengthCombo, new GBC(1, 1).setAnchor(GBC.WEST));
58         add(decompositionCombo, new GBC(1, 2).setAnchor(GBC.WEST));
59         add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
60         add(new JScrollPane(sortedWords), new GBC(0, 4, 2, 1).setFill(GBC.BOTH));
61
62         locales = (Locale[]) Collator.getAvailableLocales().clone();
63     }
64 }
```

(Continues)

Listing 7.4 (Continued)

```
63     Arrays.sort(
64         locales, (l1, l2) -> collator.compare(l1.getDisplayName(), l2.getDisplayName()));
65     for (Locale loc : locales)
66         localeCombo.addItem(loc.getDisplayName());
67     localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
68
69     strings.add("America");
70     strings.add("able");
71     strings.add("Zulu");
72     strings.add("zebra");
73     strings.add("\u00C5ngstr\u00F6m");
74     strings.add("A\u030angstro\u0308m");
75     strings.add("Angstrom");
76     strings.add("Able");
77     strings.add("office");
78     strings.add("o\uFB03ce");
79     strings.add("Java\u2122");
80     strings.add("JavaTM");
81     updateDisplay();
82
83     addButton.addActionListener(event ->
84     {
85         strings.add(newWord.getText());
86         updateDisplay();
87     });
88
89     ActionListener listener = event -> updateDisplay();
90
91     localeCombo.addActionListener(listener);
92     strengthCombo.addActionListener(listener);
93     decompositionCombo.addActionListener(listener);
94     pack();
95 }
96
97 /**
98 * Updates the display and collates the strings according to the user settings.
99 */
100 public void updateDisplay()
101 {
102     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
103     localeCombo.setLocale(currentLocale);
104
105     currentCollator = Collator.getInstance(currentLocale);
106     currentCollator.setStrength(strengthCombo.getValue());
107     currentCollator.setDecomposition(decompositionCombo.getValue());
108 }
```

```
109     Collections.sort(strings, currentCollator);
110
111     sortedWords.setText("");
112     for (int i = 0; i < strings.size(); i++)
113     {
114         String s = strings.get(i);
115         if (i > 0 && currentCollator.compare(s, strings.get(i - 1)) == 0)
116             sortedWords.append(" ");
117         sortedWords.append(s + "\n");
118     }
119     pack();
120 }
121 }
```

java.text.Collator 1.1

- static Locale[] getAvailableLocales()
returns an array of Locale objects for which Collator objects are available.
- static Collator getInstance()
- static Collator getInstance(Locale l)
returns a collator for the default locale or the given locale.
- int compare(String a, String b)
returns a negative value if a comes before b, 0 if they are considered identical, and a positive value otherwise.
- boolean equals(String a, String b)
returns true if a and b are considered identical, false otherwise.
- void setStrength(int strength)
- int getStrength()
sets or gets the strength of the collator. Stronger collators tell more words apart. Strength values are Collator.PRIMARY, Collator.SECONDARY, and Collator.TERTIARY.
- void setDecomposition(int decomp)
- int getDecompositon()
sets or gets the decomposition mode of the collator. The more a collator decomposes a string, the more strict it will be in deciding whether two strings should be considered identical. Decomposition values are Collator.NO_DECOMPOSITION, Collator.CANONICAL_DECOMPOSITION, and Collator.FULL_DECOMPOSITION.
- CollationKey getCollationKey(String a)
returns a collation key that contains a decomposition of the characters in a form that can be quickly compared against another collation key.

java.text.CollationKey 1.1

- int compareTo(CollationKey b)

returns a negative value if this key comes before b, 0 if they are considered identical, and a positive value otherwise.

java.text.Normalizer 6

- static String normalize(CharSequence str, Normalizer.Form form)

returns the normalized form of str. The form value is one of ND, NKD, NC, or NKC.

7.6 Message Formatting

The Java library has a `MessageFormat` class that formats text with variable parts. It is similar to formatting with the `printf` method, but it supports locales and formats for numbers and dates. We will examine that mechanism in the following sections.

7.6.1 Formatting Numbers and Dates

Here is a typical message format string:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

The numbers in braces are placeholders for actual names and values. The static method `MessageFormat.format` lets you substitute values for the variables. It is a “varargs” method, so you can simply supply the parameters as follows:

```
String msg = MessageFormat.format("On {2}, a {0} destroyed {1} houses and caused {3} of damage.",  
    "hurricane", 99, new GregorianCalendar(1999, 0, 1).getTime(), 10.0E8);
```

In this example, the placeholder `{0}` is replaced with “hurricane”, `{1}` is replaced with 99, and so on.

The result of our example is the string

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses and caused 100,000,000 of damage.
```

That is a start, but it is not perfect. We don’t want to display the time “12:00 AM,” and we want the damage amount printed as a currency value. The way we do this is by supplying an optional format for some of the placeholders:

```
"On {2,date,long}, a {0} destroyed {1} houses and caused {3,number,currency} of damage."
```

This example code prints:

On January 1, 1999, a hurricane destroyed 99 houses and caused \$100,000,000 of damage.

In general, the placeholder index can be followed by a *type* and a *style*. Separate the index, type, and style by commas. The type can be any of

number
time
date
choice

If the type is `number`, then the style can be

integer
currency
percent

or it can be a number format pattern such as `$,##0`. (See the documentation of the `DecimalFormat` class for more information about the possible formats.)

If the type is either `time` or `date`, then the style can be

short
medium
long
full

or a date format pattern such as `yyyy-MM-dd`. (See the documentation of the `SimpleDateFormat` class for more information about the possible formats.)



CAUTION: The static `MessageFormat.format` method uses the current locale to format the values. To format with an arbitrary locale, you have to work a bit harder because there is no “varargs” method that you can use. You need to place the values to be formatted into an `Object[]` array, like this:

```
MessageFormat mf = new MessageFormat(pattern, loc);
String msg = mf.format(new Object[] { values });
```

java.text.MessageFormat 1.1

- `MessageFormat(String pattern)`
- `MessageFormat(String pattern, Locale loc)`

constructs a message format object with the specified pattern and locale.

- `void applyPattern(String pattern)`

sets the pattern of a message format object to the specified pattern.

(Continues)

java.text.MessageFormat 1.1 (Continued)

- void setLocale(Locale loc)
- Locale getLocale()

sets or gets the locale to be used for the placeholders in the message. The locale is *only* used for subsequent patterns that you set by calling the `applyPattern` method.

- static String format(String pattern, Object... args)
formats the pattern string by using `args[i]` as input for placeholder `{i}`.
- StringBuffer format(Object args, StringBuffer result, FieldPosition pos)
formats the pattern of this `MessageFormat`. The `args` parameter must be an array of objects. The formatted string is appended to `result`, and `result` is returned. If `pos` equals `new FieldPosition(MessageFormat.Field.ARGUMENT)`, its `beginIndex` and `endIndex` properties are set to the location of the text that replaces the `{1}` placeholder. Supply `null` if you are not interested in position information.

java.text.Format 1.1

- String format(Object obj)

formats the given object, according to the rules of this formatter. This method calls `format(obj, new StringBuffer(), new FieldPosition(1)).toString()`.

7.6.2 Choice Formats

Let's look closer at the pattern of the preceding section:

"On {2}, a {0} destroyed {1} houses and caused {3} of damage."

If we replace the disaster placeholder `{0}` with "earthquake", the sentence is not grammatically correct in English:

On January 1, 1999, a earthquake destroyed . . .

What we really want to do is integrate the article "a" into the placeholder:

"On {2}, {0} destroyed {1} houses and caused {3} of damage."

The `{0}` would then be replaced with "a hurricane" or "an earthquake". That is especially appropriate if this message needs to be translated into a language where the gender of a word affects the article. For example, in German, the pattern would be

"{0} zerstörte am {2} {1} Häuser und richtete einen Schaden von {3} an."

The placeholder would then be replaced with the grammatically correct combination of article and noun, such as "Ein Wirbelsturm" or "Eine Naturkatastrophe".

Now let us turn to the {1} parameter. If the disaster wasn't all that catastrophic, {1} might be replaced with the number 1, and the message would read:

On January 1, 1999, a mudslide destroyed 1 houses and . . .

Ideally, we would like the message to vary according to the placeholder value, so it would read

```
no houses
one house
2 houses
. . .
```

depending on the placeholder value. The choice formatting option was designed for this purpose.

A choice format is a sequence of pairs, each containing

- A *lower limit*
- A *format string*

The lower limit and format string are separated by a # character, and the pairs are separated by | characters.

For example,

```
{1,choice,0#no houses|1#one house|2#{1} houses}
```

Table 7.8 shows the effect of this format string for various values of {1}.

Table 7.8 String Formatted by Choice Format

{1}	Result	{1}	Result
0	"no houses"	3	"3 houses"
1	"one house"	-1	"no houses"

Why do we use {1} twice in the format string? When the message format applies the choice format to the {1} placeholder and the value is 2, the choice format returns "{1} houses". That string is then formatted again by the message format, and the answer is spliced into the result.

NOTE: This example shows that the designer of the choice format was a bit muddleheaded. If you have three format strings, you need two limits to separate them. In general, you need *one fewer limit* than you have format strings. As you saw in Table 7.8, the `MessageFormat` class ignores the first limit.

The syntax would have been a lot clearer if the designer of this class realized that the limits belong *between* the choices, such as

```
no houses|1|one house|2|{1} houses // not the actual format
```

You can use the `<` symbol to denote that a choice should be selected if the lower bound is strictly less than the value.

You can also use the `≤` symbol (expressed as the Unicode character code `\u2264`) as a synonym for `#`. If you like, you can even specify a lower bound of $-∞$ as `-\u221E` for the first value.

For example,

```
-∞<no houses|0<one house|2≤{1} houses
```

or, using Unicode escapes,

```
-\u221E<no houses|0<one house|2\u2264{1} houses
```

Let's finish our natural disaster scenario. If we put the choice string inside the original message string, we get the following format instruction:

```
String pattern = "On {2,date,long}, {0} destroyed {1,choice,0#no houses|1#one house|2#{1}  
houses}" + " and caused {3,number,currency} of damage.;"
```

Or, in German,

```
String pattern = "{0} zerstörte am {2,date,long} {1,choice,0#kein Haus|1#ein Haus|2#{1} Häuser}"  
+ " und richtete einen Schaden von {3,number,currency} an.;"
```

Note that the ordering of the words is different in German, but the array of objects you pass to the `format` method is the *same*. The order of the placeholders in the format string takes care of the changes in the word ordering.

7.7 Text Input and Output

As you know, the Java programming language itself is fully Unicode-based. However, Windows and Mac OS X still support legacy character encodings such as Windows-1252 or Mac Roman in Western European countries, or Big5 in Taiwan. Therefore, communicating with your users through text is not as simple as

it should be. The following sections discuss the complications that you may encounter.

7.7.1 Text Files

Nowadays, it is best to use UTF-8 for saving and loading text files. But you may need to work with legacy files. If you know the expected character encoding, you can specify it when writing or reading text files:

```
PrintWriter out = new PrintWriter(filename, "Windows-1252");
```

For a guess of the best encoding to use, get the “platform encoding” by calling

```
Charset platformEncoding = Charset.defaultCharset();
```

7.7.2 Line Endings

This isn’t an issue of locales but of platforms. On Windows, text files are expected to use `\r\n` at the end of each line, where UNIX-based systems only require a `\n` character. Nowadays, most Windows programs can deal with just a `\n`. The notable exception is Notepad. If it is important to you that users can double-click on a text file that your application produces and view it in Notepad, make sure that the text file has proper line endings.

Any line written with the `println` method will be properly terminated. The only problem is if you print strings that contain `\n` characters. They are not automatically modified to the platform line ending.

Instead of using `\n` in strings, you can use `printf` and the `%n` format specifier to produce platform-dependent line endings. For example,

```
out.printf("Hello%nWorld%n");
```

produces

```
Hello\r\nWorld\r\n
```

on Windows and

```
Hello\nWorld\n
```

everywhere else.

7.7.3 The Console

If you write programs that communicate with the user through `System.in` / `System.out` or `System.console()`, you have to face the possibility that the console may use a character encoding that is different from the platform encoding reported by

`Charset.defaultCharset()`. This is particularly noticeable when working with the cmd shell on Windows. In the US version, the command shell uses the archaic IBM437 encoding that originated with IBM personal computers in 1982. There is no official API for revealing that information. The `Charset.defaultCharset()` method will return the Windows-1252 character set, which is quite different. For example, the euro symbol € is present in Windows-1252 but not in IBM437. When you call

```
System.out.println("100 €");
```

the console will display

```
100 ?
```

You can advise your users to switch the character encoding of the console. In Windows, that is achieved with the `chcp` command. For example,

```
chcp 1252
```

changes the console to the Windows-1252 code page.

Ideally, of course, your users should switch the console to UTF-8. In Windows, the command is

```
chcp 65001
```

Unfortunately, that is not enough to make Java use UTF-8 in the console. It is also necessary to set the platform encoding with the unofficial `file.encoding` system property:

```
java -Dfile.encoding=UTF-8 MyProg
```

7.7.4 Log Files

When log messages from the `java.util.logging` library are sent to the console, they are written with the console encoding. You saw how to control that in the preceding section. However, log messages in a file use a `FileHandler` which, by default, uses the platform encoding.

To change the encoding to UTF-8, you need to change the log manager settings. In the logging configuration file, set

```
java.util.logging.FileHandler.encoding=UTF-8
```

7.7.5 The UTF-8 Byte Order Mark

As already mentioned, it is a good idea to use UTF-8 for text files when you can. If your application has to read UTF-8 text files that were created by other programs, you run into another potential problem. It is perfectly legal to add a “byte order mark” character U+FEFF as the first character of a file. In the UTF-16

encoding, where each code unit is a two-byte quantity, the byte order mark tells a reader whether the file uses “big-endian” or “little-endian” byte ordering. UTF-8 is a single-byte encoding, so there is no need for specifying a byte ordering. But if a file starts with bytes 0xEF 0xBB 0xBF (the UTF-8 encoding of U+FEFF), that is a strong indication that it uses UTF-8. For that reason, the Unicode standard encourages this practice. Any reader is supposed to discard an initial byte order mark.

There is just one fly in the ointment. The Oracle Java implementation stubbornly refuses to follow the Unicode standard, citing potential compatibility issues. That means that you, the programmer, must do what the platform won’t do. When you read a text file and encounter a U+FEFF at the beginning, ignore it.



CAUTION: Unfortunately, the JDK implementors do not follow this advice. When you pass the `javac` compiler a valid UTF-8 source file that starts with a byte order mark, compilation fails with an error message “illegal character: \u00fe”.

7.7.6 Character Encoding of Source Files

You, the programmer, will need to communicate with the Java compiler—and you do that with tools on your local system. For example, you can use the Chinese version of Notepad to write your Java source code files. The resulting source code files are *not portable* because they use the local character encoding (GB or Big5, depending on which Chinese operating system you use). Only the compiled class files are portable—they will automatically use the “modified UTF-8” encoding for identifiers and strings. That means that when a program is compiling and running, three character encodings are involved:

- Source files: platform encoding
- Class files: modified UTF-8
- Virtual machine: UTF-16

(See Chapter 2 for a definition of the modified UTF-8 and UTF-16 formats.)



TIP: You can specify the character encoding of your source files with the `-encoding` flag, for example,

```
javac -encoding UTF-8 Myfile.java
```

To make your source files portable, restrict yourself to using the plain ASCII encoding. That is, change all non-ASCII characters to their equivalent Unicode escapes. For example, instead of using the string “Häuser”, use “H\u00f6user”. The JDK

contains a utility, `native2ascii`, that you can use to convert the native character encoding to plain ASCII. This utility simply replaces every non-ASCII character in the input with a \u followed by the four hex digits of the Unicode value. To use the `native2ascii` program, provide the input and output file names.

```
native2ascii Myfile.java Myfile.temp
```

You can convert the other way with the `-reverse` option:

```
native2ascii -reverse Myfile.temp Myfile.java
```

You can specify another encoding with the `-encoding` option.

```
native2ascii -encoding UTF-8 Myfile.java Myfile.temp
```

7.8 Resource Bundles

When localizing an application, you'll probably have a dauntingly large number of message strings, button labels, and so on, that all need to be translated. To make this task feasible, you'll want to define the message strings in an external location, usually called a *resource*. The person carrying out the translation can then simply edit the resource files without having to touch the source code of the program.

In Java, you can use property files to specify string resources, and you can implement classes for resources of other types.

NOTE: Java technology resources are not the same as Windows or Macintosh resources. A Macintosh or Windows executable program stores resources, such as menus, dialog boxes, icons, and messages, in a section separate from the program code. A resource editor can inspect and update these resources without affecting the program code.

NOTE: Volume I, Chapter 13 describes a concept of JAR file resources, whereby data files, sounds, and images can be placed in a JAR file. The `getResource` method of the class `Class` finds the file, opens it, and returns a URL to the resource. By placing the files into the JAR file, you leave the job of finding the files to the class loader, which already knows how to locate items in a JAR file. However, that mechanism has no locale support.

7.8.1 Locating Resource Bundles

When localizing an application, you produce a set of *resource bundles*. Each bundle is a property file or a class that describes locale-specific items (such as messages, labels, and so on). For each bundle, you have to provide versions for all locales that you want to support.

You need to use a specific naming convention for these bundles. For example, resources specific to Germany go into a file *bundleName_de_DE*, whereas those shared by all German-speaking countries go into *bundleName_de*. In general, use

bundleName_language_country

for all country-specific resources, and use

bundleName_language

for all language-specific resources. Finally, as a fallback, you can put defaults into a file without any suffix.

To load a bundle, use the command

```
ResourceBundle currentResources = ResourceBundle.getBundle(bundleName, currentLocale);
```

The `getBundle` method attempts to load the bundle that matches the current locale by language and country. If it is not successful, the country and the language are dropped in turn. Then the same search is applied to the default locale, and finally, the default bundle file is consulted. If even that attempt fails, the method throws a `MissingResourceException`.

That is, the `getBundle` method tries to load the following bundles:

```
bundleName_currentLocaleLanguage_currentLocaleCountry  
bundleName_currentLocaleLanguage  
bundleName_currentLocaleLanguage_defaultLocaleCountry  
bundleName_defaultLocaleLanguage  
bundleName
```

Once the `getBundle` method has located a bundle (say, *bundleName_de_DE*), it will still keep looking for *bundleName_de* and *bundleName*. If these bundles exist, they become the *parents* of the *bundleName_de_DE* bundle in a *resource hierarchy*. Later, when looking up a resource, the parents are searched if a lookup was not successful in the current bundle. That is, if a particular resource was not found in *bundleName_de_DE*, then the *bundleName_de* and *bundleName* will be queried as well.

This is clearly a very useful service—and one that would be tedious to program by hand. The resource bundle mechanism of the Java programming language

automatically locates the items that are the best match for a given locale. It is easy to add more and more localizations to an existing program—all you have to do is create additional resource bundles.

NOTE: We simplified the discussion of resource bundle lookup. If a locale has a script or variant, the lookup is quite a bit more complex. See the documentation of the method `ResourceBundle.Control.getCandidateLocales` for the gory details.



TIP: You need not place all resources for your application into a single bundle. You could have one bundle for button labels, one for error messages, and so on.

7.8.2 Property Files

Internationalizing strings is quite straightforward. You place all your strings into a property file such as `MyProgramStrings.properties`. This is simply a text file with one key/value pair per line. A typical file would look like this:

```
computeButton=Rechnen  
colorName=black  
defaultPaperSize=210x297
```

Then you name your property files as described in the preceding section, for example:

```
MyProgramStrings.properties  
MyProgramStrings_en.properties  
MyProgramStrings_de_DE.properties
```

You can load the bundle simply as

```
 ResourceBundle bundle = ResourceBundle.getBundle("MyProgramStrings", locale);
```

To look up a specific string, call

```
 String computeButtonLabel = bundle.getString("computeButton");
```



CAUTION: Files for storing properties are always ASCII files. If you need to place a Unicode character into a property file, encode it using the \uxxxx encoding. For example, to specify "colorName=Grün", use

```
colorName=Gr\u00FCn
```

You can use the `native2ascii` tool to generate these files.

7.8.3 Bundle Classes

To provide resources that are not strings, define classes that extend the `ResourceBundle` class. Use the standard naming convention to name your classes, for example

```
MyProgramResources.java  
MyProgramResources_en.java  
MyProgramResources_de_DE.java
```

Load the class with the same `getBundle` method that you use to load a property file:

```
 ResourceBundle bundle = ResourceBundle.getBundle("MyProgramResources", locale);
```



CAUTION: When searching for bundles, a bundle in a class is given preference over a property file when the two bundles have the same base names.

Each resource bundle class implements a lookup table. You need to provide a key string for each setting you want to localize, and use that key string to retrieve the setting. For example,

```
Color backgroundColor = (Color) bundle.getObject("backgroundColor");  
double[] paperSize = (double[]) bundle.getObject("defaultPaperSize");
```

The simplest way to implement resource bundle classes is to extend the `ListResourceBundle` class. The `ListResourceBundle` lets you place all your resources into an object array and then does the lookup for you. Follow this code outline:

```
public class bundleName_language_country extends ListResourceBundle  
{  
    private static final Object[][] contents =  
    {  
        { key1, value1 },  
        { key2, value2 },  
        ...  
    }  
    public Object[][] getContents() { return contents; }  
}
```

For example,

```
public class ProgramResources_de extends ListResourceBundle  
{  
    private static final Object[][] contents =  
    {  
        { "backgroundColor", Color.black },  
        { "defaultPaperSize", new double[] { 210, 297 } }  
    }  
    public Object[][] getContents() { return contents; }  
}
```

```
public class ProgramResources_en_US extends ListResourceBundle
{
    private static final Object[][] contents =
    {
        { "backgroundColor", Color.blue },
        { "defaultPaperSize", new double[] { 216, 279 } }
    }
    public Object[][] getContents() { return contents; }
}
```

NOTE: The paper sizes are given in millimeters. Everyone on the planet, with the exception of the United States and Canada, uses ISO 216 paper sizes. For more information, see www.cl.cam.ac.uk/~mgk25/iso-paper.html.

Alternatively, your resource bundle classes can extend the `ResourceBundle` class. Then you need to implement two methods, to enumerate all keys and to look up the value for a given key:

```
Enumeration<String> getKeys()
Object handleGetObject(String key)
```

The `getObjet` method of the `ResourceBundle` class calls the `handleGetObject` method that you supply.

java.util.ResourceBundle 1.1

- `static ResourceBundle getBundle(String baseName, Locale loc)`
- `static ResourceBundle getBundle(String baseName)`

loads the resource bundle class with the given name, for the given locale or the default locale, and its parent classes. If the resource bundle classes are located in a package, the base name must contain the full package name, such as `"intl.ProgramResources"`. The resource bundle classes must be `public` so that the `getBundle` method can access them.

- `Object getObject(String name)`

looks up an object from the resource bundle or its parents.

- `String getString(String name)`

looks up an object from the resource bundle or its parents and casts it as a string.

- `String[] getStringArray(String name)`

looks up an object from the resource bundle or its parents and casts it as a string array.

(Continues)

java.util.ResourceBundle 1.1 (Continued)

- `Enumeration<String> getKeys()`

returns an enumeration object to enumerate the keys of this resource bundle. It enumerates the keys in the parent bundles as well.

- `Object handleGetObject(String key)`

should be overridden to look up the resource value associated with the given key if you define your own resource lookup mechanism.

7.9 A Complete Example

In this section, we apply the material of this chapter to localize a retirement calculator. The program calculates whether or not you are saving enough money for your retirement. You enter your age, how much money you save every month, and so on (see Figure 7.4).

The text area and the graph show the balance of the retirement account for every year. If the numbers turn negative toward the later part of your life and the bars in the graph appear below the x axis, you need to do something—for example, save more money, postpone your retirement, die earlier, or be younger.

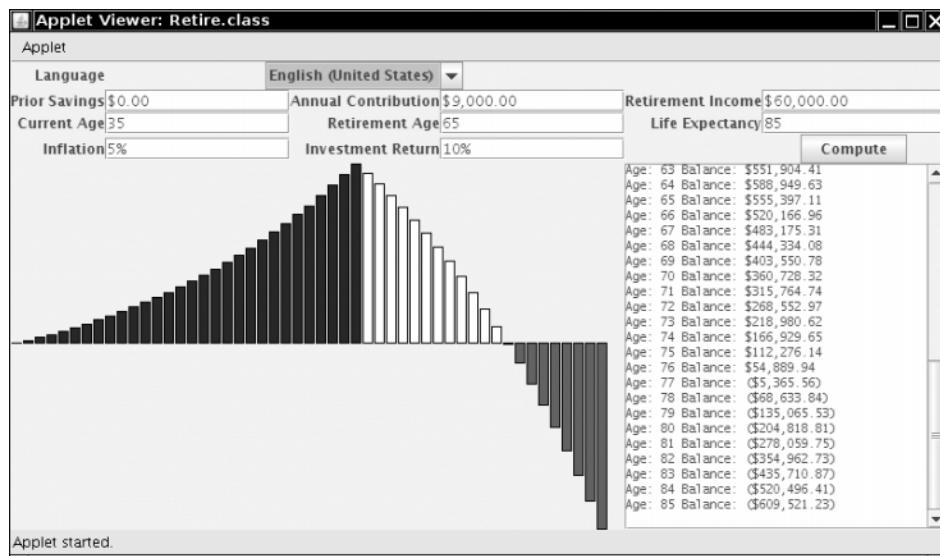


Figure 7.4 The retirement calculator in English

The retirement calculator works in three locales (English, German, and Chinese). Here are some of the highlights of the internationalization:

- The labels, buttons, and messages are translated into German and Chinese. You can find them in the classes `RetireResources_de` and `RetireResources_zh`. English is used as the fallback—see the `RetireResources` file. To generate the Chinese messages, we first typed the file, using Notepad running in Chinese Windows, and then we used the `native2ascii` utility to convert the characters to Unicode.
- Whenever the locale changes, we reset the labels and reformat the contents of the text fields.
- The text fields handle numbers, currency amounts, and percentages in the local format.
- The computation field uses a `MessageFormat`. The format string is stored in the resource bundle of each language.
- Just to show that it can be done, we use different colors for the bar graph, depending on the language chosen by the user.

Listings 7.5 through 7.8 show the code. Listings 7.9 through 7.11 are the property files for the localized strings. Figures 7.5 and 7.6 show the outputs in German and Chinese, respectively. To see Chinese characters, be sure you have Chinese fonts installed and configured with your Java runtime. Otherwise, Chinese characters will show up as “missing character” icons.

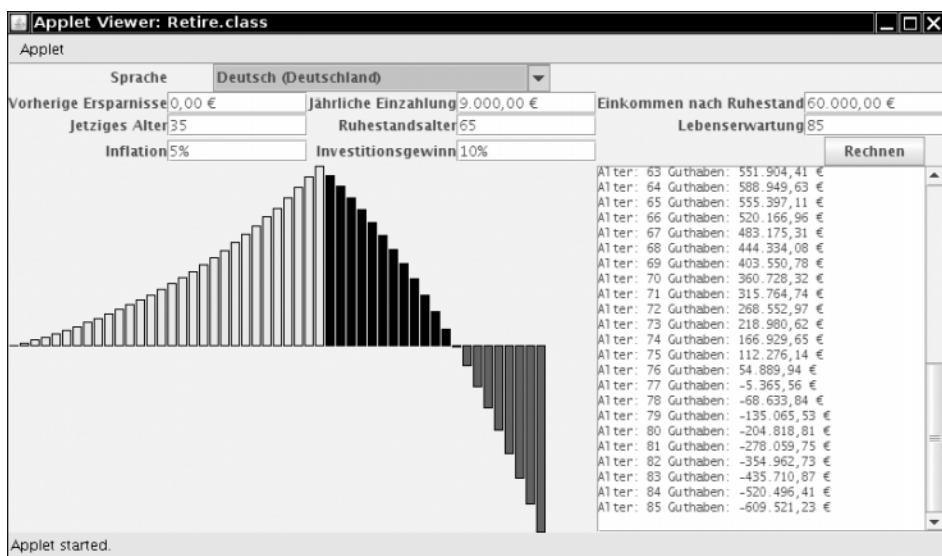


Figure 7.5 The retirement calculator in German

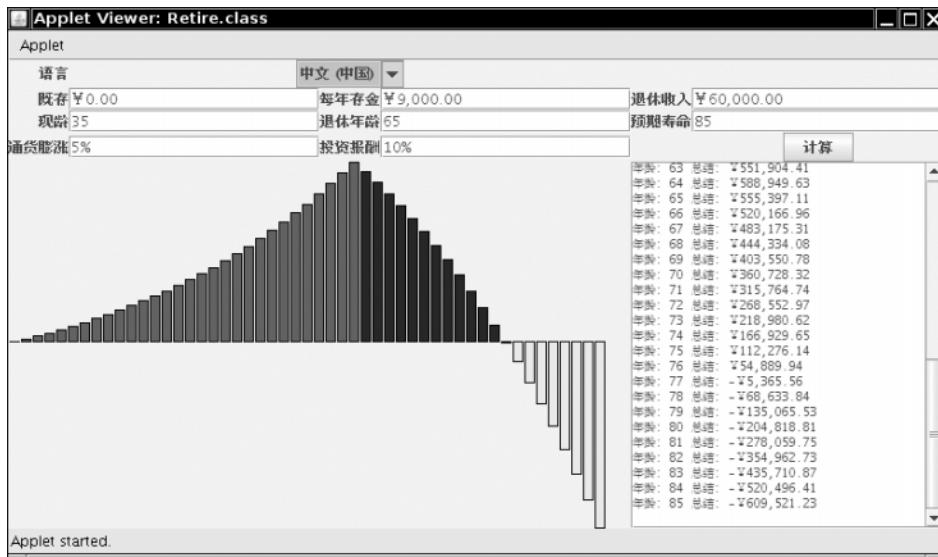


Figure 7.6 The retirement calculator in Chinese

Listing 7.5 retire/Retire.java

```

1 package retire;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.text.*;
6 import java.util.*;
7
8 import javax.swing.*;
9
10 /**
11 * This program shows a retirement calculator. The UI is displayed in English, German, and
12 * Chinese.
13 * @version 1.24 2016-05-06
14 * @author Cay Horstmann
15 */
16 public class Retire
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(() ->
21         {
22             JFrame frame = new RetireFrame();

```

(Continues)

Listing 7.5 (Continued)

```
23         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24         frame.setVisible(true);
25     });
26 }
27 }
28
29 class RetireFrame extends JFrame
30 {
31     private JTextField savingsField = new JTextField(10);
32     private JTextField contribField = new JTextField(10);
33     private JTextField incomeField = new JTextField(10);
34     private JTextField currentAgeField = new JTextField(4);
35     private JTextField retireAgeField = new JTextField(4);
36     private JTextField deathAgeField = new JTextField(4);
37     private JTextField inflationPercentField = new JTextField(6);
38     private JTextField investPercentField = new JTextField(6);
39     private JTextArea retireText = new JTextArea(10, 25);
40     private RetireComponent retireCanvas = new RetireComponent();
41     private JButton computeButton = new JButton();
42     private JLabel languageLabel = new JLabel();
43     private JLabel savingsLabel = new JLabel();
44     private JLabel contribLabel = new JLabel();
45     private JLabel incomeLabel = new JLabel();
46     private JLabel currentAgeLabel = new JLabel();
47     private JLabel retireAgeLabel = new JLabel();
48     private JLabel deathAgeLabel = new JLabel();
49     private JLabel inflationPercentLabel = new JLabel();
50     private JLabel investPercentLabel = new JLabel();
51     private RetireInfo info = new RetireInfo();
52     private Locale[] locales = { Locale.US, Locale.CHINA, Locale.GERMANY };
53     private Locale currentLocale;
54     private JComboBox<Locale> localeCombo = new LocaleCombo(locales);
55     private ResourceBundle res;
56     private ResourceBundle resStrings;
57     private NumberFormat currencyFmt;
58     private NumberFormat numberFmt;
59     private NumberFormat percentFmt;
60
61     public RetireFrame()
62     {
63         setLayout(new GridBagLayout());
64         add(languageLabel, new GBC(0, 0).setAnchor(GBC.EAST));
65         add(savingsLabel, new GBC(0, 1).setAnchor(GBC.EAST));
66         add(contribLabel, new GBC(2, 1).setAnchor(GBC.EAST));
67         add(incomeLabel, new GBC(4, 1).setAnchor(GBC.EAST));
68         add(currentAgeLabel, new GBC(0, 2).setAnchor(GBC.EAST));
69         add(retireAgeLabel, new GBC(2, 2).setAnchor(GBC.EAST));
70         add(deathAgeLabel, new GBC(4, 2).setAnchor(GBC.EAST));
```

```
71    add(inflationPercentLabel, new GBC(0, 3).setAnchor(GBC.EAST));
72    add(investPercentLabel, new GBC(2, 3).setAnchor(GBC.EAST));
73    add(localeCombo, new GBC(1, 0, 3, 1));
74    add(savingsField, new GBC(1, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
75    add(contribField, new GBC(3, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
76    add(incomeField, new GBC(5, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
77    add(currentAgeField, new GBC(1, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
78    add(retireAgeField, new GBC(3, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
79    add(deathAgeField, new GBC(5, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
80    add(inflationPercentField, new GBC(1, 3).setWeight(100, 0).setFill(GBC.HORIZONTAL));
81    add(investPercentField, new GBC(3, 3).setWeight(100, 0).setFill(GBC.HORIZONTAL));
82    add(retireCanvas, new GBC(0, 4, 4, 1).setWeight(100, 100).setFill(GBC.BOTH));
83    add(new JScrollPane(retireText), new GBC(4, 4, 2, 1).setWeight(0, 100).setFill(GBC.BOTH));
84
85    computeButton.setName("computeButton");
86    computeButton.addActionListener(event ->
87    {
88        getInfo();
89        updateData();
90        updateGraph();
91    });
92    add(computeButton, new GBC(5, 3));
93
94    retireText.setEditable(false);
95    retireText.setFont(new Font("Monospaced", Font.PLAIN, 10));
96
97    info.setSavings(0);
98    info.setContrib(9000);
99    info.setIncome(60000);
100   info.setCurrentAge(35);
101   info.setRetireAge(65);
102   info.setDeathAge(85);
103   info.setInvestPercent(0.1);
104   info.setInflationPercent(0.05);
105
106   int localeIndex = 0; // US locale is default selection
107   for (int i = 0; i < locales.length; i++)
108       // if current locale one of the choices, select it
109       if (getLocale().equals(locales[i])) localeIndex = i;
110   setCurrentLocale(locales[localeIndex]);
111
112   localeCombo.addActionListener(event ->
113   {
114       setCurrentLocale((Locale) localeCombo.getSelectedItem());
115       validate();
116   });
117   pack();
118 }
119
```

(Continues)

Listing 7.5 (Continued)

```
120  /**
121   * Sets the current locale.
122   * @param locale the desired locale
123   */
124  public void setCurrentLocale(Locale locale)
125  {
126      currentLocale = locale;
127      localeCombo.setLocale(currentLocale);
128      localeCombo.setSelectedItem(currentLocale);
129
130      res = ResourceBundle.getBundle("retire.RetireResources", currentLocale);
131      resStrings = ResourceBundle.getBundle("retire.RetireStrings", currentLocale);
132      currencyFmt = NumberFormat.getCurrencyInstance(currentLocale);
133      numberFmt = NumberFormat.getNumberInstance(currentLocale);
134      percentFmt = NumberFormat.getPercentInstance(currentLocale);
135
136      updateDisplay();
137      updateInfo();
138      updateData();
139      updateGraph();
140  }
141
142  /**
143   * Updates all labels in the display.
144   */
145  public void updateDisplay()
146  {
147      languageLabel.setText(resStrings.getString("language"));
148      savingsLabel.setText(resStrings.getString("savings"));
149      contribLabel.setText(resStrings.getString("contrib"));
150      incomeLabel.setText(resStrings.getString("income"));
151      currentAgeLabel.setText(resStrings.getString("currentAge"));
152      retireAgeLabel.setText(resStrings.getString("retireAge"));
153      deathAgeLabel.setText(resStrings.getString("deathAge"));
154      inflationPercentLabel.setText(resStrings.getString("inflationPercent"));
155      investPercentLabel.setText(resStrings.getString("investPercent"));
156      computeButton.setText(resStrings.getString("computeButton"));
157  }
158
159  /**
160   * Updates the information in the text fields.
161   */
162  public void updateInfo()
163  {
164      savingsField.setText(currencyFmt.format(info.getSavings()));
165      contribField.setText(currencyFmt.format(info.getContrib()));
166      incomeField.setText(currencyFmt.format(info.getIncome()));
167      currentAgeField.setText(numberFmt.format(info.getCurrentAge()));
```

```
168     retireAgeField.setText(numberFmt.format(info.getRetireAge()));
169     deathAgeField.setText(numberFmt.format(info.getDeathAge()));
170     investPercentField.setText(percentFmt.format(info.getInvestPercent()));
171     inflationPercentField.setText(percentFmt.format(info.getInflationPercent()));
172 }
173
174 /**
175 * Updates the data displayed in the text area.
176 */
177 public void updateData()
178 {
179     retireText.setText("");
180     MessageFormat retireMsg = new MessageFormat("");
181     retireMsg.setLocale(currentLocale);
182     retireMsg.applyPattern(resStrings.getString("retire"));

183     for (int i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
184     {
185         Object[] args = { i, info.getBalance(i) };
186         retireText.append(retireMsg.format(args) + "\n");
187     }
188 }
189
190 /**
191 * Updates the graph.
192 */
193 public void updateGraph()
194 {
195     retireCanvas.setColorPre((Color) res.getObject("colorPre"));
196     retireCanvas.setColorGain((Color) res.getObject("colorGain"));
197     retireCanvas.setColorLoss((Color) res.getObject("colorLoss"));
198     retireCanvas.setInfo(info);
199     repaint();
200 }
201
202 /**
203 * Reads the user input from the text fields.
204 */
205 public void getInfo()
206 {
207     try
208     {
209         info.setSavings(currencyFmt.parse(savingsField.getText()).doubleValue());
210         info.setContrib(currencyFmt.parse(contribField.getText()).doubleValue());
211         info.setIncome(currencyFmt.parse(incomeField.getText()).doubleValue());
212         info.setCurrentAge(numberFmt.parse(currentAgeField.getText()).intValue());
213         info.setRetireAge(numberFmt.parse(retireAgeField.getText()).intValue());
214         info.setDeathAge(numberFmt.parse(deathAgeField.getText()).intValue());
215         info.setInvestPercent(percentFmt.parse(investPercentField.getText()).doubleValue());
216     }
217 }
```

(Continues)

Listing 7.5 (Continued)

```
217         info.setInflationPercent(
218             percentFmt.parse(inflationPercentField.getText()).doubleValue());
219     }
220     catch (ParseException ex)
221     {
222         ex.printStackTrace();
223     }
224 }
225 /**
226 * The information required to compute retirement income data.
227 */
228 class RetireInfo
229 {
230     private double savings;
231     private double contrib;
232     private double income;
233     private int currentAge;
234     private int retireAge;
235     private int deathAge;
236     private double inflationPercent;
237     private double investPercent;
238     private int age;
239     private double balance;
240
241 /**
242 * Gets the available balance for a given year.
243 * @param year the year for which to compute the balance
244 * @return the amount of money available (or required) in that year
245 */
246 public double getBalance(int year)
247 {
248     if (year < currentAge) return 0;
249     else if (year == currentAge)
250     {
251         age = year;
252         balance = savings;
253         return balance;
254     }
255     else if (year == age) return balance;
256     if (year != age + 1) getBalance(year - 1);
257     age = year;
258     if (age < retireAge) balance += contrib;
259     else balance -= income;
260     balance = balance * (1 + (investPercent - inflationPercent));
261     return balance;
262 }
263 }
```

```
265
266 /**
267 * Gets the amount of prior savings.
268 * @return the savings amount
269 */
270 public double getSavings()
271 {
272     return savings;
273 }
274
275 /**
276 * Sets the amount of prior savings.
277 * @param newValue the savings amount
278 */
279 public void setSavings(double newValue)
280 {
281     savings = newValue;
282 }
283
284 /**
285 * Gets the annual contribution to the retirement account.
286 * @return the contribution amount
287 */
288 public double getContrib()
289 {
290     return contrib;
291 }
292
293 /**
294 * Sets the annual contribution to the retirement account.
295 * @param newValue the contribution amount
296 */
297 public void setContrib(double newValue)
298 {
299     contrib = newValue;
300 }
301
302 /**
303 * Gets the annual income.
304 * @return the income amount
305 */
306 public double getIncome()
307 {
308     return income;
309 }
310
311 /**
312 * Sets the annual income.
313 * @param newValue the income amount
```

(Continues)

Listing 7.5 (Continued)

```
314     */
315     public void setIncome(double newValue)
316     {
317         income = newValue;
318     }
319
320     /**
321      * Gets the current age.
322      * @return the age
323      */
324     public int getCurrentAge()
325     {
326         return currentAge;
327     }
328
329     /**
330      * Sets the current age.
331      * @param newValue the age
332      */
333     public void setCurrentAge(int newValue)
334     {
335         currentAge = newValue;
336     }
337
338     /**
339      * Gets the desired retirement age.
340      * @return the age
341      */
342     public int getRetireAge()
343     {
344         return retireAge;
345     }
346
347     /**
348      * Sets the desired retirement age.
349      * @param newValue the age
350      */
351     public void setRetireAge(int newValue)
352     {
353         retireAge = newValue;
354     }
355
356     /**
357      * Gets the expected age of death.
358      * @return the age
359      */
360     public int getDeathAge()
```

```
361     {
362         return deathAge;
363     }
364
365     /**
366      * Sets the expected age of death.
367      * @param newValue the age
368      */
369     public void setDeathAge(int newValue)
370     {
371         deathAge = newValue;
372     }
373
374     /**
375      * Gets the estimated percentage of inflation.
376      * @return the percentage
377      */
378     public double getInflationPercent()
379     {
380         return inflationPercent;
381     }
382
383     /**
384      * Sets the estimated percentage of inflation.
385      * @param newValue the percentage
386      */
387     public void setInflationPercent(double newValue)
388     {
389         inflationPercent = newValue;
390     }
391
392     /**
393      * Gets the estimated yield of the investment.
394      * @return the percentage
395      */
396     public double getInvestPercent()
397     {
398         return investPercent;
399     }
400
401     /**
402      * Sets the estimated yield of the investment.
403      * @param newValue the percentage
404      */
405     public void setInvestPercent(double newValue)
406     {
407         investPercent = newValue;
408     }
409 }
```

(Continues)

Listing 7.5 (Continued)

```
410 /**
411 * This component draws a graph of the investment result.
412 */
413
414 class RetireComponent extends JPanel
415 {
416     private static final int PANEL_WIDTH = 400;
417     private static final int PANEL_HEIGHT = 200;
418     private static final Dimension PREFERRED_SIZE = new Dimension(800, 600);
419     private RetireInfo info = null;
420     private Color colorPre;
421     private Color colorGain;
422     private Color colorLoss;
423
424     public RetireComponent()
425     {
426         setSize(PANEL_WIDTH, PANEL_HEIGHT);
427     }
428
429     /**
430      * Sets the retirement information to be plotted.
431      * @param newInfo the new retirement info
432      */
433     public void setInfo(RetireInfo newInfo)
434     {
435         info = newInfo;
436         repaint();
437     }
438
439     public void paintComponent(Graphics g)
440     {
441         Graphics2D g2 = (Graphics2D) g;
442         if (info == null) return;
443
444         double minValue = 0;
445         double maxValue = 0;
446         int i;
447         for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
448         {
449             double v = info.getBalance(i);
450             if (minValue > v) minValue = v;
451             if (maxValue < v) maxValue = v;
452         }
453         if (maxValue == minValue) return;
```

```
454
455     int barWidth = getWidth() / (info.getDeathAge() - info.getCurrentAge() + 1);
456     double scale = getHeight() / (maxValue - minValue);
457
458     for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
459     {
460         int x1 = (i - info.getCurrentAge()) * barWidth + 1;
461         int y1;
462         double v = info.getBalance(i);
463         int height;
464         int yOrigin = (int) (maxValue * scale);
465
466         if (v >= 0)
467         {
468             y1 = (int) ((maxValue - v) * scale);
469             height = yOrigin - y1;
470         }
471         else
472         {
473             y1 = yOrigin;
474             height = (int) (-v * scale);
475         }
476
477         if (i < info.getRetireAge()) g2.setPaint(colorPre);
478         else if (v >= 0) g2.setPaint(colorGain);
479         else g2.setPaint(colorLoss);
480         Rectangle2D bar = new Rectangle2D.Double(x1, y1, barWidth - 2, height);
481         g2.fill(bar);
482         g2.setPaint(Color.black);
483         g2.draw(bar);
484     }
485 }
486
487 /**
488 * Sets the color to be used before retirement.
489 * @param color the desired color
490 */
491 public void setColorPre(Color color)
492 {
493     colorPre = color;
494     repaint();
495 }
496
497 /**
498 * Sets the color to be used after retirement while the account balance is positive.
499 * @param color the desired color
500 */
```

(Continues)

Listing 7.5 (Continued)

```
501 public void setColorGain(Color color)
502 {
503     colorGain = color;
504     repaint();
505 }
506 /**
507 * Sets the color to be used after retirement when the account balance is negative.
508 * @param color the desired color
509 */
510 public void setColorLoss(Color color)
511 {
512     colorLoss = color;
513     repaint();
514 }
515
516 public Dimension getPreferredSize() { return PREFERRED_SIZE; }
517 }
518 }
```

Listing 7.6 retire/RetireResources.java

```
1 package retire;
2
3 import java.awt.*;
4
5 /**
6 * These are the English non-string resources for the retirement calculator.
7 * @version 1.21 2001-08-27
8 * @author Cay Horstmann
9 */
10 public class RetireResources extends java.util.ListResourceBundle
11 {
12     private static final Object[][] contents = {
13         // BEGIN LOCALIZE
14         { "colorPre", Color.blue }, { "colorGain", Color.white }, { "colorLoss", Color.red }
15         // END LOCALIZE
16     };
17
18     public Object[][] getContents()
19     {
20         return contents;
21     }
22 }
```

Listing 7.7 retire/RetireResources_de.java

```
1 package retire;
2
3 import java.awt.*;
4
5 /**
6  * These are the German non-string resources for the retirement calculator.
7  * @version 1.21 2001-08-27
8  * @author Cay Horstmann
9  */
10 public class RetireResources_de extends java.util.ListResourceBundle
11 {
12     private static final Object[][] contents = {
13         // BEGIN LOCALIZE
14         { "colorPre", Color.yellow }, { "colorGain", Color.black }, { "colorLoss", Color.red }
15         // END LOCALIZE
16     };
17
18     public Object[][] getContents()
19     {
20         return contents;
21     }
22 }
```

Listing 7.8 retire/RetireResources_zh.java

```
1 package retire;
2
3 import java.awt.*;
4
5 /**
6  * These are the Chinese non-string resources for the retirement calculator.
7  * @version 1.21 2001-08-27
8  * @author Cay Horstmann
9  */
10 public class RetireResources_zh extends java.util.ListResourceBundle
11 {
12     private static final Object[][] contents = {
13         // BEGIN LOCALIZE
14         { "colorPre", Color.red }, { "colorGain", Color.blue }, { "colorLoss", Color.yellow }
15         // END LOCALIZE
16     };
17
18     public Object[][] getContents()
19     {
20         return contents;
21     }
22 }
```

Listing 7.9 retire/RetireStrings.properties

```

1 language=Language
2 computeButton=Compute
3 savings=Prior Savings
4 contrib=Annual Contribution
5 income=Retirement Income
6 currentAge=Current Age
7 retireAge=Retirement Age
8 deathAge=Life Expectancy
9 inflationPercent=Inflation
10 investPercent=Investment Return
11 retire=Age: {0,number} Balance: {1,number,currency}

```

Listing 7.10 retire/RetireStrings_de.properties

```

1 language=Sprache
2 computeButton=Rechnen
3 savings=Vorherige Ersparnisse
4 contrib=J\u00e4hrliche Einzahlung
5 income=Einkommen nach Ruhestand
6 currentAge=Jetziges Alter
7 retireAge=Ruhestandsalter
8 deathAge=Lebenserwartung
9 inflationPercent=Inflation
10 investPercent=Investitionsgewinn
11 retire=Alter: {0,number} Guthaben: {1,number,currency}

```

Listing 7.11 retire/RetireStrings_zh.properties

```

1 language=\u8bed\u8a00
2 computeButton=\u8ba1\u7b97
3 savings=\u65e2\u5b58
4 contrib=\u6bcf\u5e74\u5b58\u91d1
5 income=\u9000\u4f11\u6536\u5165
6 currentAge=\u73b0\u9f84
7 retireAge=\u9000\u4f11\u5e74\u9f84
8 deathAge=\u9884\u671f\u5bff\u547d
9 inflationPercent=\u901a\u8d27\u81a8\u6da8
10 investPercent=\u6295\u8d44\u62a5\u916c
11 retire=\u5e74\u9f84: {0,number} \u603b\u7ed3: {1,number,currency}

```

You have seen how to use the internationalization features of the Java language. You can now use resource bundles to provide translations into multiple languages, and use formatters and collators for locale-specific text processing.

In the next chapter, we will delve into scripting, compiling, and annotation processing.

8

CHAPTER

Scripting, Compiling, and Annotation Processing

In this chapter

- 8.1 Scripting for the Java Platform, page 430
- 8.2 The Compiler API, page 443
- 8.3 Using Annotations, page 455
- 8.4 Annotation Syntax, page 462
- 8.5 Standard Annotations, page 470
- 8.6 Source-Level Annotation Processing, page 475
- 8.7 Bytecode Engineering, page 481

This chapter introduces three techniques for processing code. The scripting API lets you invoke code in a scripting language such as JavaScript or Groovy. You can use the compiler API when you want to compile Java code inside your application. Annotation processors operate on Java source or class files that contain annotations. As you will see, there are many applications for annotation processing, ranging from simple diagnostics to “bytecode engineering”—the insertion of bytecodes into class files or even running programs.

8.1 Scripting for the Java Platform

A scripting language is a language that avoids the usual edit/compile/link/run cycle by interpreting the program text at runtime. Scripting languages have a number of advantages:

- Rapid turnaround, encouraging experimentation
- Changing the behavior of a running program
- Enabling customization by program users

On the other hand, most scripting languages lack features that are beneficial for programming complex applications, such as strong typing, encapsulation, and modularity.

It is therefore tempting to combine the advantages of scripting and traditional languages. The scripting API lets you do just that for the Java platform. It enables you to invoke scripts written in JavaScript, Groovy, Ruby, and even exotic languages such as Scheme and Haskell, from a Java program. For example, the Renjin project (www.renjin.org) provides a Java implementation of the R programming language, which is commonly used for statistical programming, together with an “engine” of the scripting API.

In the following sections, we’ll show you how to select an engine for a particular language, how to execute scripts, and how to make use of advanced features that some scripting engines offer.

8.1.1 Getting a Scripting Engine

A scripting engine is a library that can execute scripts in a particular language. When the virtual machine starts, it discovers the available scripting engines. To enumerate them, construct a `ScriptEngineManager` and invoke the `getEngineFactories` method. You can ask each engine factory for the supported engine names, MIME types, and file extensions. Table 8.1 shows typical values.

Usually, you know which engine you need, and you can simply request it by name, MIME type, or extension. For example:

```
ScriptEngine engine = manager.getEngineByName("nashorn");
```

Java SE 8 includes a version of Nashorn, a JavaScript interpreter developed by Oracle. You can add more languages by providing the necessary JAR files on the class path.

Table 8.1 Properties of Scripting Engine Factories

Engine	Names	MIME types	Extensions
Nashorn (included with Java SE)	nashorn, Nashorn, js, JS, JavaScript, javascript, ECMAScript, ecmascript	application/javascript, application/ecmascript, text/javascript, text/ecmascript	.js
Groovy	groovy	None	.groovy
Renjin	Renjin	text/x-R	.R, .r, .S, .s
SISC Scheme	sisc	None	.scheme, .sisc

javax.script.ScriptEngineManager 6

- `List<ScriptEngineFactory> getEngineFactories()`
gets a list of all discovered engine factories.
- `ScriptEngine getEngineByName(String name)`
- `ScriptEngine getEngineByExtension(String extension)`
- `ScriptEngine getEngineByMimeType(String mimeType)`
gets the script engine with the given name, script file extension, or MIME type.

javax.script.ScriptEngineFactory 6

- `List<String> getNames()`
- `List<String> getExtensions()`
- `List<String> getMimeTypes()`
gets the names, script file extensions, and MIME types under which this factory is known.

8.1.2 Script Evaluation and Bindings

Once you have an engine, you can call a script simply by invoking

```
Object result = engine.eval(scriptString);
```

If the script is stored in a file, open a Reader and call

```
Object result = engine.eval(reader);
```

You can invoke multiple scripts on the same engine. If one script defines variables, functions, or classes, most scripting engines retain the definitions for later use. For example,

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```

will return 1729.

NOTE: To find out whether it is safe to concurrently execute scripts in multiple threads, call

```
Object param = factory.getParameter("THREADING");
```

The returned value is one of the following:

- `null`: Concurrent execution is not safe.
 - `"MULTITHREADED"`: Concurrent execution is safe. Effects from one thread might be visible from another thread.
 - `"THREAD-ISOLATED"`: In addition to `"MULTITHREADED"`, different variable bindings are maintained for each thread.
 - `"STATELESS"`: In addition to `"THREAD-ISOLATED"`, scripts do not alter variable bindings.
-

You will often want to add variable bindings to the engine. A binding consists of a name and an associated Java object. For example, consider these statements:

```
engine.put("k", 1728);
Object result = engine.eval("k + 1");
```

The script code reads the definition of `k` from the bindings in the “engine scope.” This is particularly important because most scripting languages can access Java objects, often with a syntax that is simpler than the Java syntax. For example,

```
engine.put("b", new JButton());
engine.eval("b.text = 'Ok'");
```

Conversely, you can retrieve variables that were bound by scripting statements:

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

In addition to the engine scope, there is also a global scope. Any bindings that you add to the `ScriptEngineManager` are visible to all engines.

Instead of adding bindings to the engine or global scope, you can collect them in an object of type `Bindings` and pass it to the `eval` method:

```
Bindings scope = engine.createBindings();
scope.put("b", new JButton());
engine.eval(scriptString, scope);
```

This is useful if a set of bindings should not persist for future calls to the `eval` method.

NOTE: You might want to have scopes other than the engine and global scopes.

For example, a web container might need request and session scopes. However, then you are on your own. You will need to write a class that implements the `ScriptContext` interface, managing a collection of scopes. Each scope is identified by an integer number, and scopes with lower numbers should be searched first. (The standard library provides a `SimpleScriptContext` class, but it only holds global and engine scopes.)

`javax.script.ScriptEngine` 6

- `Object eval(String script)`
- `Object eval(Reader reader)`
- `Object eval(String script, Bindings bindings)`
- `Object eval(Reader reader, Bindings bindings)`
evaluates the script given by the string or reader, subject to the given bindings.
- `Object get(String key)`
- `void put(String key, Object value)`
gets or puts a binding in the engine scope.
- `Bindings createBindings()`
creates an empty `Bindings` object suitable for this engine.

`javax.script.ScriptEngineManager` 6

- `Object get(String key)`
- `void put(String key, Object value)`
gets or puts a binding in the global scope.

***javax.script.Bindings* 6**

- `Object get(String key)`
- `void put(String key, Object value)`

gets or puts a binding into the scope represented by this `Bindings` object.

8.1.3 Redirecting Input and Output

You can redirect the standard input and output of a script by calling the `setReader` and `setWriter` methods of the script context. For example,

```
StringWriter writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

Any output written with the JavaScript `print` or `println` functions is sent to `writer`.

The `setReader` and `setWriter` methods only affect the scripting engine's standard input and output sources. For example, if you execute the JavaScript code

```
println("Hello");
java.lang.System.out.println("World");
```

only the first output is redirected.

The Nashorn engine does not have the notion of a standard input source. Calling `setReader` has no effect.

***javax.script.ScriptEngine* 6**

- `ScriptContext getContext()`
- gets the default script context for this engine.

***javax.script.ScriptContext* 6**

- `Reader getReader()`
- `void setReader(Reader reader)`
- `Writer getWriter()`
- `void setWriter(Writer writer)`
- `Writer getErrorWriter()`
- `void setErrorWriter(Writer writer)`

gets or sets the reader for input or writer for normal or error output.

8.1.4 Calling Scripting Functions and Methods

With many script engines, you can invoke a function in the scripting language without having to evaluate the actual script code. This is useful if you allow users to implement a service in a scripting language of their choice.

The script engines that offer this functionality implement the `Invocable` interface. In particular, the Nashorn engine implements `Invocable`.

To call a function, call the `invokeFunction` method with the function name, followed by the function parameters:

```
// Define greet function in JavaScript
engine.eval("function greet(how, whom) { return how + ', ' + whom + '!' }");

// Call the function with arguments "Hello", "World"
result = ((Invocable) engine).invokeFunction("greet", "Hello", "World");
```

If the scripting language is object-oriented, call `invokeMethod`:

```
// Define Greeter class in JavaScript
engine.eval("function Greeter(how) { this.how = how }");
engine.eval("Greeter.prototype.welcome =
    + " function(whom) { return this.how + ', ' + whom + '!' }");

// Construct an instance
Object yo = engine.eval("new Greeter('Yo')");

// Call the welcome method on the instance
result = ((Invocable) engine).invokeMethod(yo, "welcome", "World");
```

NOTE: For more information on how to define classes in JavaScript, see *JavaScript—The Good Parts* by Douglas Crockford (O'Reilly, 2008).

NOTE: If the script engine does not implement the `Invocable` interface, you might still be able to call a method in a language-independent way. The `getMethodCallSyntax` method of the `ScriptEngineFactory` interface produces a string that you can pass to the `eval` method. However, all method parameters must be bound to names, whereas `invokeMethod` can be called with arbitrary values.

You can go a step further and ask the scripting engine to implement a Java interface. Then you can call scripting functions and methods with the Java method call syntax.

The details depend on the scripting engine, but typically you need to supply a function for each method of the interface. For example, consider a Java interface

```
public interface Greeter
{
    String welcome(String whom);
}
```

If you define a global function with the same name in Nashorn, you can call it through this interface.

```
// Define welcome function in JavaScript
engine.eval("function welcome(whom) { return 'Hello, ' + whom + '!' }");

// Get a Java object and call a Java method
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
result = g.welcome("World");
```

In an object-oriented scripting language, you can access a script class through a matching Java interface. For example, here is how to call an object of the JavaScript SimpleGreeter class with Java syntax:

```
Greeter g = ((Invocable) engine).getInterface(yo, Greeter.class);
result = g.welcome("World");
```

In summary, the Invocable interface is useful if you want to call scripting code from Java without worrying about the scripting language syntax.

javas.ript.Invocable 6

- `Object invokeFunction(String name, Object... parameters)`
- `Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)`
invokes the function or method with the given name, passing the given parameters.
- `<T> T getInterface(Class<T> iface)`
returns an implementation of the given interface, implementing the methods with functions in the scripting engine.
- `<T> T getInterface(Object implicitParameter, Class<T> iface)`
returns an implementation of the given interface, implementing the methods with the methods of the given object.

8.1.5 Compiling a Script

Some scripting engines can compile scripting code into an intermediate form for efficient execution. Those engines implement the `Compilable` interface. The following example shows how to compile and evaluate code contained in a script file:

```
Reader reader = new FileReader("myscript.js");
CompiledScript script = null;
if (engine implements Compilable)
    script = ((Compilable) engine).compile(reader);
```

Once the script is compiled, you can execute it. The following code executes the compiled script if compilation was successful, or the original script if the engine didn't support compilation:

```
if (script != null)
    script.eval();
else
    engine.eval(reader);
```

Of course, it only makes sense to compile a script if you need to execute it repeatedly.

`javax.script.Compilable` 6

- `CompiledScript compile(String script)`
 - `CompiledScript compile(Reader reader)`
- compiles the script given by a string or reader.

`javax.script.CompiledScript` 6

- `Object eval()`
 - `Object eval(Bindings bindings)`
- evaluates this script.

8.1.6 An Example: Scripting GUI Events

To illustrate the scripting API, we will write a sample program that allows users to specify event handlers in a scripting language of their choice.

Have a look at the program in Listing 8.1 that adds scripting to an arbitrary frame class. By default it reads the `ButtonFrame` class in Listing 8.2, which is similar to the event handling demo in Volume I, with two differences:

- Each component has its `name` property set.
- There are no event handlers.

The event handlers are defined in a property file. Each property definition has the form

```
componentName.eventName = scriptCode
```

For example, if you choose to use JavaScript, supply the event handlers in a file `js.properties`, like this:

```
yellowButton.action=panel.background = java.awt.Color.YELLOW  
blueButton.action=panel.background = java.awt.Color.BLUE  
redButton.action=panel.background = java.awt.Color.RED
```

The companion code also has files for Groovy, R, and SISC Scheme.

The program starts by loading an engine for the language specified on the command line. If no language is specified, we use JavaScript.

We then process a script `init.language` if it is present. This is useful for the R and Scheme languages, which need some initializations that we did not want to include in every event handler script.

Next, we recursively traverse all child components and add the bindings (`name, object`) into a map of components. Then we add the bindings to the engine.

Next, we read the file `language.properties`. For each property, we synthesize an event handler proxy that causes the script code to be executed. The details are a bit technical. You might want to read the section on proxies in Volume I, Chapter 6, if you want to follow the implementation in detail. The essential part, however, is that each event handler calls

```
engine.eval(scriptCode);
```

Let us look at the `yellowButton` in more detail. When the line

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

is processed, we find the `JButton` component with the name "yellowButton". We then attach an `ActionListener` with an `actionPerformed` method that executes the script

```
panel.background = java.awt.Color.YELLOW
```

if the scripting is done with Nashorn.

The engine contains a binding that binds the name "panel" to the `JPanel` object. When the event occurs, the `setBackground` method of the panel is executed, and the color changes.

You can run this program with the JavaScript event handlers simply by executing

```
java ScriptTest
```

For the Groovy handlers, use

```
java -classpath .:groovy/lib/* ScriptTest groovy
```

Here, *groovy* is the directory into which you installed Groovy.

For the Renjin implementation of R, include the JAR files for Renjin Studio and the Renjin script engine on the classpath. Both are available at www.renjin.org/downloads.html.

To try out Scheme, download SISC Scheme from <http://sisc-scheme.org> and run

```
java -classpath .:sisc/*:jsr223-engines/scheme/build/scheme-engine.jar ScriptTest scheme
```

where *sisc* is the installation directory for SISC Scheme and *jsr223-engines* is the directory that contains the engine adapters from <http://java.net/projects/scripting>.

This application demonstrates how to use scripting for Java GUI programming. One could go a step further and describe the GUI with an XML file, as you have seen in Chapter 3. Then our program would become an interpreter for GUIs that have visual presentation defined by XML and behavior defined by a scripting language. Note the similarity to a dynamic HTML page or a dynamic server-side scripting environment.

Listing 8.1 *script/ScriptTest.java*

```
1 package script;
2
3 import java.awt.*;
4 import java.beans.*;
5 import java.io.*;
6 import java.lang.reflect.*;
7 import java.util.*;
8 import javax.script.*;
9 import javax.swing.*;
10
11 /**
12  * @version 1.02 2016-05-10
13  * @author Cay Horstmann
14  */
15 public class ScriptTest
16 {
17     public static void main(String[] args)
18     {
19         EventQueue.invokeLater(() ->
20             {
```

(Continues)

Listing 8.1 (*Continued*)

```
21     try
22     {
23         ScriptEngineManager manager = new ScriptEngineManager();
24         String language;
25         if (args.length == 0)
26         {
27             System.out.println("Available factories: ");
28             for (ScriptEngineFactory factory : manager.getEngineFactories())
29                 System.out.println(factory.getEngineName());
30
31             language = "nashorn";
32         }
33         else language = args[0];
34
35         final ScriptEngine engine = manager.getEngineByName(language);
36         if (engine == null)
37         {
38             System.err.println("No engine for " + language);
39             System.exit(1);
40         }
41
42         final String frameClassName = args.length < 2 ? "buttons1.ButtonFrame" : args[1];
43         JFrame frame = (JFrame) Class.forName(frameClassName).newInstance();
44         InputStream in = frame.getClass().getResourceAsStream("init." + language);
45         if (in != null) engine.eval(new InputStreamReader(in));
46         Map<String, Component> components = new HashMap<>();
47         getComponentBindings(frame, components);
48         components.forEach((name, c) -> engine.put(name, c));
49
50         final Properties events = new Properties();
51         in = frame.getClass().getResourceAsStream(language + ".properties");
52         events.load(in);
53
54         for (final Object e : events.keySet())
55         {
56             String[] s = ((String) e).split("\\.");
57             addListener(s[0], s[1], (String) events.get(e), engine, components);
58         }
59         frame.setTitle("ScriptTest");
60         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
61         frame.setVisible(true);
62     }
63     catch (ReflectiveOperationException | IOException
64          | ScriptException | IntrospectionException ex)
65     {
66         ex.printStackTrace();
67     }
68 );
```

```
69 }
70
71 /**
72 * Gathers all named components in a container.
73 * @param c the component
74 * @param namedComponents a map into which to enter the component names and components
75 */
76 private static void getComponentBindings(Component c, Map<String, Component> namedComponents)
77 {
78     String name = c.getName();
79     if (name != null) { namedComponents.put(name, c); }
80     if (c instanceof Container)
81     {
82         for (Component child : ((Container) c).getComponents())
83             getComponentBindings(child, namedComponents);
84     }
85 }
86
87 /**
88 * Adds a listener to an object whose listener method executes a script.
89 * @param beanName the name of the bean to which the listener should be added
90 * @param eventName the name of the listener type, such as "action" or "change"
91 * @param scriptCode the script code to be executed
92 * @param engine the engine that executes the code
93 * @param bindings the bindings for the execution
94 * @throws IntrospectionException
95 */
96 private static void addListener(String beanName, String eventName, final String scriptCode,
97     final ScriptEngine engine, Map<String, Component> components)
98     throws ReflectiveOperationException, IntrospectionException
99 {
100     Object bean = components.get(beanName);
101     EventSetDescriptor descriptor = getEventSetDescriptor(bean, eventName);
102     if (descriptor == null) return;
103     descriptor.getAddListenerMethod().invoke(bean,
104         Proxy.newProxyInstance(null, new Class[] { descriptor.getListenerType() },
105             (proxy, method, args) ->
106             {
107                 engine.eval(scriptCode);
108                 return null;
109             }
110         ));
111 }
112
113 private static EventSetDescriptor getEventSetDescriptor(Object bean, String eventName)
114     throws IntrospectionException
115 {
116     for (EventSetDescriptor descriptor : Introspector.getBeanInfo(bean.getClass())
117         .getEventSetDescriptors())
118     {
119         if (descriptor.getListenerType().isAssignableFrom(method.getDeclaringClass()))
120             return descriptor;
121     }
122 }
123
124 /**
125 * Returns the value of the specified property for the specified component.
126 * @param component the component
127 * @param name the name of the property
128 * @return the value of the property
129 */
130 private static Object getProperty(Component component, String name)
131 {
132     if (component instanceof PropertyHolder)
133     {
134         PropertyHolder ph = (PropertyHolder) component;
135         if (ph.getPropertyNames().contains(name))
136             return ph.getProperty(name);
137     }
138     return null;
139 }
```

(Continues)

Listing 8.1 (*Continued*)

```
118     if (descriptor.getName().equals(eventName)) return descriptor;
119     return null;
120   }
121 }
```

Listing 8.2 buttons1/ButtonFrame.java

```
1 package buttons1;
2
3 import javax.swing.*;
4
5 /**
6  * A frame with a button panel.
7  * @version 1.00 2007-11-02
8  * @author Cay Horstmann
9 */
10 public class ButtonFrame extends JFrame
11 {
12     private static final int DEFAULT_WIDTH = 300;
13     private static final int DEFAULT_HEIGHT = 200;
14
15     private JPanel panel;
16     private JButton yellowButton;
17     private JButton blueButton;
18     private JButton redButton;
19
20     public ButtonFrame()
21     {
22         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24         panel = new JPanel();
25         panel.setName("panel");
26         add(panel);
27
28         yellowButton = new JButton("Yellow");
29         yellowButton.setName("yellowButton");
30         blueButton = new JButton("Blue");
31         blueButton.setName("blueButton");
32         redButton = new JButton("Red");
33         redButton.setName("redButton");
34
35         panel.add(yellowButton);
36         panel.add(blueButton);
37         panel.add(redButton);
38     }
39 }
```

8.2 The Compiler API

In the preceding sections, you saw how to interact with code in a scripting language. Now we turn to a different scenario: Java programs that compile Java code. There are quite a few tools that need to invoke the Java compiler, such as:

- Development environments
- Java teaching and tutoring programs
- Build and test automation tools
- Templating tools that process snippets of Java code, such as JavaServer Pages (JSP)

In the past, applications invoked the Java compiler by calling undocumented classes in the *jdk/lib/tools.jar* library. Nowadays, a public API for compilation is a part of the Java platform, and it is no longer necessary to use *tools.jar*. This section explains the compiler API.

8.2.1 Compiling the Easy Way

It is very easy to invoke the compiler. Here is a sample call:

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
OutputStream outStream = . . .;
OutputStream errStream = . . .;
int result = compiler.run(null, outStream, errStream, "-sourcepath", "src", "Test.java");
```

A result value of 0 indicates successful compilation.

The compiler sends output and error messages to the provided streams. You can set these parameters to `null`, in which case `System.out` and `System.err` are used. The first parameter of the `run` method is an input stream. As the compiler takes no console input, you can always leave it as `null`. (The `run` method is inherited from a generic `Tool` interface, which allows for tools that read input.)

The remaining parameters of the `run` method are simply the arguments that you would pass to `javac` if you invoked it on the command line. These can be options or file names.

8.2.2 Using Compilation Tasks

You can have even more control over the compilation process with a `CompilationTask` object. In particular, you can

- Control the source of program code—for example, by providing code in a string builder instead of a file

- Control the placement of class files—for example, by storing them in a database
- Listen to error and warning messages as they occur during compilation
- Run the compiler in the background

The location of source and class files is controlled by a `JavaFileManager`. It is responsible for determining `JavaFileObject` instances for source and class files. A `JavaFileObject` can correspond to a disk file, or it can provide another mechanism for reading and writing its contents.

To listen to error messages, install a `DiagnosticListener`. The listener receives a `Diagnostic` object whenever the compiler reports a warning or error message. The `DiagnosticCollector` class implements this interface. It simply collects all diagnostics so that you can iterate through them after the compilation is complete.

A `Diagnostic` object contains information about the problem location (including file name, line number, and column number) as well as a human-readable description.

To obtain a `CompilationTask` object, call the `getTask` method of the `JavaCompiler` class. You need to specify:

- A `Writer` for any compiler output that is not reported as a `Diagnostic`, or `null` to use `System.err`
- A `JavaFileManager`, or `null` to use the compiler's standard file manager
- A `DiagnosticListener`
- Option strings, or `null` for no options
- Class names for annotation processing, or `null` if none are specified (we'll discuss annotation processing later in this chapter)
- `JavaFileObject` instances for source files

You need to provide the last three arguments as `Iterable` objects. For example, a sequence of options might be specified as

```
Iterable<String> options = Arrays.asList("-g", "-d", "classes");
```

Alternatively, you can use any collection class.

If you want the compiler to read source files from disk, you can ask the `StandardJavaFileManager` to translate the file name strings or `File` objects to `JavaFileObject` instances. For example,

```
StandardJavaFileManager fileManager = compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> fileObjects = fileManager.getJavaFileObjectsFromStrings(fileName);
```

However, if you want the compiler to read source code from somewhere other than a disk file, you need to supply your own `JavaFileObject` subclass. Listing 8.3

shows the code for a source file object with data contained in a `StringBuilder`. The class extends the `SimpleJavaFileObject` convenience class and overrides the `getCharContent` method to return the content of the string builder. We'll use this class in our example program in which we dynamically produce the code for a Java class and then compile it.

The `CompilationTask` interface extends the `Callable<Boolean>` interface. You can pass it to an `Executor` for execution in another thread, or you can simply invoke the `call` method. A return value of `Boolean.FALSE` indicates failure.

```
Callable<Boolean> task = new JavaCompiler.CompilationTask(null, fileManager, diagnostics,
    options, null, fileObjects);
if (!task.call())
    System.out.println("Compilation failed");
```

If you simply want the compiler to produce class files on disk, you need not customize the `JavaFileManager`. However, our sample application will generate class files in byte arrays and later read them from memory, using a special class loader. Listing 8.4 defines a class that implements the `JavaFileObject` interface. Its `openOutputStream` method returns the `ByteArrayOutputStream` into which the compiler will deposit the bytecodes.

It turns out a bit tricky to tell the compiler's file manager to use these file objects. The library doesn't supply a class that implements the `StandardJavaFileManager` interface. Instead, you subclass the `ForwardingJavaFileManager` class that delegates all calls to a given file manager. In our situation, we only want to change the `getJavaFileForOutput` method. We achieve this with the following outline:

```
JavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);
fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
{
    public JavaFileObject getJavaFileForOutput(Location location, final String className,
        Kind kind, FileObject sibling) throws IOException
    {
        return custom file object
    }
};
```

In summary, call the `run` method of the `JavaCompiler` task if you simply want to invoke the compiler in the usual way, reading and writing disk files. You can capture the output and error messages, but you need to parse them yourself.

If you want more control over file handling or error reporting, use the `CompilationTask` interface instead. Its API is quite complex, but you can control every aspect of the compilation process.

Listing 8.3 compiler/StringBuilderJavaSource.java

```
1 package compiler;
2
3 import java.net.*;
4 import javax.tools.*;
5
6 /**
7 * A Java source that holds the code in a string builder.
8 * @version 1.00 2007-11-02
9 * @author Cay Horstmann
10 */
11 public class StringBuilderJavaSource extends SimpleJavaFileObject
12 {
13     private StringBuilder code;
14
15     /**
16      * Constructs a new StringBuilderJavaSource.
17      * @param name the name of the source file represented by this file object
18      */
19     public StringBuilderJavaSource(String name)
20     {
21         super(URI.create("string:/// " + name.replace('.', '/') + Kind.SOURCE.extension),
22               Kind.SOURCE);
23         code = new StringBuilder();
24     }
25
26     public CharSequence getCharContent(boolean ignoreEncodingErrors)
27     {
28         return code;
29     }
30
31     public void append(String str)
32     {
33         code.append(str);
34         code.append('\n');
35     }
36 }
```

Listing 8.4 compiler/ByteArrayJavaClass.java

```
1 package compiler;
2
3 import java.io.*;
4 import java.net.*;
5 import javax.tools.*;
6
```

```
7 /**
8  * A Java class that holds the bytecodes in a byte array.
9  * @version 1.00 2007-11-02
10 * @author Cay Horstmann
11 */
12 public class ByteArrayJavaClass extends SimpleJavaFileObject
13 {
14     private ByteArrayOutputStream stream;
15
16     /**
17      * Constructs a new ByteArrayJavaClass.
18      * @param name the name of the class file represented by this file object
19      */
20     public ByteArrayJavaClass(String name)
21     {
22         super(URI.create("bytes:/// " + name), Kind.CLASS);
23         stream = new ByteArrayOutputStream();
24     }
25
26     public OutputStream openOutputStream() throws IOException
27     {
28         return stream;
29     }
30
31     public byte[] getBytes()
32     {
33         return stream.toByteArray();
34     }
35 }
```

***javax.tools.Tool* 6**

- `int run(InputStream in, OutputStream out, OutputStream err, String... arguments)`
runs the tool with the given input, output, and error streams and the given arguments. Returns 0 for success, a nonzero value for failure.

***javax.tools.JavaCompiler* 6**

- `StandardJavaFileManager getStandardFileManager(DiagnosticListener<? super JavaFileObject> diagnosticListener, Locale locale, Charset charset)`
gets the standard file manager for this compiler. You can supply `null` for default error reporting, locale, and character set.

(Continues)

javax.tools.JavaCompiler 6 (Continued)

- `JavaCompiler.CompilationTask getTask(Writerout, JavaFileManager fileManager, DiagnosticListener<? super JavaFileObject> diagnosticListener, Iterable<String> options, Iterable<String> classesForAnnotationProcessing, Iterable<? extends JavaFileObject> sourceFiles)`
gets a compilation task that, when called, will compile the given source files. See the discussion in the preceding section for details.

javax.tools.StandardJavaFileManager 6

- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings(Iterable<String> fileNames)`
- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles(Iterable<? extends File> files)`
translates a sequence of file names or files into a sequence of `JavaFileObject` instances.

javax.tools.JavaCompiler.CompilationTask 6

- `Boolean call()`
performs the compilation task.

javax.tools.DiagnosticCollector<S> 6

- `DiagnosticCollector<S> DiagnosticCollector()`
constructs an empty collector.
- `List<Diagnostic<? extends S>> getDiagnostics()`
gets the collected diagnostics.

javax.tools.Diagnostic<S> 6

- `S getSource()`
gets the source object associated with this diagnostic.
- `Diagnostic.Kind getKind()`
gets the type of this diagnostic—one of `ERROR`, `WARNING`, `MANDATORY_WARNING`, `NOTE`, or `OTHER`.

(Continues)

javax.tools.Diagnostic<S> 6 (Continued)

- `String getMessage(Locale locale)`
gets the message describing the issue raised in this diagnostic. Pass `null` for the default locale.
- `long getLineNumber()`
- `long getColumnNumber()`
gets the position of the issue raised in this diagnostic.

javax.tools.SimpleJavaFileObject 6

- `CharSequence getCharContent(boolean ignoreEncodingErrors)`
Override this method for a file object that represents a source file and produces the source code.
- `OutputStream openOutputStream()`
Override this method for a file object that represents a class file and produces a stream to which the bytecodes can be written.

javax.tools.ForwardingJavaFileManager<M extends JavaFileManager> 6

- `protected ForwardingJavaFileManager(M fileManager)`
constructs a `JavaFileManager` that delegates all calls to the given file manager.
- `FileObject getFileForOutput(JavaFileManager.Location location, String className, JavaFileObject.Kind kind, FileObject sibling)`
intercept this call if you want to substitute a file object for writing class files; `kind` is one of `SOURCE`, `CLASS`, `HTML`, or `OTHER`.

8.2.3 An Example: Dynamic Java Code Generation

In the JSP technology for dynamic web pages, you can mix HTML with snippets of Java code, for example

```
<p>The current date and time is <b><%= new java.util.Date() %></b>.</p>
```

The JSP engine dynamically compiles the Java code into a servlet. In our sample application, we use a simpler example and generate dynamic Swing code instead. The idea is that you use a GUI builder to lay out the components in a frame and specify the behavior of the components in an external file. Listing 8.5 shows a

very simple example of a frame class, and Listing 8.6 shows the code for the button actions. Note that the constructor of the frame class calls an abstract method `addEventHandlers`. Our code generator will produce a subclass that implements the `addEventHandlers` method, adding an action listener for each line in the `action.properties` file. (We leave it as the proverbial exercise to the reader to extend the code generation to other event types.)

We place the subclass into a package with the name `x`, which we hope is not used anywhere else in the program. The generated code has the form

```
package x;
public class Frame extends SuperclassName
{
    protected void addEventHandlers()
    {
        componentName1.addActionListener(new java.awt.event.ActionListener()
        {
            public void actionPerformed(java.awt.event.ActionEvent) { code for event handler1 }
        });
        // repeat for the other event handlers ...
    }
}
```

The `buildSource` method in the program of Listing 8.7 builds up this code and places it into a `StringBuilderJavaSource` object. That object is passed to the Java compiler.

We use a `ForwardingJavaFileManager` with a `getJavaFileForOutput` method that constructs a `ByteArrayJavaClass` object for every class in the `x` package. These objects capture the class files generated when the `x.Frame` class is compiled. The method adds each file object to a list before returning it so that we can locate the bytecodes later. Note that compiling the `x.Frame` class produces a class file for the main class and one class file per listener class.

After compilation, we build a map that associates class names with bytecode arrays. A simple class loader (shown in Listing 8.8) loads the classes stored in this map.

We ask the class loader to load the class that we just compiled. Then, we construct and display the application's frame class.

```
ClassLoader loader = new MapClassLoader(byteCodeMap);
Class<?> cl = loader.loadClass("x.Frame");
Frame frame = (JFrame) cl.newInstance();
frame.setVisible(true);
```

When you click the buttons, the background color changes in the usual way. To see that the actions are dynamically compiled, change one of the lines in `action.properties`, for example, like this:

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW); yellowButton.setEnabled(false);
```

Run the program again. Now the Yellow button is disabled after you click it. Also, have a look at the code directories. You will not find any source or class files for the classes in the `x` package. This example demonstrates how you can use dynamic compilation with in-memory source and class files.

Listing 8.5 buttons2/ButtonFrame.java

```
1 package buttons2;
2 import javax.swing.*;
3
4 /**
5  * A frame with a button panel.
6  * @version 1.00 2007-11-02
7  * @author Cay Horstmann
8 */
9 public abstract class ButtonFrame extends JFrame
10 {
11     public static final int DEFAULT_WIDTH = 300;
12     public static final int DEFAULT_HEIGHT = 200;
13
14     protected JPanel panel;
15     protected JButton yellowButton;
16     protected JButton blueButton;
17     protected JButton redButton;
18
19     protected abstract void addEventHandlers();
20
21     public ButtonFrame()
22     {
23         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
24
25         panel = new JPanel();
26         add(panel);
27
28         yellowButton = new JButton("Yellow");
29         blueButton = new JButton("Blue");
30         redButton = new JButton("Red");
31
32         panel.add(yellowButton);
33         panel.add(blueButton);
34         panel.add(redButton);
35
36         addEventHandlers();
37     }
38 }
```

Listing 8.6 buttons2/action.properties

```
1 yellowButton=panel.setBackground(java.awt.Color.YELLOW);
2 blueButton=panel.setBackground(java.awt.Color.BLUE);
```

Listing 8.7 compiler/CompilerTest.java

```
1 package compiler;
2
3 import java.awt.*;
4 import java.io.*;
5 import java.util.*;
6 import java.util.List;
7 import javax.swing.*;
8 import javax.tools.*;
9 import javax.tools.JavaFileObject.*;
10
11 /**
12 * @version 1.01 2016-05-10
13 * @author Cay Horstmann
14 */
15 public class CompilerTest
16 {
17     public static void main(final String[] args) throws IOException, ClassNotFoundException
18     {
19         JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
20
21         final List<ByteArrayJavaClass> classFileObjects = new ArrayList<>();
22
23         DiagnosticCollector<JavaFileObject> diagnostics = new DiagnosticCollector<>();
24
25         JavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);
26         fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
27         {
28             public JavaFileObject getJavaFileForOutput(Location location, final String className,
29                 Kind kind, FileObject sibling) throws IOException
30             {
31                 if (className.startsWith("x."))
32                 {
33                     ByteArrayJavaClass fileObject = new ByteArrayJavaClass(className);
34                     classFileObjects.add(fileObject);
35                     return fileObject;
36                 }
37                 else return super.getJavaFileForOutput(location, className, kind, sibling);
38             }
39         };
40     }
```

```
41 String frameClassName = args.length == 0 ? "buttons2.ButtonFrame" : args[0];
42 JavaFileObject source = buildSource(frameClassName);
43 JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager, diagnostics, null,
44     null, Arrays.asList(source));
45 Boolean result = task.call();
46
47 for (Diagnostic<? extends JavaFileObject> d : diagnostics.getDiagnostics())
48     System.out.println(d.getKind() + ": " + d.getMessage(null));
49 fileManager.close();
50 if (!result)
51 {
52     System.out.println("Compilation failed.");
53     System.exit(1);
54 }
55
56 EventQueue.invokeLater(() ->
57 {
58     try
59     {
60         Map<String, byte[]> byteCodeMap = new HashMap<>();
61         for (ByteArrayJavaClass cl : classFileObjects)
62             byteCodeMap.put(cl.getName().substring(1), cl.getBytes());
63         ClassLoader loader = new MapClassLoader(byteCodeMap);
64         JFrame frame = (JFrame) loader.loadClass("x.Frame").newInstance();
65         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
66         frame.setTitle("CompilerTest");
67         frame.setVisible(true);
68     }
69     catch (Exception ex)
70     {
71         ex.printStackTrace();
72     }
73 });
74 }
75
76 /**
77 * Builds the source for the subclass that implements the addEventHandlers method.
78 * @return a file object containing the source in a string builder
79 */
80 static JavaFileObject buildSource(String superClass)
81     throws IOException, ClassNotFoundException
82 {
83     StringBuilderJavaSource source = new StringBuilderJavaSource("x.Frame");
84     source.append("package x;\n");
85     source.append("public class Frame extends " + superClass + " {\n");
86     source.append("protected void addEventHandlers() {\n");
87     final Properties props = new Properties();
```

(Continues)

Listing 8.7 (*Continued*)

```
88     props.load(Class.forName(superclassName).getResourceAsStream("action.properties"));
89     for (Map.Entry<Object, Object> e : props.entrySet())
90     {
91         String beanName = (String) e.getKey();
92         String eventCode = (String) e.getValue();
93         source.append(beanName + ".addActionListener(event -> {");
94         source.append(eventCode);
95         source.append("} );");
96     }
97     source.append("} }");
98     return source;
99 }
100 }
```

Listing 8.8 compiler/MapClassLoader.java

```
1 package compiler;
2
3 import java.util.*;
4
5 /**
6 * A class loader that loads classes from a map whose keys are class names and whose values are
7 * byte code arrays.
8 * @version 1.00 2007-11-02
9 * @author Cay Horstmann
10 */
11 public class MapClassLoader extends ClassLoader
12 {
13     private Map<String, byte[]> classes;
14
15     public MapClassLoader(Map<String, byte[]> classes)
16     {
17         this.classes = classes;
18     }
19
20     protected Class<?> findClass(String name) throws ClassNotFoundException
21     {
22         byte[] classBytes = classes.get(name);
23         if (classBytes == null) throw new ClassNotFoundException(name);
24         Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
25         if (cl == null) throw new ClassNotFoundException(name);
26         return cl;
27     }
28 }
```

8.3 Using Annotations

Annotations are tags that you insert into your source code so that some tool can process them. The tools can operate on the source level, or they can process class files into which the compiler has placed annotations.

Annotations do not change the way in which your programs are compiled. The Java compiler generates the same virtual machine instructions with or without the annotations.

To benefit from annotations, you need to select a *processing tool*. Use annotations that your processing tool understands, then apply the processing tool to your code.

There is a wide range of uses for annotations, and that generality can be confusing at first. Here are some uses for annotations:

- Automatic generation of auxiliary files, such as deployment descriptors or bean information classes
- Automatic generation of code for testing, logging, transaction semantics, and so on

8.3.1 An Introduction into Annotations

We'll start our discussion of annotations with the basic concepts and put them to use in a concrete example: We will mark methods as event listeners for AWT components, and show you an annotation processor that analyzes the annotations and hooks up the listeners. We'll then discuss the syntax rules in detail and finish the chapter with two advanced examples of annotation processing. One of them processes source-level annotations, the other uses the Apache Bytecode Engineering Library to process class files, injecting additional bytecodes into annotated methods.

Here is an example of a simple annotation:

```
public class MyClass
{
    ...
    @Test public void checkRandomInsertions()
}
```

The annotation `@Test` annotates the `checkRandomInsertions` method.

In Java, an annotation is used like a *modifier* and is placed before the annotated item *without a semicolon*. (A modifier is a keyword such as `public` or `static`.) The name of each annotation is preceded by an `@` symbol, similar to Javadoc comments.

However, Javadoc comments occur inside `/** . . . */` delimiters, whereas annotations are part of the code.

By itself, the `@Test` annotation does not do anything. It needs a tool to be useful. For example, the JUnit 4 testing tool (available at <http://junit.org>) calls all methods that are labeled `@Test` when testing a class. Another tool might remove all test methods from a class file so they are not shipped with the program after it has been tested.

Annotations can be defined to have *elements*, such as

```
@Test(timeout="10000")
```

These elements can be processed by the tools that read the annotations. Other forms of elements are possible; we'll discuss them later in this chapter.

Besides methods, you can annotate classes, fields, and local variables—an annotation can be anywhere you could put a modifier such as `public` or `static`. In addition, as you will see in Section 8.4, “Annotation Syntax,” on p. 462, you can annotate packages, parameter variables, type parameters, and type uses.

Each annotation must be defined by an *annotation interface*. The methods of the interface correspond to the elements of the annotation. For example, the JUnit `Test` annotation is defined by the following interface:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test
{
    long timeout() default 0L;
    ...
}
```

The `@interface` declaration creates an actual Java interface. Tools that process annotations receive objects that implement the annotation interface. A tool would call the `timeout` method to retrieve the `timeout` element of a particular `Test` annotation.

The `Target` and `Retention` annotations are *meta-annotations*. They annotate the `Test` annotation, marking it as an annotation that can be applied to methods only and is retained when the class file is loaded into the virtual machine. We'll discuss these in detail in Section 8.5.3, “Meta-Annotations,” on p. 472.

You have now seen the basic concepts of program metadata and annotations. In the next section, we'll walk through a concrete example of annotation processing.

8.3.2 An Example: Annotating Event Handlers

One of the more boring tasks in user interface programming is the wiring of listeners to event sources. Many listeners are of the form

```
myButton.addActionListener(() -> doSomething());
```

In this section, we'll design an annotation to reverse the wiring. The annotation, defined in Listing 8.11, is used as follows:

```
@ActionListenerFor(source="myButton") void doSomething() { . . . }
```

The programmer no longer has to make calls to `addActionListener`. Instead, each method is tagged with an annotation. Listing 8.10 shows the `ButtonFrame` class from Volume I, Chapter 11, reimplemented with these annotations.

We also need to define an annotation interface. The code is in Listing 8.11.

Of course, the annotations don't do anything by themselves. They sit in the source file. The compiler places them in the class file, and the virtual machine loads them. We now need a mechanism to analyze them and install action listeners. That is the job of the `ActionListenerInstaller` class. The `ButtonFrame` constructor calls

```
ActionListenerInstaller.processAnnotations(this);
```

The static `processAnnotations` method enumerates all methods of the object it received. For each method, it gets the `ActionListenerFor` annotation object and processes it.

```
Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
    ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
    if (a != null) . . .
}
```

Here, we use the `getAnnotation` method defined in the `AnnotatedElement` interface. The classes `Method`, `Constructor`, `Field`, `Class`, and `Package` implement this interface.

The name of the source field is stored in the annotation object. We retrieve it by calling the `source` method, and then look up the matching field.

```
String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);
```

This shows a limitation of our annotation. The source element must be the name of a field. It cannot be a local variable.

The remainder of the code is rather technical. For each annotated method, we construct a proxy object, implementing the `ActionListener` interface, with an `actionPerformed` method that calls the annotated method. (For more information about proxies, see Volume I, Chapter 6.) The details are not important. The key observation is that the functionality of the annotations was established by the `processAnnotations` method.

Figure 8.1 shows how annotations are handled in this example.

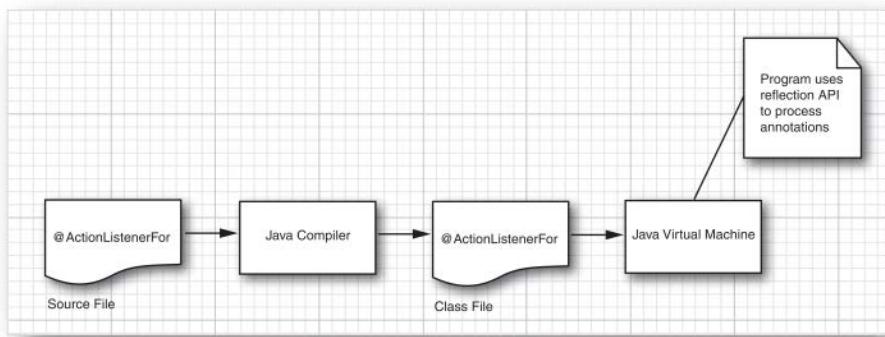


Figure 8.1 Processing annotations at runtime

In this example, the annotations were processed at runtime. It is also possible to process them at the source level: A source code generator would then produce the code for adding the listeners. Alternatively, the annotations can be processed at the bytecode level: A bytecode editor could inject the calls to `addActionListener` into the frame constructor. This sounds complex, but libraries are available to make this task relatively straightforward. You can see an example in Section 8.7, “Bytecode Engineering,” on p. 481.

Our example was not intended as a serious tool for user interface programmers. A utility method for adding a listener could be just as convenient for the programmer as the annotation. (In fact, the `java.beans.EventHandler` class tries to do just that. You could make the class truly useful by supplying a method that adds the event handler instead of just constructing it.)

However, this example shows the mechanics of annotating a program and of analyzing the annotations. Having seen a concrete example, you are now more prepared (we hope) for the following sections that describe the annotation syntax in complete detail.

Listing 8.9 runtimeAnnotations/ActionListenerInstaller.java

```
1 package runtimeAnnotations;
2
3 import java.awt.event.*;
4 import java.lang.reflect.*;
5
6 /**
7  * @version 1.00 2004-08-17
8  * @author Cay Horstmann
9  */
10 public class ActionListenerInstaller
11 {
12     /**
13      * Processes all ActionListenerFor annotations in the given object.
14      * @param obj an object whose methods may have ActionListenerFor annotations
15      */
16     public static void processAnnotations(Object obj)
17     {
18         try
19         {
20             Class<?> cl = obj.getClass();
21             for (Method m : cl.getDeclaredMethods())
22             {
23                 ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
24                 if (a != null)
25                 {
26                     Field f = cl.getDeclaredField(a.source());
27                     f.setAccessible(true);
28                     addListener(f.get(obj), obj, m);
29                 }
30             }
31         }
32         catch (ReflectiveOperationException e)
33         {
34             e.printStackTrace();
35         }
36     }
37
38     /**
39      * Adds an action listener that calls a given method.
40      * @param source the event source to which an action listener is added
41      * @param param the implicit parameter of the method that the listener calls
42      * @param m the method that the listener calls
43      */
44     public static void addListener(Object source, final Object param, final Method m)
45         throws ReflectiveOperationException
46     {
```

(Continues)

Listing 8.9 (*Continued*)

```
47     InvocationHandler handler = new InvocationHandler()
48     {
49         public Object invoke(Object proxy, Method mm, Object[] args) throws Throwable
50         {
51             return m.invoke(param);
52         }
53     };
54
55     Object listener = Proxy.newProxyInstance(null,
56         new Class[] { java.awt.event.ActionListener.class }, handler);
57     Method adder = source.getClass().getMethod("addActionListener", ActionListener.class);
58     adder.invoke(source, listener);
59 }
60 }
```

Listing 8.10 buttons3/ButtonFrame.java

```
1 package buttons3;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import runtimeAnnotations.*;
6
7 /**
8  * A frame with a button panel.
9  * @version 1.00 2004-08-17
10 * @author Cay Horstmann
11 */
12 public class ButtonFrame extends JFrame
13 {
14     private static final int DEFAULT_WIDTH = 300;
15     private static final int DEFAULT_HEIGHT = 200;
16
17     private JPanel panel;
18     private JButton yellowButton;
19     private JButton blueButton;
20     private JButton redButton;
21
22     public ButtonFrame()
23     {
24         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
25
26         panel = new JPanel();
27         add(panel);
28     }
}
```

```
29     yellowButton = new JButton("Yellow");
30     blueButton = new JButton("Blue");
31     redButton = new JButton("Red");
32
33     panel.add(yellowButton);
34     panel.add(blueButton);
35     panel.add(redButton);
36
37     ActionListenerInstaller.processAnnotations(this);
38 }
39
40 @ActionListenerFor(source = "yellowButton")
41 public void yellowBackground()
42 {
43     panel.setBackground(Color.YELLOW);
44 }
45
46 @ActionListenerFor(source = "blueButton")
47 public void blueBackground()
48 {
49     panel.setBackground(Color.BLUE);
50 }
51
52 @ActionListenerFor(source = "redButton")
53 public void redBackground()
54 {
55     panel.setBackground(Color.RED);
56 }
57 }
```

Listing 8.11 runtimeAnnotations/ActionListenerFor.java

```
1 package runtimeAnnotations;
2
3 import java.lang.annotation.*;
4
5 /**
6  * @version 1.00 2004-08-17
7  * @author Cay Horstmann
8  */
9
10 @Target(ElementType.METHOD)
11 @Retention(RetentionPolicy.RUNTIME)
12 public @interface ActionListenerFor
13 {
14     String source();
15 }
```

java.lang.reflect.AnnotatedElement 5.0

- `boolean isAnnotationPresent(Class<? extends Annotation> annotationType)`
returns true if this item has an annotation of the given type.
- `<T extends Annotation> T getAnnotation(Class<T> annotationType)`
gets the annotation of the given type, or null if this item has no such annotation.
- `<T extends Annotation> T[] getAnnotationsByType(Class<T> annotationType) 8`
gets all annotations of a repeatable annotation type (see Section 8.5.3, “Meta-Annotations,” on p. 472), or an array of length 0.
- `Annotation[] getAnnotations()`
gets all annotations present for this item, including inherited annotations. If no annotations are present, an array of length 0 is returned.
- `Annotation[] getDeclaredAnnotations()`
gets all annotations declared for this item, excluding inherited annotations. If no annotations are present, an array of length 0 is returned.

8.4 Annotation Syntax

In the following sections, we cover everything you need to know about the annotation syntax.

8.4.1 Annotation Interfaces

An annotation is defined by an annotation interface:

```
modifiers @interface AnnotationName
{
    elementDeclaration1
    elementDeclaration2
    ...
}
```

Each element declaration has the form

```
type elementName();
```

or

```
type elementName() default value;
```

For example, the following annotation has two elements, `assignedTo` and `severity`:

```
public @interface BugReport
{
    String assignedTo() default "[none]";
    int severity();
}
```

All annotation interfaces implicitly extend the `java.lang.annotation.Annotation` interface. That interface is a regular interface, *not* an annotation interface. See the API notes at the end of this section for the methods provided by this interface. You cannot extend annotation interfaces. In other words, all annotation interfaces directly extend `java.lang.annotation.Annotation`. You never supply classes that implement annotation interfaces.

The methods of an annotation interface have no parameters and no `throws` clauses. They cannot be `default` or `static` methods, and they cannot have type parameters.

The type of an annotation element is one of the following:

- A primitive type (`int`, `short`, `long`, `byte`, `char`, `double`, `float`, or `boolean`)
- `String`
- `Class` (with an optional type parameter such as `Class<? extends MyClass>`)
- An `enum` type
- An annotation type
- An array of the preceding types (an array of arrays is not a legal element type)

Here are examples of valid element declarations:

```
public @interface BugReport
{
    enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
    boolean showStopper() default false;
    String assignedTo() default "[none]";
    Class<?> testCase() default Void.class;
    Status status() default Status.UNCONFIRMED;
    Reference ref() default @Reference(); // an annotation type
    String[] reportedBy();
}
```

`java.lang.annotation.Annotation` 5.0

- `Class<? extends Annotation> annotationType()`

returns the `Class` object that represents the annotation interface of this annotation object. Note that calling `getClass` on an annotation object would return the actual class, not the interface.

(Continues)

java.lang.annotation.Annotation 5.0 (Continued)

- `boolean equals(Object other)`
returns true if other is an object that implements the same annotation interface as this annotation object and if all elements of this object and other are equal.
- `int hashCode()`
returns a hash code, compatible with the equals method, derived from the name of the annotation interface and the element values.
- `String toString()`
returns a string representation that contains the annotation interface name and the element values; for example, `@BugReport(assignedTo=[none], severity=0)`.

8.4.2 Annotations

Each annotation has the format

`@AnnotationName(elementName1=value1, elementName2=value2, . . .)`

For example,

`@BugReport(assignedTo="Harry", severity=10)`

The order of the elements does not matter. The annotation

`@BugReport(severity=10, assignedTo="Harry")`

is identical to the preceding one.

The default value of the declaration is used if an element value is not specified. For example, consider the annotation

`@BugReport(severity=10)`

The value of the `assignedTo` element is the string "`[none]`".



CAUTION: Defaults are not stored with the annotation; instead, they are dynamically computed. For example, if you change the default for the `assignedTo` element to "`[]`" and recompile the `BugReport` interface, the annotation `@BugReport(severity=10)` will use the new default, even in class files that have been compiled before the default changed.

Two special shortcuts can simplify annotations.

If no elements are specified, either because the annotation doesn't have any or because all of them use the default value, you don't need to use parentheses. For example,

```
@BugReport
```

is the same as

```
@BugReport(assignedTo="[none]", severity=0)
```

Such an annotation is called a *marker annotation*.

The other shortcut is the *single value annotation*. If an element has the special name `value` and no other element is specified, you can omit the element name and the `=` symbol. For example, had we defined the `ActionListenerFor` annotation interface of the preceding section as

```
public @interface ActionListenerFor
{
    String value();
}
```

then the annotations could be written as

```
@ActionListenerFor("yellowButton")
```

instead of

```
@ActionListenerFor(value="yellowButton")
```

An item can have multiple annotations:

```
@Test
@BugReport(showStopper=true, reportedBy="Joe")
public void checkRandomInsertions()
```

If the author of an annotation declared it to be repeatable, you can repeat the same annotation multiple times:

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
public void checkRandomInsertions()
```

NOTE: Since annotations are evaluated by the compiler, all element values must be compile-time constants. For example,

```
@BugReport(showStopper=true, assignedTo="Harry", testCase=MyTestCase.class,
           status=BugReport.Status.CONFIRMED, . . .)
```



CAUTION: An annotation element can never be set to `null`. Not even a default of `null` is permissible. This can be rather inconvenient in practice. You will need to find other defaults, such as "" or `Void.class`.

If an element value is an array, enclose its values in braces:

```
@BugReport(. . ., reportedBy={"Harry", "Carl"})
```

You can omit the braces if the element has a single value:

```
@BugReport(. . ., reportedBy="Joe") // OK, same as {"Joe"}
```

Since an annotation element can be another annotation, you can build arbitrarily complex annotations. For example,

```
@BugReport(ref=@Reference(id="3352627"), . . .)
```

NOTE: It is an error to introduce circular dependencies in annotations. For example, `BugReport` has an element of the annotation type `Reference`, therefore `Reference` cannot have an element of type `BugReport`.

8.4.3 Annotating Declarations

There are many places where annotations can occur. They fall into two categories: *declarations* and *type uses*. Declaration annotations can appear at the declarations of

- Packages
- Classes (including `enum`)
- Interfaces (including annotation interfaces)
- Methods
- Constructors
- Instance fields (including `enum` constants)
- Local variables
- Parameter variables
- Type parameters

For classes and interfaces, put the annotations before the `class` or `interface` keyword:

```
@Entity public class User { . . . }
```

For variables, put them before the type:

```
@SuppressWarnings("unchecked") List<User> users = . . .;
public User getUser(@Param("id") String userId)
```

A type parameter in a generic class or method can be annotated like this:

```
public class Cache<@Immutable V> { . . . }
```

A package is annotated in a file `package-info.java` that contains only the package statement preceded by annotations.

```
/**  
 * Package-level Javadoc  
 */  
@GPL(version="3")  
package com.horstmann.corejava;  
import org.gnu.GPL;
```

NOTE: Annotations for local variables can only be processed at the source level. Class files do not describe local variables. Therefore, all local variable annotations are discarded when a class is compiled. Similarly, annotations for packages are not retained beyond the source level.

8.4.4 Annotating Type Uses

A declaration annotation provides some information about the item being declared. For example, in the declaration

```
public User getUser(@NonNull String userId)
```

it is asserted that the `userId` parameter is not null.

NOTE: The `@NonNull` annotation is a part of the Checker Framework (<http://types.cs.washington.edu/checker-framework>). With that framework, you can include assertions in your program, such that a parameter is non-null or that a `String` contains a regular expression. A static analysis tool then checks whether the assertions are valid in a given body of source code.

Now suppose we have a parameter of type `List<String>`, and we want to express that all of the strings are non-null. That is where type use annotations come in. Place the annotation before the type argument: `List<@NonNull String>`.

Type use annotations can appear in the following places:

- With generic type arguments: `List<@NonNull String>`, `Comparator.<@NonNull String reverseOrder()`.
- In any position of an array: `@NonNull String[][] words` (`words[i][j]` is not null), `String @NonNull [][] words` (`words` is not null), `String[] @NonNull [] words` (`words[i]` is not null).

- With superclasses and implemented interfaces: `class Warning extends @Localized Message.`
- With constructor invocations: `new @Localized String(. . .).`
- With casts and `instanceof` checks: `(@Localized String) text, if (text instanceof @Localized String).` (The annotations are only for use by external tools. They have no effect on the behavior of a cast or an `instanceof` check.)
- With exception specifications: `public String read() throws @Localized IOException.`
- With wildcards and type bounds: `List<@Localized ? extends Message>, List<? extends @Localized Message>.`
- With method and constructor references: `@Localized Message::getText.`

There are a few type positions that cannot be annotated:

```
@NotNull String.class // ERROR: Cannot annotate class literal  
import java.lang.@NotNull String; // ERROR: Cannot annotate import
```

You can place annotations before or after other modifiers such as `private` and `static`. It is customary (but not required) to put type use annotations after other modifiers, and declaration annotations before other modifiers. For example,

```
private @NotNull String text; // Annotates the type use  
@Id private String userId; // Annotates the variable
```

NOTE: An annotation author needs to specify where a particular annotation can appear. If an annotation is permissible both for a variable and a type use, and it is used in a variable declaration, then both the variable and the type use are annotated. For example, consider

```
public User getUser(@NotNull String userId)
```

If `@NotNull` can apply both to parameters and to type uses, the `userId` parameter is annotated, and the parameter type is `@NotNull String`.

8.4.5 Annotating this

Suppose you want to annotate parameters that are not being mutated by a method.

```
public class Point  
{  
    public boolean equals(@ReadOnly Object other) { . . . }
```

Then a tool that processes this annotation would, upon seeing a call

```
p.equals(q)
```

reason that `q` has not been changed.

But what about `p`?

When the method is called, the `this` variable is bound to `p`. But `this` is never declared, so you cannot annotate it.

Actually, you can declare it, with a rarely used syntax variant, just so that you can add an annotation:

```
public class Point
{
    public boolean equals(@ReadOnly Point this, @ReadOnly Object other) { . . . }
}
```

The first parameter is called the *receiver parameter*. It must be named `this`. Its type is the class that is being constructed.

NOTE: You can provide a receiver parameter only for methods, not for constructors. Conceptually, the `this` reference in a constructor is not an object of the given type until the constructor has completed. Instead, an annotation placed on the constructor describes a property of the constructed object.

A different hidden parameter is passed to the constructor of an inner class, namely the reference to the enclosing class object. You can make that parameter explicit as well:

```
public class Sequence
{
    private int from;
    private int to;

    class Iterator implements java.util.Iterator<Integer>
    {
        private int current;

        public Iterator(@ReadOnly Sequence Sequence.this)
        {
            this.current = Sequence.this.from;
        }

        . .
    }
}
```

The parameter must be named just like when you refer to it, `EnclosingClass.this`, and its type is the enclosing class.

8.5 Standard Annotations

Java SE defines a number of annotation interfaces in the `java.lang`, `java.lang.annotation`, and `javax.annotation` packages. Four of them are meta-annotations that describe the behavior of annotation interfaces. The others are regular annotations that you can use to annotate items in your source code. Table 8.2 shows these annotations. We'll discuss them in detail in the following two sections.

Table 8.2 The Standard Annotations

Annotation Interface	Applicable To	Purpose
Deprecated	All	Marks item as deprecated.
SuppressWarnings	All but packages and annotations	Suppresses warnings of the given type.
SafeVarargs	Methods and constructors	Asserts that the varargs parameter is safe to use.
Override	Methods	Checks that this method overrides a superclass method.
FunctionalInterface	Interfaces	Marks an interface as functional (with a single abstract method).
PostConstruct PreDestroy	Methods	The marked method should be invoked immediately after construction or before removal.
Resource	Classes, interfaces, methods, fields	On a class or interface, marks it as a resource to be used elsewhere. On a method or field, marks it for “injection.”
Resources	Classes, interfaces	Specifies an array of resources.
Generated	All	Marks an item as source code that has been generated by a tool.
Target	Annotations	Specifies the items to which this annotation can be applied.
Retention	Annotations	Specifies how long this annotation is retained.
Documented	Annotations	Specifies that this annotation should be included in the documentation of annotated items.

(Continues)

Table 8.2 (Continued)

Annotation Interface	Applicable To	Purpose
Inherited	Annotations	Specifies that this annotation, when applied to a class, is automatically inherited by its subclasses.
Repeatable	Annotations	Specifies that this annotation can be applied multiple times to the same item.

8.5.1 Annotations for Compilation

The `@Deprecated` annotation can be attached to any items whose use is no longer encouraged. The compiler will warn when you use a deprecated item. This annotation has the same role as the `@deprecated` Javadoc tag.

The `@SuppressWarnings` annotation tells the compiler to suppress warnings of a particular type, for example,

```
@SuppressWarnings("unchecked")
```

The `@Override` annotation applies only to methods. The compiler checks that a method with this annotation really overrides a method from the superclass. For example, if you declare

```
public MyClass
{
    @Override public boolean equals(MyClass other);
    ...
}
```

then the compiler will report an error. After all, the `equals` method does *not* override the `equals` method of the `Object` class because that method has a parameter of type `Object`, not `MyClass`.

The `@Generated` annotation is intended for use by code generator tools. Any generated source code can be annotated to differentiate it from programmer-provided code. For example, a code editor can hide the generated code, or a code generator can remove older versions of generated code. Each annotation must contain a unique identifier for the code generator. A date string (in ISO 8601 format) and a comment string are optional. For example,

```
@Generated("com.horstmann.beanproperty", "2008-01-04T12:08:56.235-0700");
```

8.5.2 Annotations for Managing Resources

The `@PostConstruct` and `@PreDestroy` annotations are used in environments that control the lifecycle of objects, such as web containers and application servers. Methods tagged with these annotations should be invoked immediately after an object has been constructed or immediately before it is being removed.

The `@Resource` annotation is intended for resource injection. For example, consider a web application that accesses a database. Of course, the database access information should not be hardwired into the web application. Instead, the web container has some user interface for setting connection parameters and a JNDI name for a data source. In the web application, you can reference the data source like this:

```
@Resource(name="jdbc/mydb")
private DataSource source;
```

When an object containing this field is constructed, the container “injects” a reference to the data source.

8.5.3 Meta-Annotations

The `@Target` meta-annotation is applied to an annotation, restricting the items to which the annotation applies. For example,

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

Table 8.3 shows all possible values. They belong to the enumerated type `ElementType`. You can specify any number of element types, enclosed in braces.

An annotation without an `@Target` restriction can be applied to any item. The compiler checks that you apply an annotation only to a permitted item. For example, if you apply `@BugReport` to a field, a compile-time error results.

The `@Retention` meta-annotation specifies how long an annotation is retained. You can specify at most one of the values in Table 8.4. The default is `RetentionPolicy.CLASS`.

In Listing 8.11 on p. 461, the `@ActionListenerFor` annotation was declared with `RetentionPolicy.RUNTIME` because we used reflection to process annotations. In the following two sections, you will see examples of processing annotations at the source and class file levels.

The `@Documented` meta-annotation gives a hint to documentation tools such as Javadoc. Documented annotations should be treated just like other modifiers, such as `protected` or `static`, for documentation purposes. The use of other annotations is not

Table 8.3 Element Types for the @Target Annotation

Element Type	Annotation Applies To
ANNOTATION_TYPE	Annotation type declarations
PACKAGE	Packages
TYPE	Classes (including enum) and interfaces (including annotation types)
METHOD	Methods
CONSTRUCTOR	Constructors
FIELD	Fields (including enum constants)
PARAMETER	Method or constructor parameters
LOCAL_VARIABLE	Local variables
TYPE_PARAMETER	Type parameters
TYPE_USE	Uses of a type

Table 8.4 Retention Policies for the @Retention Annotation

Retention Policy	Description
SOURCE	Annotations are not included in class files.
CLASS	Annotations are included in class files, but the virtual machine need not load them.
RUNTIME	Annotations are included in class files and loaded by the virtual machine. They are available through the reflection API.

included in the documentation. For example, suppose we declare @ActionListenerFor as a documented annotation:

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

Now the documentation of each annotated method contains the annotation, as shown in Figure 8.2.

If an annotation is transient (such as @BugReport), you should probably not document its use.

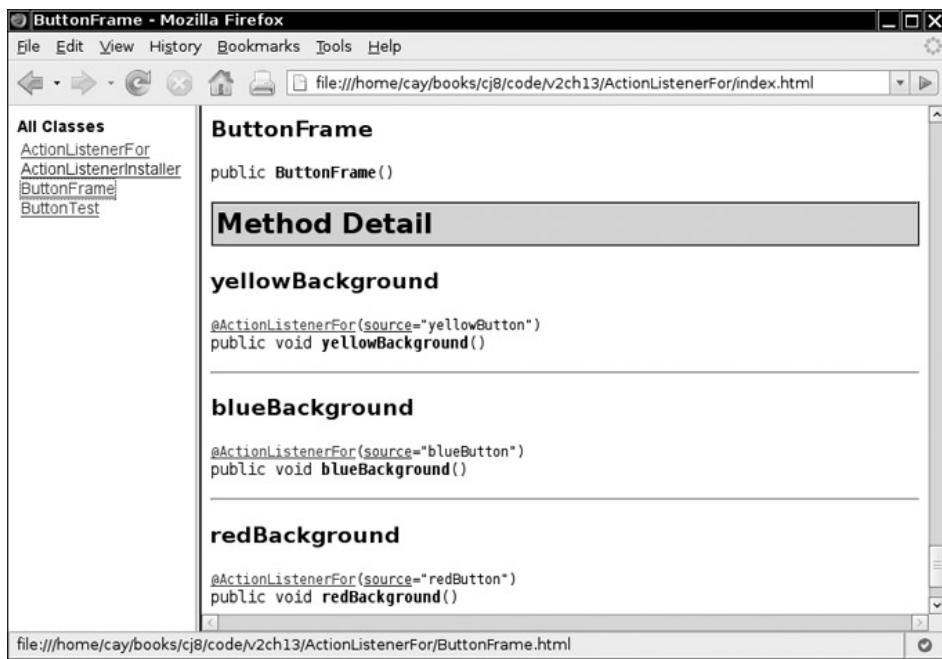


Figure 8.2 Documented annotations

NOTE: It is legal to apply an annotation to itself. For example, the @Documented annotation is itself annotated as @Documented. Therefore, the Javadoc documentation for annotations shows whether they are documented.

The @Inherited meta-annotation applies only to annotations for classes. When a class has an inherited annotation, then all of its subclasses automatically have the same annotation. This makes it easy to create annotations that work as marker interfaces, such as Serializable.

In fact, an annotation @Serializable would be more appropriate than the Serializable marker interface with no methods. A class is serializable because there is runtime support for reading and writing its fields, not because of any principles of object-oriented design. An annotation describes this fact better than does interface inheritance. Of course, the Serializable interface was created in JDK 1.1, long before annotations existed.

Suppose you define an inherited annotation @Persistent to indicate that objects of a class can be saved in a database. Then the subclasses of persistent classes are automatically annotated as persistent.

```
@Inherited @interface Persistent { }
@Persistent class Employee { . . . }
class Manager extends Employee { . . . } // also @Persistent
```

When the persistence mechanism searches for objects to store in the database, it will detect both `Employee` and `Manager` objects.

As of Java SE 8, it is legal to apply the same annotation type multiple times to an item. For backward compatibility, the implementor of a repeatable annotation needs to provide a *container annotation* that holds the repeated annotations in an array.

Here is how to define the `@TestCase` annotation and its container:

```
@Repeatable(TestCases.class)
@interface TestCase
{
    String params();
    String expected();
}

@interface TestCases
{
    TestCase[] value();
}
```

Whenever the user supplies two or more `@TestCase` annotations, they are automatically wrapped into a `@TestCases` annotation.



CAUTION: You have to be careful when processing repeatable annotations. If you call `getAnnotation` to look up a repeatable annotation, and the annotation was actually repeated, then you get `null`. That is because the repeated annotations were wrapped into the container annotation.

In that case, you should call `getAnnotationsByType`. That call “looks through” the container and gives you an array of the repeated annotations. If there was just one annotation, you get it in an array of length 1. With this method, you don’t have to worry about the container annotation.

8.6 Source-Level Annotation Processing

In the preceding section, you saw how to analyze annotations in a running program. Another use for annotation is the automatic processing of source files to produce more source code, configuration files, scripts, or whatever else one might want to generate.

8.6.1 Annotation Processors

Annotation processing is integrated into the Java compiler. During compilation, you can *invoke annotation processors* by running

```
javac -processor ProcessorClassName1,ProcessorClassName2,... sourceFiles
```

The compiler locates the annotations of the source files. Each annotation processor is executed in turn and given the annotations in which it expressed an interest. If an annotation processor creates a new source file, the process is repeated. Once a processing round yields no further source files, all source files are compiled.

NOTE: An annotation processor can only generate new source files. It cannot modify an existing source file.

An annotation processor implements the `Processor` interface, generally by extending the `AbstractProcessor` class. You need to specify which annotations your processor supports. In our case:

```
@SupportedAnnotationTypes("com.horstmann.annotations.ToString")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class ToStringAnnotationProcessor extends AbstractProcessor
{
    public boolean process(Set<? extends TypeElement> annotations,
                          RoundEnvironment currentRound)
    {
        ...
    }
}
```

A processor can claim specific annotation types, wildcards such as "com.horstmann.*" (all annotations in the `com.horstmann` package or any subpackage), or even "*" (all annotations).

The `process` method is called once for each round, with the set of all annotations that were found in any files during this round, and a `RoundEnvironment` reference that contains information about the current processing round.

8.6.2 The Language Model API

Use the *language model* API for analyzing source-level annotations. Unlike the reflection API, which presents the virtual machine representation of classes and methods, the language model API lets you analyze a Java program according to the rules of the Java language.

The compiler produces a tree whose nodes are instances of classes that implement the `javax.lang.model.element.Element` interface and its subinterfaces: `TypeElement`, `VariableElement`, `ExecutableElement`, and so on. These are the compile-time analogs to the `Class`, `Field`/`Parameter`, `Method`/`Constructor` reflection classes.

I do not want to cover the API in detail, but here are the highlights that you need to know for processing annotations:

- The `RoundEnvironment` gives you a set of all elements annotated with a particular annotation, by calling the method

```
Set<? extends Element> getElementsAnnotatedWith(Class<? extends Annotation> a)
```

- The source-level equivalent of the `AnnotateElement` interface is `AnnotatedConstruct`. Use the methods

```
A getAnnotation(Class<A> annotationType)  
A[] getAnnotationsByType(Class<A> annotationType)
```

to get the annotation or repeated annotations for a given annotation class.

- A `TypeElement` represents a class or interface. The `getEnclosedElements` method yields a list of its fields and methods.
- Calling `getSimpleName` on an `Element` or `getQualifiedName` on a `TypeElement` yields a `Name` object that can be converted to a string with `toString`.

8.6.3 Using Annotations to Generate Source Code

As an example, we will use annotations to reduce the tedium of implementing `toString` methods. We can't put these methods into the original classes—annotation processors can only produce new classes, not modify existing ones.

Therefore, we'll add all methods into a utility class `ToStrings`:

```
public class ToStrings  
{  
    public static String toString(Point obj)  
    {  
        Generated code  
    }  
    public static String toString(Rectangle obj)  
    {  
        Generated code  
    }  
    ...
```

```
public static String toString(Object obj)
{
    return Objects.toString(obj);
}
```

We don't want to use reflection, so we annotate accessor methods, not fields:

```
@ToString
public class Rectangle
{
    ...
    @ToString(includeName=false) public Point getTopLeft() { return topLeft; }
    @ToString public int getWidth() { return width; }
    @ToString public int getHeight() { return height; }
}
```

The annotation processor should then generate the following source code:

```
public static String toString(Rectangle obj)
{
    StringBuilder result = new StringBuilder();
    result.append("Rectangle");
    result.append("[");
    result.append(toString(obj.getTopLeft()));
    result.append(",");
    result.append("width=");
    result.append(toString(obj.getWidth()));
    result.append(",");
    result.append("height=");
    result.append(toString(obj.getHeight()));
    result.append("]");
    return result.toString();
}
```

The “boilerplate” code is in gray. Here is an outline of the method that produces the `toString` method for a class with given `TypeElement`:

```
private void writeToStringMethod(PrintWriter out, TypeElement te)
{
    String className = te.getQualifiedName().toString();
    Print method header and declaration of string builder
    ToString ann = te.getAnnotation(ToString.class);
    if (ann.includeName())
        Print code to add class name
    for (Element c : te.getEnclosedElements())
    {
        ann = c.getAnnotation(ToString.class);
        if (ann != null)
        {
```

```

        if (ann.includeName()) Print code to add field name
        Print code to append toString(obj.methodName())
    }
}
Print code to return string
}

```

And here is an outline of the process method of the annotation processor. It creates a source file for the helper class and writes the class header and one method for each annotated class.

```

public boolean process(Set<? extends TypeElement> annotations,
                      RoundEnvironment currentRound)
{
    if (annotations.size() == 0) return true;
    try
    {
        JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(
            "com.horstmann.annotations.ToStrings");
        try (PrintWriter out = new PrintWriter(sourceFile.openWriter()))
        {
            Print code for package and class
            for (Element e : currentRound.getElementsAnnotatedWith(ToString.class))
            {
                if (e instanceof TypeElement)
                {
                    TypeElement te = (TypeElement) e;
                    writeToStringMethod(out, te);
                }
            }
            Print code for toString(Object)
        }
        catch (IOException ex)
        {
            processingEnv.getMessager().printMessage(
                Kind.ERROR, ex.getMessage());
        }
    }
    return true;
}

```

For the tedious details, check the book's companion code.

Note that the process method is called in subsequent rounds with an empty list of annotations. It then returns immediately so it doesn't create the source file twice.

First compile the annotation processor, and then compile and run the test program as follows:

```
javac sourceAnnotations/ToStringAnnotationProcessor.java  
javac -processor sourceAnnotations.ToStringAnnotationProcessor rect/*.java  
java rect.SourceLevelAnnotationDemo
```



TIP: To see the rounds, run the javac command with the `-XprintRounds` flag:

```
Round 1:  
  input files: {rect.Point, rect.Rectangle,  
                rect.SourceLevelAnnotationDemo}  
  annotations: [sourceAnnotations.ToString]  
  last round: false  
Round 2:  
  input files: {sourceAnnotations.ToStrings}  
  annotations: []  
  last round: false  
Round 3:  
  input files: {}  
  annotations: []  
  last round: true
```

This example demonstrates how tools can harvest source file annotations to produce other files. The generated files don't have to be source files. Annotation processors may choose to generate XML descriptors, property files, shell scripts, HTML documentation, and so on.

NOTE: Some people have suggested using annotations to remove an even bigger drudgery. Wouldn't it be nice if trivial getters and setters were generated automatically? For example, the annotation

```
@Property private String title;  
could produce the methods  
public String getTitle() { return title; }  
public void setTitle(String title) { this.title = title; }
```

However, those methods need to be added to the *same class*. This requires editing a source file, not just generating another file, and is beyond the capabilities of annotation processors. It would be possible to build another tool for this purpose, but such a tool would go beyond the mission of annotations. An annotation is intended as a description *about* a code item, not a directive for adding or changing code.

8.7 Bytecode Engineering

You have seen how annotations can be processed at runtime or at the source code level. There is a third possibility: processing at the bytecode level. Unless annotations are removed at the source level, they are present in the class files. The class file format is documented (see <http://docs.oracle.com/javase/specs/jvms/se8/html>). The format is rather complex, and it would be challenging to process class files without special libraries. One such library is the ASM library, available at <http://asm.ow2.org>.

8.7.1 Modifying Class Files

In this section, we use ASM to add logging messages to annotated methods. If a method is annotated with

```
@LogEntry(logger=loggerName)
```

then we add the bytecodes for the following statement at the beginning of the method:

```
Logger.getLogger(loggerName).entering(className, methodName);
```

For example, if you annotate the `hashCode` method of the `Item` class as

```
@LogEntry(logger="global") public int hashCode()
```

then a message similar to the following is printed whenever the method is called:

```
May 17, 2016 10:57:59 AM Item hashCode  
FINER: ENTRY
```

To achieve this, we do the following:

1. Load the bytecodes in the class file.
2. Locate all methods.
3. For each method, check whether it has a `LogEntry` annotation.
4. If it does, add the bytecodes for the following instructions at the beginning of the method:

```
ldc loggerName  
invokestatic  
    java/util/logging/Logger.getLogger:(Ljava/lang/String;)Ljava/util/logging/Logger;  
ldc className  
ldc methodName  
invokevirtual java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/lang/String;)V
```

Inserting these bytecodes sounds tricky, but ASM makes it fairly straightforward. We don't describe the process of analyzing and inserting bytecodes in detail. The

important point is that the program in Listing 8.12 edits a class file and inserts a logging call at the beginning of the methods annotated with the `LogEntry` annotation.

For example, here is how you add the logging instructions to `Item.java` in Listing 8.13, where `asm` is the directory into which you installed the ASM library:

```
javac set/Item.java
javac -classpath ..:asm/lib/* bytecodeAnnotations/EntryLogger.java
java -classpath ..:asm/lib/* bytecodeAnnotations.EntryLogger set.Item
```

Try running

```
javap -c set.Item
```

before and after modifying the `Item` class file. You can see the inserted instructions at the beginning of the `hashCode`, `equals`, and `compareTo` methods.

```
public int hashCode();
  Code:
  0: ldc #85; // String global
  2: invokespecial #80;
     // Method java/util/logging/Logger.getLogger:(Ljava/lang/String;)Ljava/util/logging/Logger;
  5: ldc #86; //String Item
  7: ldc #88; //String hashCode
  9: invokevirtual #84;
     // Method java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/lang/String;)V
 12: bipush 13
 14: aload_0
 15: getfield      #2; // Field description:Ljava/lang/String;
 18: invokevirtual #15; // Method java/lang/String.hashCode():I
 21: imul
 22: bipush 17
 24: aload_0
 25: getfield      #3; // Field partNumber:I
 28: imul
 29: iadd
 30: ireturn
```

The `SetTest` program in Listing 8.14 inserts `Item` objects into a hash set. When you run it with the modified class file, you will see the logging messages.

```
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item equals
FINER: ENTRY
[[description=Toaster, partNumber=1729], [description=Microwave, partNumber=4104]]
```

Note the call to `equals` when we insert the same item twice.

This example shows the power of bytecode engineering. Annotations are used to add directives to a program, and a bytecode editing tool picks up the directives and modifies the virtual machine instructions.

Listing 8.12 `bytecodeAnnotations/EntryLogger.java`

```
1 package bytecodeAnnotations;
2
3 import java.io.*;
4 import java.nio.file.*;
5
6 import org.objectweb.asm.*;
7 import org.objectweb.asm.commons.*;
8
9 /**
10  * Adds "entering" logs to all methods of a class that have the LogEntry annotation.
11  * @version 1.20 2016-05-10
12  * @author Cay Horstmann
13  */
14 public class EntryLogger extends ClassVisitor
15 {
16     private String className;
17
18     /**
19      * Constructs an EntryLogger that inserts logging into annotated methods of a given class.
20      * @param cg the class
21      */
22     public EntryLogger(ClassWriter writer, String className)
23     {
24         super(OpcodesASM5, writer);
25         this.className = className;
26     }
27
28     @Override
29     public MethodVisitor visitMethod(int access, String methodName, String desc,
30         String signature, String[] exceptions)
31     {
32         MethodVisitor mv = cv.visitMethod(access, methodName, desc, signature, exceptions);
33         return new AdviceAdapter(OpcodesASM5, mv, access, methodName, desc)
34     {
35         private String loggerName;
36
37         public AnnotationVisitor visitAnnotation(String desc, boolean visible)
38         {
39             return new AnnotationVisitor(OpcodesASM5)
40             {
```

(Continues)

Listing 8.12 (Continued)

```
41         public void visit(String name, Object value)
42         {
43             if (desc.equals("LbytecodeAnnotations/LogEntry;") && name.equals("logger"))
44                 loggerName = value.toString();
45             }
46         };
47     }
48
49     public void onMethodEnter()
50     {
51         if (loggerName != null)
52         {
53             visitLdcInsn(loggerName);
54             visitMethodInsn(INVOKESTATIC, "java/util/logging/Logger", "getLogger",
55                             "(Ljava/lang/String;)Ljava/util/logging/Logger;", false);
56             visitLdcInsn(className);
57             visitLdcInsn(methodName);
58             visitMethodInsn(INVOKEVIRTUAL, "java/util/logging/Logger", "entering",
59                             "(Ljava/lang/String;Ljava/lang/String;)V", false);
60             loggerName = null;
61         }
62     }
63 };
64 }
65 /**
66 * Adds entry logging code to the given class.
67 * @param args the name of the class file to patch
68 */
69 public static void main(String[] args) throws IOException
70 {
71     if (args.length == 0)
72     {
73         System.out.println("USAGE: java bytecodeAnnotations.EntryLogger classfile");
74         System.exit(1);
75     }
76     Path path = Paths.get(args[0]);
77     ClassReader reader = new ClassReader(Files.newInputStream(path));
78     ClassWriter writer = new ClassWriter(
79         ClassWriter.COMPUTE_MAXS | ClassWriter.COMPUTE_FRAMES);
80     EntryLogger entryLogger = new EntryLogger(writer,
81         path.toString().replace(".class", "").replaceAll("[\\\\\\\\]", "."));
82     reader.accept(entryLogger, ClassReader.EXPAND_FRAMES);
83     Files.write(Paths.get(args[0]), writer.toByteArray());
84 }
85 }
86 }
```

Listing 8.13 set/Item.java

```
1 package set;
2
3 import java.util.*;
4 import bytecodeAnnotations.*;
5
6 /**
7  * An item with a description and a part number.
8  * @version 1.01 2012-01-26
9  * @author Cay Horstmann
10 */
11 public class Item
12 {
13     private String description;
14     private int partNumber;
15
16     /**
17      * Constructs an item.
18      * @param aDescription the item's description
19      * @param aPartNumber the item's part number
20      */
21     public Item(String aDescription, int aPartNumber)
22     {
23         description = aDescription;
24         partNumber = aPartNumber;
25     }
26
27     /**
28      * Gets the description of this item.
29      * @return the description
30      */
31     public String getDescription()
32     {
33         return description;
34     }
35
36     public String toString()
37     {
38         return "[description=" + description + ", partNumber=" + partNumber + "]";
39     }
40
41     @LogEntry(logger = "com.horstmann")
42     public boolean equals(Object otherObject)
43     {
44         if (this == otherObject) return true;
45         if (otherObject == null) return false;
```

(Continues)

Listing 8.13 (Continued)

```
46     if (getClass() != otherObject.getClass()) return false;
47     Item other = (Item) otherObject;
48     return Objects.equals(description, other.description) && partNumber == other.partNumber;
49   }
50
51   @LogEntry(logger = "com.horstmann")
52   public int hashCode()
53   {
54     return Objects.hash(description, partNumber);
55   }
56 }
```

Listing 8.14 set/SetTest.java

```
1 package set;
2
3 import java.util.*;
4 import java.util.logging.*;
5
6 /**
7 * @version 1.02 2012-01-26
8 * @author Cay Horstmann
9 */
10 public class SetTest
11 {
12   public static void main(String[] args)
13   {
14     Logger.getLogger("com.horstmann").setLevel(Level.FINEST);
15     Handler handler = new ConsoleHandler();
16     handler.setLevel(Level.FINEST);
17     Logger.getLogger("com.horstmann").addHandler(handler);
18
19     Set<Item> parts = new HashSet<>();
20     parts.add(new Item("Toaster", 1279));
21     parts.add(new Item("Microwave", 4104));
22     parts.add(new Item("Toaster", 1279));
23     System.out.println(parts);
24   }
25 }
```

8.7.2 Modifying Bytecodes at Load Time

In the preceding section, you saw a tool that edits class files. However, it can be cumbersome to add yet another tool into the build process. An attractive

alternative is to defer the bytecode engineering until *load time*, when the class loader loads the class.

The *instrumentation API* has a hook for installing a bytecode transformer. The transformer must be installed before the `main` method of the program is called. You can meet this requirement by defining an *agent*, a library that is loaded to monitor a program in some way. The agent code can carry out initializations in a `premain` method.

Here are the steps required to build an agent:

1. Implement a class with a method

```
public static void premain(String arg, Instrumentation instr)
```

This method is called when the agent is loaded. The agent can get a single command-line argument, which is passed in the `arg` parameter. The `instr` parameter can be used to install various hooks.

2. Make a manifest file `EntryLoggingAgent.mf` that sets the `Premain-Class` attribute, for example:

```
Premain-Class: bytecodeAnnotations.EntryLoggingAgent
```

3. Package the agent code and the manifest into a JAR file:

```
javac -classpath .:asm/lib/* bytecodeAnnotations/EntryLoggingAgent.java  
jar cvfm EntryLoggingAgent.jar bytecodeAnnotations/EntryLoggingAgent.mf \  
bytecodeAnnotations/Entry*.class
```

To launch a Java program together with the agent, use the following command-line options:

```
java -javaagent:AgentJARFile=agentArgument . . .
```

For example, to run the `SetTest` program with the entry logging agent, call

```
javac set/SetTest.java  
java -javaagent:EntryLoggingAgent.jar=set.Item -classpath .:asm/lib/* set.SetTest
```

The `Item` argument is the name of the class that the agent should modify.

Listing 8.15 shows the agent code. The agent installs a class file transformer. The transformer first checks whether the class name matches the `agent` argument. If so, it uses the `EntryLogger` class from the preceding section to modify the bytecodes. However, the modified bytecodes are not saved to a file. Instead, the transformer returns them for loading into the virtual machine (see Figure 8.3). In other words, this technique carries out “just in time” modification of the bytecodes.

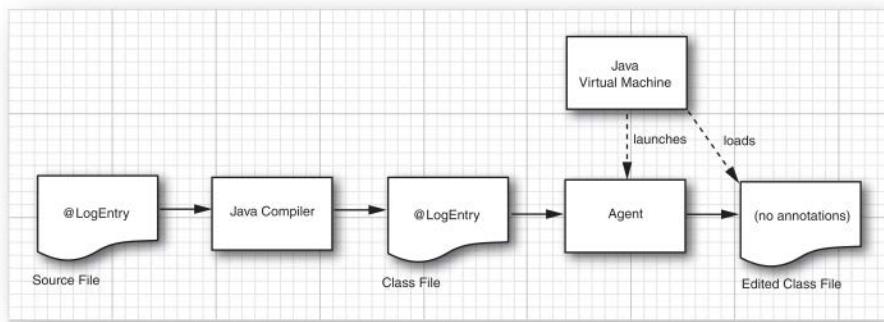


Figure 8.3 Modifying classes at load time

Listing 8.15 bytecodeAnnotations/EntryLoggingAgent.java

```
1 package bytecodeAnnotations;
2
3 import java.lang.instrument.*;
4
5 import org.objectweb.asm.*;
6
7 /**
8 * @version 1.10 2016-05-10
9 * @author Cay Horstmann
10 */
11 public class EntryLoggingAgent
12 {
13     public static void premain(final String arg, Instrumentation instr)
14     {
15         instr.addTransformer((loader, className, cl, pd, data) ->
16         {
17             if (!className.equals(arg)) return null;
18             ClassReader reader = new ClassReader(data);
19             ClassWriter writer = new ClassWriter(
20                 ClassWriter.COMPUTE_MAXS | ClassWriter.COMPUTE_FRAMES);
21             EntryLogger el = new EntryLogger(writer, className);
22             reader.accept(el, ClassReader.EXPAND_FRAMES);
23             return writer.toByteArray();
24         });
25     }
26 }
```

In this chapter, you have learned how to

- Add annotations to Java programs
- Design your own annotation interfaces
- Implement tools that make use of the annotations

You have seen three technologies for processing code: scripting, compiling Java programs, and processing annotations. The first two were quite straightforward. On the other hand, building annotation tools is undeniably complex and not something that most developers will need to tackle. This chapter gave you the background for understanding the inner workings of the annotation tools you will encounter, and perhaps piqued your interest in developing your own tools.

In the next chapter, we'll move on to an entirely different topic: security. Security has always been a core feature of the Java platform. As the world in which we live and compute gets more dangerous, a thorough understanding of Java security will be of increasing importance for many developers.

This page intentionally left blank

Security

In this chapter

- 9.1 Class Loaders, page 492
- 9.2 Security Managers and Permissions, page 509
- 9.3 User Authentication, page 530
- 9.4 Digital Signatures, page 546
- 9.5 Encryption, page 567

When Java technology first appeared on the scene, the excitement was not about a well-crafted programming language but about the possibility of safely executing applets delivered over the Internet. Obviously, serving executable applets is only practical when the recipients can be sure the code won't wreak havoc on their machines. Security therefore was and is a major concern of both the designers and the users of Java technology. This means that unlike other languages and systems, where security was implemented as an afterthought or as a reaction to break-ins, security mechanisms are an integral part of Java technology.

Three mechanisms help ensure safety:

- Language design features (bounds checking on arrays, no unchecked type conversions, no pointer arithmetic, and so on).
- An access control mechanism that controls what the code can do (file access, network access, and so on).

- Code signing, whereby code authors can use standard cryptographic algorithms to authenticate Java code. Then, the users of the code can determine exactly who created the code and whether the code has been altered after it was signed.

We will first discuss *class loaders* that check class files for integrity when they are loaded into the virtual machine. We will demonstrate how that mechanism can detect tampering with class files.

For maximum security, both the default mechanism for loading a class and a custom class loader need to work with a *security manager* class that controls what actions code can perform. You'll see in detail how to configure Java platform security.

Finally, you'll see the cryptographic algorithms supplied in the `java.security` package, which allow for code signing and user authentication.

As always, we'll focus on those topics that are of greatest interest to application programmers. For an in-depth view, we recommend the book *Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation, Second Edition*, by Li Gong, Gary Ellison, and Mary Dageforde (Prentice Hall, 2003).

9.1 Class Loaders

A Java compiler converts source instructions into code for the Java virtual machine. The virtual machine code is stored in a class file with a `.class` extension. Each class file contains the definition and implementation code for one class or interface. In the following section, you will see how the virtual machine loads these class files.

9.1.1 The Class Loading Process

The virtual machine loads only those class files that are needed for the execution of a program. For example, suppose program execution starts with `MyProgram.class`. Here are the steps that the virtual machine carries out:

1. The virtual machine has a mechanism for loading class files—for example, by reading the files from disk or by requesting them from the Web; it uses this mechanism to load the contents of the `MyProgram` class file.
2. If the `MyProgram` class has fields or superclasses of another class type, their class files are loaded as well. (The process of loading all the classes that a given class depends on is called *resolving* the class.)

3. The virtual machine then executes the `main` method in `MyProgram` (which is static, so no instance of a class needs to be created).
4. If the `main` method or a method that `main` calls requires additional classes, these are loaded next.

The class loading mechanism doesn't just use a single class loader, however. Every Java program has at least three class loaders:

- The bootstrap class loader
- The extension class loader
- The system class loader (sometimes also called the application class loader)

The bootstrap class loader loads the system classes (typically, from the JAR file `rt.jar`). It is an integral part of the virtual machine and is usually implemented in C. There is no `ClassLoader` object corresponding to the bootstrap class loader. For example,

```
String.class.getClassLoader()  
returns null.
```

The extension class loader loads "standard extensions" from the `jre/lib/ext` directory. You can drop JAR files into that directory, and the extension class loader will find the classes in them, even without any class path. (Some people recommend this mechanism to avoid the "class path hell," but see the next cautionary note.)

The system class loader loads the application classes. It locates classes in the directories and JAR/ZIP files on the class path, as set by the `CLASSPATH` environment variable or the `-classpath` command-line option.

In Oracle Java implementation, the extension and system class loaders are implemented in Java. Both are instances of the `URLClassLoader` class.



CAUTION: You can run into grief if you drop a JAR file into the `jre/lib/ext` directory and one of its classes needs to load a class that is not a system or extension class. The extension class loader *does not use the class path*. Keep that in mind before you use the extension directory as a way to manage your class file hassles.

NOTE: In addition to all the places already mentioned, classes can be loaded from the *jre/lib/endorsed* directory. This mechanism can only be used to replace certain standard Java libraries (such as those for XML and CORBA support) with newer versions. See <http://docs.oracle.com/javase/8/docs/technotes/guides/standards> for details.

9.1.2 The Class Loader Hierarchy

Class loaders have a *parent/child* relationship. Every class loader except for the bootstrap one has a parent class loader. A class loader is supposed to give its parent a chance to load any given class and to only load it if the parent has failed. For example, when the system class loader is asked to load a system class (say, `java.util.ArrayList`), it first asks the extension class loader. That class loader first asks the bootstrap class loader. The bootstrap class loader finds and loads the class in `rt.jar`, so neither of the other class loaders searches any further.

Some programs have a plugin architecture in which certain parts of the code are packaged as optional plugins. If the plugins are packaged as JAR files, you can simply load the plugin classes with an instance of `URLClassLoader`.

```
URL url = new URL("file:///path/to/plugin.jar");
URLClassLoader pluginLoader = new URLClassLoader(new URL[] { url });
Class<?> cl = pluginLoader.loadClass("mypackage.MyClass");
```

Since no parent was specified in the `URLClassLoader` constructor, the parent of the `pluginLoader` is the system class loader. Figure 9.1 shows the hierarchy.

Most of the time, you don't have to worry about the class loader hierarchy. Generally, classes are loaded because they are required by other classes, and that process is transparent to you.

Occasionally, however, you need to intervene and specify a class loader. Consider this example:

- Your application code contains a helper method that calls `Class.forName(classNameString)`.
- That method is called from a plugin class.
- The `classNameString` specifies a class that is contained in the plugin JAR.

The author of the plugin has reasons to expect that the class should be loaded. However, the helper method's class was loaded by the system class loader, and that is the class loader used by `Class.forName`. The classes in the plugin JAR are not visible. This phenomenon is called *classloader inversion*.

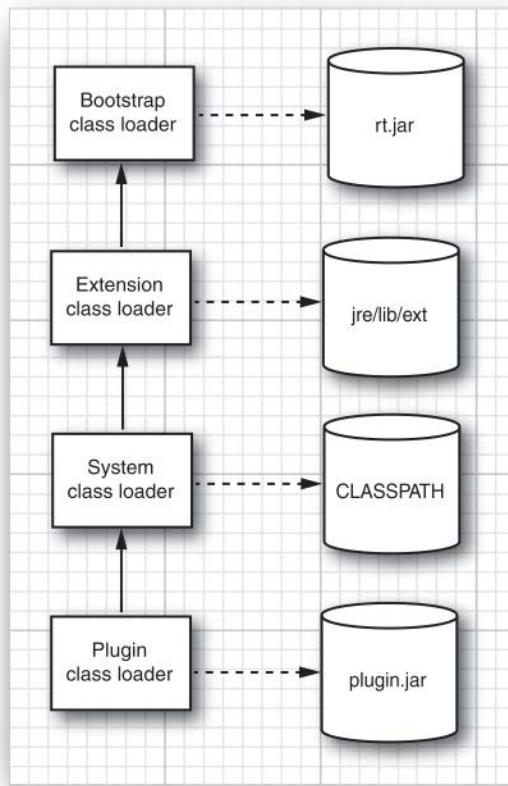


Figure 9.1 The class loader hierarchy

To overcome this problem, the helper method needs to use the correct class loader. It can require the class loader as a parameter. Alternatively, it can require that the correct class loader is set as the *context class loader* of the current thread. This strategy is used by many frameworks (such as the JAXP and JNDI frameworks that we discussed in Chapters 3 and 5).

Each thread has a reference to a class loader, called the context class loader. The main thread's context class loader is the system class loader. When a new thread is created, its context class loader is set to the creating thread's context class loader. Thus, if you don't do anything, all threads will have their context class loaders set to the system class loader.

However, you can set any class loader by calling

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

The helper method can then retrieve the context class loader:

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class cl = loader.loadClass(className);
```

The question remains when the context class loader is set to the plugin class loader. The application designer must make this decision. Generally, it is a good idea to set the context class loader when invoking a method of a plugin class that was loaded with a different class loader. Alternatively, the caller of the helper method can set the context class loader.



TIP: If you write a method that loads a class by name, it is a good idea to offer the caller the choice between passing an explicit class loader and using the context class loader. Don't simply use the class loader of the method's class.

9.1.3 Using Class Loaders as Namespaces

Every Java programmer knows that package names are used to eliminate name conflicts. There are two classes called `Date` in the standard library, but of course their real names are `java.util.Date` and `java.sql.Date`. The simple name is only a programmer convenience and requires the inclusion of appropriate `import` statements. In a running program, all class names contain their package names.

It might surprise you, however, that you can have two classes in the same virtual machine that have the same class *and* package name. A class is determined by its full name *and* the class loader. This technique is useful for loading code from multiple sources. For example, a browser uses separate instances of the applet class loader for each web page. This allows the virtual machine to separate classes from different web pages, no matter what they are named. Figure 9.2 shows an example. Suppose a web page contains two applets, provided by different advertisers, and each applet has a class called `Banner`. Since each applet is loaded by a separate class loader, these classes are entirely distinct and do not conflict with each other.

NOTE: This technique has other uses as well, such as “hot deployment” of servlets and Enterprise JavaBeans. See <http://zeroturnaround.com/labs/rjc301> for more information.

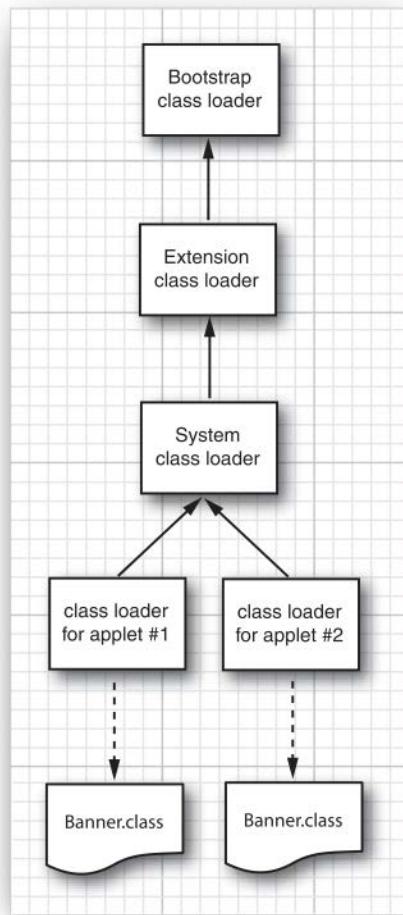


Figure 9.2 Two class loaders load different classes with the same name.

9.1.4 Writing Your Own Class Loader

You can write your own class loader for specialized purposes. That lets you carry out custom checks before you pass the bytecodes to the virtual machine. For example, your class loader can refuse to load a class that has not been marked as “paid for.”

To write your own class loader, simply extend the `ClassLoader` class and override the method

```
findClass(String className)
```

The `loadClass` method of the `ClassLoader` superclass takes care of the delegation to the parent and calls `findClass` only if the class hasn't already been loaded and if the parent class loader was unable to load the class.

Your implementation of this method must do the following:

1. Load the bytecodes for the class from the local file system or some other source.
2. Call the `defineClass` method of the `ClassLoader` superclass to present the bytecodes to the virtual machine.

In the program of Listing 9.1, we implement a class loader that loads encrypted class files. The program asks the user for the name of the first class to load (that is, the class containing `main`) and the decryption key. It then uses a special class loader to load the specified class and calls the `main` method. The class loader decrypts the specified class and all nonsystem classes that are referenced by it. Finally, the program calls the `main` method of the loaded class (see Figure 9.3).

For simplicity, we ignore the 2,000 years of progress in the field of cryptography and use the venerable Caesar cipher for encrypting the class files.

NOTE: David Kahn's wonderful book *The Codebreakers* (Macmillan, 1967, p. 84) refers to Suetonius as a historical source for the Caesar cipher. Caesar shifted the 24 letters of the Roman alphabet by 3 letters, which at the time baffled his adversaries.

When this chapter was first written, the U.S. government restricted the export of strong encryption methods. Therefore, we used Caesar's method for our example because it was clearly legal for export.

Our version of the Caesar cipher has as a key a number between 1 and 255. To decrypt, simply add that key to every byte and reduce modulo 256. The `Caesar.java` program of Listing 9.2 carries out the encryption.

To not confuse the regular class loader, we use a different extension, `.caesar`, for the encrypted class files.

To decrypt, the class loader simply subtracts the key from every byte. In the companion code for this book, you will find four class files, encrypted with a key value of 3—the traditional choice. To run the encrypted program, you'll need the custom class loader defined in our `ClassLoaderTest` program.

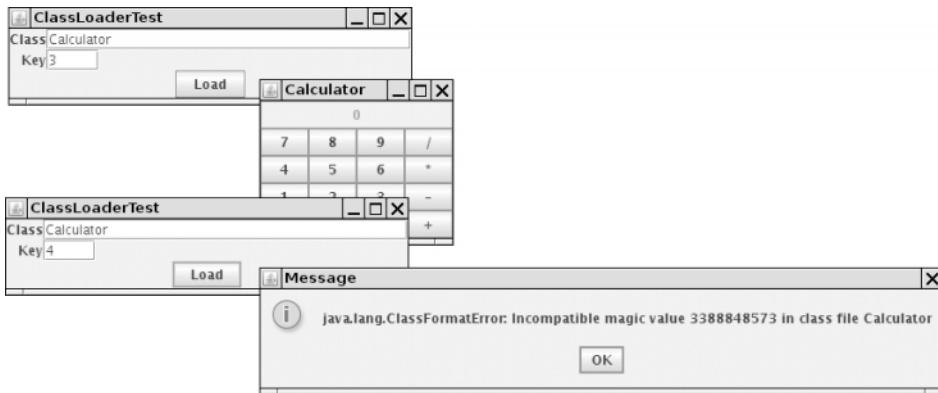


Figure 9.3 The ClassLoaderTest program

Encrypting class files has a number of practical uses (provided, of course, that you use something stronger than the Caesar cipher). Without the decryption key, the class files are useless. They can neither be executed by a standard virtual machine nor readily disassembled.

This means that you can use a custom class loader to authenticate the user of the class or to ensure that a program has been paid for before it will be allowed to run. Of course, encryption is only one application of a custom class loader. You can use other types of class loaders to solve other problems—for example, storing class files in a database.

Listing 9.1 classLoader/ClassLoaderTest.java

```
1 package classLoader;
2
3 import java.io.*;
4 import java.lang.reflect.*;
5 import java.nio.file.*;
6 import java.awt.*;
7 import java.awt.event.*;
8 import javax.swing.*;
9
10 /**
11  * This program demonstrates a custom class loader that decrypts class files.
12  * @version 1.24 2016-05-10
13  * @author Cay Horstmann
14  */
15 public class ClassLoaderTest
16 {
```

(Continues)

Listing 9.1 (Continued)

```
17  public static void main(String[] args)
18  {
19      EventQueue.invokeLater(() ->
20      {
21          JFrame frame = new ClassLoaderFrame();
22          frame.setTitle("ClassLoaderTest");
23          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24          frame.setVisible(true);
25      });
26  }
27 }
28 /**
29 * This frame contains two text fields for the name of the class to load and the decryption key.
30 */
31 class ClassLoaderFrame extends JFrame
32 {
33     {
34         private JTextField keyField = new JTextField("3", 4);
35         private JTextField nameField = new JTextField("Calculator", 30);
36         private static final int DEFAULT_WIDTH = 300;
37         private static final int DEFAULT_HEIGHT = 200;
38
39     public ClassLoaderFrame()
40     {
41         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
42         setLayout(new GridBagLayout());
43         add(new JLabel("Class"), new GBC(0, 0).setAnchor(GBC.EAST));
44         add(nameField, new GBC(1, 0).setWeight(100, 0).setAnchor(GBC.WEST));
45         add(new JLabel("Key"), new GBC(0, 1).setAnchor(GBC.EAST));
46         add(keyField, new GBC(1, 1).setWeight(100, 0).setAnchor(GBC.WEST));
47         JButton loadButton = new JButton("Load");
48         add(loadButton, new GBC(0, 2, 2, 1));
49         loadButton.addActionListener(event -> runClass(nameField.getText(), keyField.getText()));
50         pack();
51     }
52
53 /**
54 * Runs the main method of a given class.
55 * @param name the class name
56 * @param key the decryption key for the class files
57 */
58 public void runClass(String name, String key)
59 {
60     try
61     {
62         ClassLoader loader = new CryptoClassLoader(Integer.parseInt(key));
```

```
63         Class<?> c = loader.loadClass(name);
64         Method m = c.getMethod("main", String[].class);
65         m.invoke(null, (Object) new String[] {});
66     }
67     catch (Throwable e)
68     {
69         JOptionPane.showMessageDialog(this, e);
70     }
71 }
72
73 /**
74 * This class loader loads encrypted class files.
75 */
76 class CryptoClassLoader extends ClassLoader
77 {
78     private int key;
79
80     /**
81      * Constructs a crypto class loader.
82      * @param k the decryption key
83      */
84     public CryptoClassLoader(int k)
85     {
86         key = k;
87     }
88
89
90     protected Class<?> findClass(String name) throws ClassNotFoundException
91     {
92         try
93         {
94             byte[] classBytes = null;
95             classBytes = loadClassBytes(name);
96             Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
97             if (cl == null) throw new ClassNotFoundException(name);
98             return cl;
99         }
100        catch (IOException e)
101        {
102            throw new ClassNotFoundException(name);
103        }
104    }
105
106 /**
107 * Loads and decrypt the class file bytes.
108 * @param name the class name
109 * @return an array with the class file bytes
110 */
111 }
```

(Continues)

Listing 9.1 (*Continued*)

```
112     private byte[] loadClassBytes(String name) throws IOException
113     {
114         String cname = name.replace('.', '/') + ".caesar";
115         byte[] bytes = Files.readAllBytes(Paths.get(cname));
116         for (int i = 0; i < bytes.length; i++)
117             bytes[i] = (byte) (bytes[i] - key);
118         return bytes;
119     }
120 }
```

Listing 9.2 classLoader/Caesar.java

```
1 package classLoader;
2
3 import java.io.*;
4
5 /**
6  * Encrypts a file using the Caesar cipher.
7  * @version 1.01 2012-06-10
8  * @author Cay Horstmann
9 */
10 public class Caesar
11 {
12     public static void main(String[] args) throws Exception
13     {
14         if (args.length != 3)
15         {
16             System.out.println("USAGE: java classLoader.Caesar in out key");
17             return;
18         }
19
20         try(FileInputStream in = new FileInputStream(args[0]);
21             FileOutputStream out = new FileOutputStream(args[1]))
22         {
23             int key = Integer.parseInt(args[2]);
24             int ch;
25             while ((ch = in.read()) != -1)
26             {
27                 byte c = (byte) (ch + key);
28                 out.write(c);
29             }
30         }
31     }
32 }
```

java.lang.Class 1.0

- `ClassLoader getClassLoader()` 1.0
gets the class loader that loaded this class.

java.lang.ClassLoader 1.0

- `ClassLoader getParent() 1.2`
returns the parent class loader, or `null` if the parent class loader is the bootstrap class loader.
- `static ClassLoader getSystemClassLoader() 1.2`
gets the system class loader—that is, the class loader that was used to load the first application class.
- `protected Class findClass(String name) 1.2`
should be overridden by a class loader to find the bytecodes for a class and present them to the virtual machine by calling the `defineClass` method. In the name of the class, use `.` as package name separator, and don't use a `.class` suffix.
- `Class defineClass(String name, byte[] bytecodeData, int offset, int length)`
adds a new class to the virtual machine whose bytecodes are provided in the given data range.

java.net.URLClassLoader 1.2

- `URLClassLoader(URL[] urls)`
- `URLClassLoader(URL[] urls, ClassLoader parent)`
constructs a class loader that loads classes from the given URLs. If a URL ends in a `/`, it is assumed to be a directory, otherwise it is assumed to be a JAR file.

java.lang.Thread 1.0

- `ClassLoader getContextClassLoader() 1.2`
gets the class loader that the creator of this thread has designated as the most reasonable class loader to use when executing this thread.
- `void setContextClassLoader(ClassLoader loader) 1.2`
sets a class loader for code in this thread to retrieve for loading classes. If no context class loader is set explicitly when a thread is started, the parent's context class loader is used.

9.1.5 Bytecode Verification

When a class loader presents the bytecodes of a newly loaded Java platform class to the virtual machine, these bytecodes are first inspected by a *verifier*. The verifier checks that the instructions cannot perform actions that are obviously damaging. All classes except for system classes are verified.

Here are some of the checks that the verifier carries out:

- Variables are initialized before they are used.
- Method calls match the types of object references.
- Rules for accessing private data and methods are not violated.
- Local variable accesses fall within the runtime stack.
- The runtime stack does not overflow.

If any of these checks fails, the class is considered corrupted and will not be loaded.

NOTE: If you are familiar with Gödel's theorem, you might wonder how the verifier can prove that a class file is free from type mismatches, uninitialized variables, and stack overflows. Gödel's theorem states that it is impossible to design algorithms that process program files and decide whether the input programs have a particular property (such as being free from stack overflows). Is this a conflict between the public relations department at Oracle and the laws of logic? No—in fact, the verifier is *not* a decision algorithm in the sense of Gödel. If the verifier accepts a program, it is indeed safe. However, the verifier might reject virtual machine instructions even though they would actually be safe. (You might have run into this issue when you were forced to initialize a variable with a dummy value because the verifier couldn't see that it was going to be properly initialized.)

This strict verification is an important security consideration. Accidental errors, such as uninitialized variables, can easily wreak havoc if they are not caught. More importantly, in the wide open world of the Internet, you must be protected against malicious programmers who create evil effects on purpose. For example, by modifying values on the runtime stack or by writing to the private data fields of system objects, a program can break through the security system of a browser.

You might wonder, however, why a special verifier checks all these features. After all, the compiler would never allow you to generate a class file in which an uninitialized variable is used or in which a private data field is accessed from another class. Indeed, a class file generated by a compiler for the Java programming language always passes verification. However, the bytecode format used

in the class files is well documented, and it is an easy matter for someone with experience in assembly programming and a hex editor to manually produce a class file containing valid but unsafe instructions for the Java virtual machine. The verifier is always guarding against maliciously altered class files, not just checking the class files produced by a compiler.

Here's an example of how to construct such an altered class file. We start with the program `VerifierTest.java` of Listing 9.3. This is a simple program that calls a method and displays the method's result. The program can be run both as a console program and as an applet. The `fun` method itself just computes $1 + 2$.

```
static int fun()
{
    int m;
    int n;
    m = 1;
    n = 2;
    int r = m + n;
    return r;
}
```

As an experiment, try to compile the following modification of this program:

```
static int fun()
{
    int m = 1;
    int n;
    m = 1;
    m = 2;
    int r = m + n;
    return r;
}
```

In this case, `n` is not initialized, and it could have any random value. Of course, the compiler detects that problem and refuses to compile the program. To create a bad class file, we have to work a little harder. First, run the `javap` program to find out how the compiler translates the `fun` method. The command

```
javap -c verifier.VerifierTest
```

shows the bytecodes in the class file in mnemonic form.

```
Method int fun()
  0  iconst_1
  1  istore_0
  2  iconst_2
  3  istore_1
  4  iload_0
  5  iload_1
  6  iadd
```

```
7 istore_2
8 iload_2
9 ireturn
```

Use a hex editor to change instruction 3 from `istore_1` to `istore_0`. That is, local variable 0 (which is `m`) is initialized twice, and local variable 1 (which is `n`) is not initialized at all. We need to know the hexadecimal values for these instructions; these values are readily available from *The Java™ Virtual Machine Specification, Second Edition*, by Tim Lindholm and Frank Yellin (Prentice Hall, 1999).

```
0 iconst_1 04
1 istore_0 3B
2 iconst_2 05
3 istore_1 3C
4 iload_0 1A
5 iload_1 1B
6 iadd 60
7 istore_2 3D
8 iload_2 1C
9 ireturn AC
```

You can use any hex editor to carry out the modification. In Figure 9.4, you see the class file `VerifierTest.class` loaded into the Gnome hex editor, with the bytecodes of the `fun` method highlighted.

Change `3C` to `3B` and save the class file. Then try running the `VerifierTest` program. You get an error message:

```
Exception in thread "main" java.lang.VerifyError: (class: VerifierTest, method:fun signature: ()I) Accessing value from uninitialized register 1
```

That is good—the virtual machine detected our modification.

Now run the program with the `-noverify` (or `-Xverify:none`) option.

```
java -noverify verifier.VerifierTest
```

The `fun` method returns a seemingly random value. This is actually 2 plus the value that happened to be stored in the variable `n`, which never was initialized. Here is a typical printout:

```
1 + 2 == 15102330
```

To see how browsers handle verification, we wrote this program to run either as an application or an applet. Load the applet into a browser, using a file URL such as

```
file:///C:/CoreJavaBook/v2ch9/verifier/VerifierTest.html
```

You then see an error message displayed indicating that verification has failed (Figure 9.5).

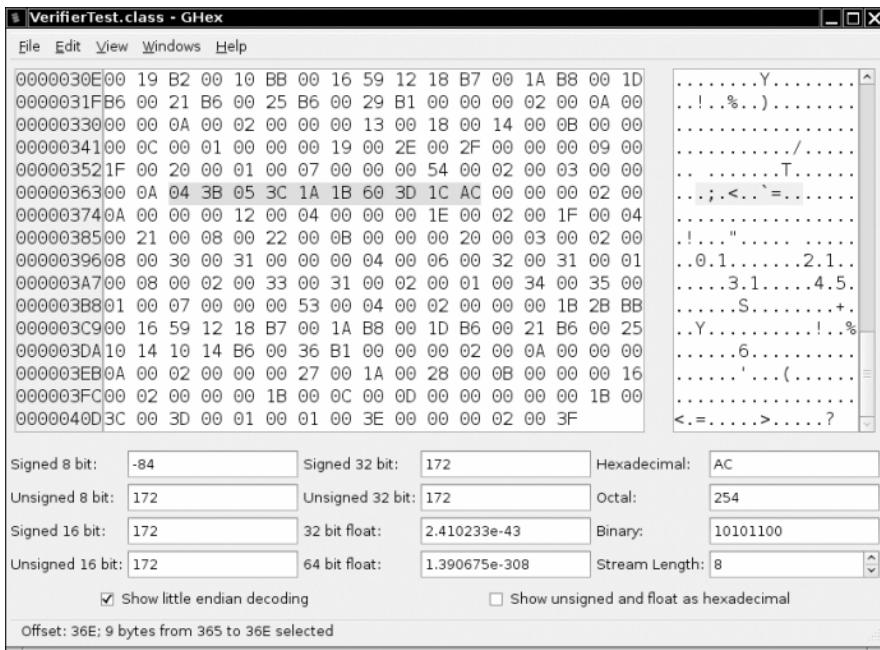


Figure 9.4 Modifying bytecodes with a hex editor

Listing 9.3 verifier/VerifierTest.java

```

1 package verifier;
2
3 import java.applet.*;
4 import java.awt.*;
5
6 /**
7 * This application demonstrates the bytecode verifier of the virtual machine. If you use a hex
8 * editor to modify the class file, then the virtual machine should detect the tampering.
9 * @version 1.00 1997-09-10
10 * @author Cay Horstmann
11 */
12 public class VerifierTest extends Applet
13 {
14     public static void main(String[] args)
15     {
16         System.out.println("1 + 2 == " + fun());
17     }
18 }
```

(Continues)

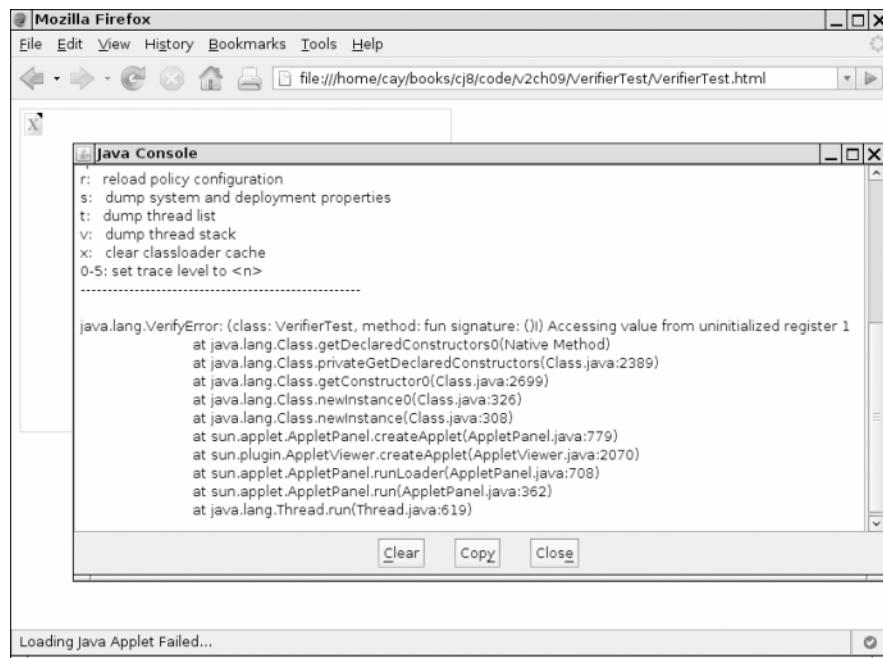


Figure 9.5 Loading a corrupted class file raises a method verification error.

Listing 9.3 (Continued)

```
19  /**
20  * A function that computes 1 + 2.
21  * @return 3, if the code has not been corrupted
22  */
23  public static int fun()
24  {
25      int m;
26      int n;
27      m = 1;
28      n = 2;
29      // use hex editor to change to "m = 2" in class file
30      int r = m + n;
31      return r;
32  }
33
34  public void paint(Graphics g)
35  {
36      g.drawString("1 + 2 == " + fun(), 20, 20);
37  }
38 }
```

9.2 Security Managers and Permissions

Once a class has been loaded into the virtual machine and checked by the verifier, the second security mechanism of the Java platform springs into action: the *security manager*. This is the topic of the following sections.

9.2.1 Permission Checking

The security manager controls whether a specific operation is permitted. Operations checked by the security manager include the following:

- Creating a new class loader
- Exiting the virtual machine
- Accessing a field of another class by using reflection
- Accessing a file
- Opening a socket connection
- Starting a print job
- Accessing the system clipboard
- Accessing the AWT event queue
- Bringing up a top-level window

There are many other checks throughout the Java library.

The default behavior when running Java applications is to install *no* security manager, so all these operations are permitted. The applet viewer, on the other hand, enforces a security policy that is quite restrictive.

For example, applets are not allowed to exit the virtual machine. If they try calling the `exit` method, a security exception is thrown. Here is what happens in detail. The `exit` method of the `Runtime` class calls the `checkExit` method of the security manager. Here is the entire code of the `exit` method:

```
public void exit(int status)
{
    SecurityManager security = System.getSecurityManager();
    if (security != null)
        security.checkExit(status);
    exitInternal(status);
}
```

The security manager now checks if the exit request came from the browser or an individual applet. If the security manager agrees with the exit request, the `checkExit` method simply returns and normal processing continues. However, if

the security manager doesn't want to grant the request, the `checkExit` method throws a `SecurityException`.

The `exit` method continues only if no exception occurred. It then calls the *private native* `exitInternal` method that actually terminates the virtual machine. There is no other way to terminate the virtual machine, and since the `exitInternal` method is private, it cannot be called from any other class. Thus, any code that attempts to exit the virtual machine must go through the `exit` method and thus through the `checkExit` security check without triggering a security exception.

Clearly, the integrity of the security policy depends on careful coding. The providers of system services in the standard library must always consult the security manager before attempting any sensitive operation.

The security manager of the Java platform allows both programmers and system administrators fine-grained control over individual security permissions. We will describe these features in the following section. First, we'll summarize the Java 2 platform security model. We'll then show how you can control permissions with *policy files*. Finally, we'll explain how you can define your own permission types.

NOTE: It is possible to implement and install your own security manager, but you should not attempt this unless you are an expert in computer security. It is much safer to configure the standard security manager.

9.2.2 Java Platform Security

JDK 1.0 had a very simple security model: Local classes had full permissions, and remote classes were confined to the *sandbox*. Just like a child that can only play in a sandbox, remote code was only allowed to paint on the screen and interact with the user. The applet security manager denied all access to local resources. JDK 1.1 implemented a slight modification: Remote code that was signed by a trusted entity was granted the same permissions as local classes. However, both versions of the JDK used an all-or-nothing approach. Programs either had full access or they had to play in the sandbox.

Starting with Java SE 1.2, the Java platform has a much more flexible mechanism. A *security policy* maps *code sources* to *permission sets* (see Figure 9.6).

A *code source* is specified by a *code base* and a set of *certificates*. The code base specifies the origin of the code. For example, the code base of remote applet code is the HTTP URL from which the applet was loaded. The code base of code in a JAR file is the file's URL. A certificate, if present, is an assurance by some party

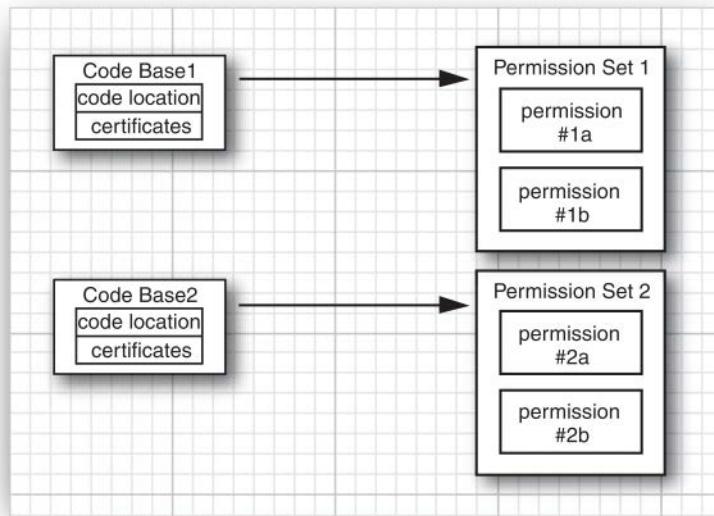


Figure 9.6 A security policy

that the code has not been tampered with. We'll cover certificates later in this chapter.

A *permission* is any property that is checked by a security manager. The Java platform supports a number of permission classes, each encapsulating the details of a particular permission. For example, the following instance of the `FilePermission` class states that it is okay to read and write any file in the `/tmp` directory:

```
FilePermission p = new FilePermission("/tmp/*", "read,write");
```

More importantly, the default implementation of the `Policy` class reads permissions from a *permission file*. Inside a permission file, the same read permission is expressed as

```
permission java.io.FilePermission "/tmp/*", "read,write";
```

We'll discuss permission files in the next section.

Figure 9.7 shows the hierarchy of the permission classes that were supplied with Java SE 1.2. Many more permission classes have been added in subsequent Java releases.

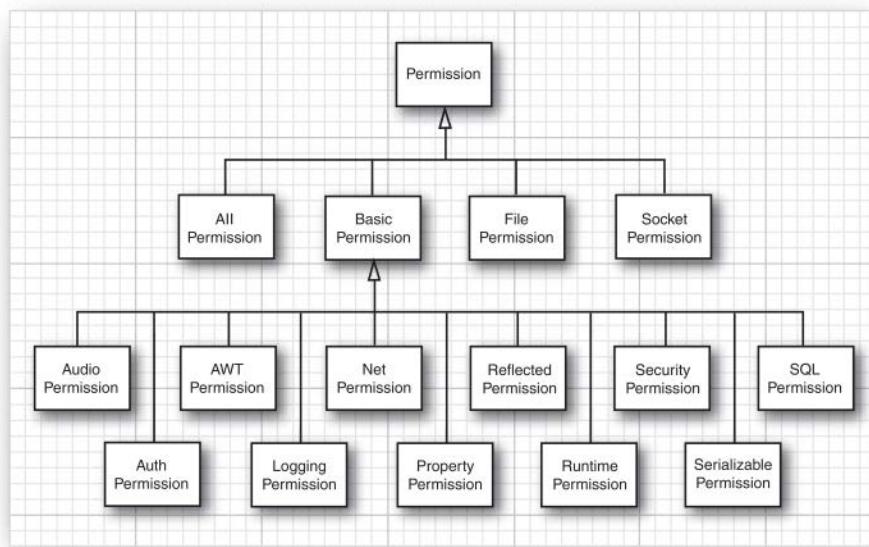


Figure 9.7 A part of the hierarchy of permission classes

In the preceding section, you saw that the `SecurityManager` class has security check methods such as `checkExit`. These methods exist only for the convenience of the programmer and for backward compatibility. They all map into standard permission checks. For example, here is the source code for the `checkExit` method:

```

public void checkExit()
{
    checkPermission(new RuntimePermission("exitVM"));
}
  
```

Each class has a *protection domain*—an object that encapsulates both the code source and the collection of permissions of the class. When the `SecurityManager` needs to check a permission, it looks at the classes of all methods currently on the call stack. It then gets the protection domains of all classes and asks each protection domain if its permission collection allows the operation currently being checked. If all domains agree, the check passes. Otherwise, a `SecurityException` is thrown.

Why do all methods on the call stack need to allow a particular operation? Let us work through an example. Suppose the `init` method of an applet wants to open a file. It might call

```
Reader in = new FileReader(name);
```

The `FileReader` constructor calls the `FileInputStream` constructor, which calls the `checkRead` method of the security manager, which finally calls `checkPermission` with a `FilePermission(name, "read")` object. Table 9.1 shows the call stack.

Table 9.1 Call Stack During Permission Checking

Class	Method	Code Source	Permissions
SecurityManager	checkPermission	null	AllPermission
SecurityManager	checkRead	null	AllPermission
FileInputStream	Constructor	null	AllPermission
FileReader	Constructor	null	AllPermission
Applet	init	Applet code source	Applet permissions
...			

The `FileInputStream` and `SecurityManager` classes are *system classes* for which `CodeSource` is `null` and the permissions consist of an instance of the `AllPermission` class, which allows all operations. Clearly, their permissions alone can't determine the outcome of the check. As you can see, the `checkPermission` method must take into account the restricted permissions of the applet class. By checking the entire call stack, the security mechanism ensures that one class can never ask another class to carry out a sensitive operation on its behalf.

NOTE: This brief discussion of permission checking explains the basic concepts. However, we omit a number of technical details here. With security, the devil lies in the details, and we encourage you to read the book by Li Gong for more information. For a more critical view of the Java platform's security model, see the book *Securing Java: Getting Down to Business with Mobile Code, Second Edition*, by Gary McGraw and Ed W. Felten (Wiley, 1999). You can find an online version of that book at www.securingjava.com.

java.lang.SecurityManager 1.0

- `void checkPermission(Permission p)` 1.2

checks whether this security manager grants the given permission. The method throws a `SecurityException` if the permission is not granted.

java.lang.Class 1.0

- `ProtectionDomain getProtectionDomain() 1.2`

gets the protection domain for this class, or `null` if this class was loaded without a protection domain.

java.security.ProtectionDomain 1.2

- `ProtectionDomain(CodeSource source, PermissionCollection permissions)`
constructs a protection domain with the given code source and permissions.
- `CodeSource getCodeSource()`
gets the code source of this protection domain.
- `boolean implies(Permission p)`
returns `true` if the given permission is allowed by this protection domain.

java.security.CodeSource 1.2

- `Certificate[] getCertificates()`
gets the certificate chain for class file signatures associated with this code source.
- `URL getLocation()`
gets the code base of class files associated with this code source.

9.2.3 Security Policy Files

The *policy manager* reads *policy files* that contain instructions for mapping code sources to permissions. Here is a typical policy file:

```
grant codeBase "http://www.horstmann.com/classes"
{
    permission java.io.FilePermission "/tmp/*", "read,write";
};
```

This file grants permission to read and write files in the `/tmp` directory to all code that was downloaded from `www.horstmann.com/classes`.

You can install policy files in standard locations. By default, there are two locations:

- The file `java.policy` in the Java platform's home directory
- The file `.java.policy` (notice the period at the beginning of the file name) in the user's home directory

NOTE: You can change the locations of these files in the `java.security` configuration file in the `jre/lib/security`. The defaults are specified as

```
policy.url.1=file:${java.home}/lib/security/java.policy  
policy.url.2=file:${user.home}/.java.policy
```

A system administrator can modify the `java.security` file and specify policy URLs that reside on another server and cannot be edited by users. There can be any number of policy URLs (with consecutive numbers) in the policy file. The permissions of all files are combined.

If you want to store policies outside the file system, you can implement a subclass of the `Policy` class that gathers the permissions. Then change the line

```
policy.provider=sun.security.provider.PolicyFile
```

in the `java.security` configuration file.

During testing, we don't like to constantly modify the standard policy files. Therefore, we prefer to explicitly name the policy file required for each application. Place the permissions into a separate file—say, `MyApp.policy`. To apply the policy, you have two choices. You can set a system property inside your applications' `main` method:

```
System.setProperty("java.security.policy", "MyApp.policy");
```

Alternatively, you can start the virtual machine as

```
java -Djava.security.policy=MyApp.policy MyApp
```

For applets, you should instead use

```
appletviewer -J-Djava.security.policy=MyApplet.policy MyApplet.html
```

(You can use the `-J` option of the `appletviewer` to pass any command-line argument to the virtual machine.)

In these examples, the `MyApp.policy` file is added to the other policies in effect. If you add a second equal sign, such as

```
java -Djava.security.policy==MyApp.policy MyApp
```

then your application will use *only* the specified policy file, and the standard policy files will be ignored.



CAUTION: An easy mistake during testing is to accidentally leave a `.java.policy` file that grants a lot of permissions, perhaps even `AllPermission`, in the home directory. If you find that your application doesn't seem to pay attention to the restrictions in your policy file, check for a left-behind `.java.policy` file in your home directory. If you use a UNIX system, this is a particularly easy mistake to make because files with names that start with a period are not displayed by default.

As you saw previously, Java applications by default do not install a security manager. Therefore, you won't see the effect of policy files until you install one. You can, of course, add a line

```
System.setSecurityManager(new SecurityManager());
```

into your `main` method. Or you can add the command-line option `-Djava.security.manager` when starting the virtual machine.

```
java -Djava.security.manager -Djava.security.policy=MyApp.policy MyApp
```

In the remainder of this section, we'll show you in detail how to describe permissions in the policy file. We'll describe the entire policy file format, except for code certificates which we cover later in this chapter.

A policy file contains a sequence of `grant` entries. Each entry has the following form:

```
grant codesource
{
    permission1;
    permission2;
    ...
};
```

The code source contains a code base (which can be omitted if the entry applies to code from all sources) and the names of trusted principals and certificate signers (which can be omitted if signatures are not required for this entry).

The code base is specified as

```
codeBase "url"
```

If the URL ends in a /, it refers to a directory. Otherwise, it is taken to be the name of a JAR file. For example,

```
grant codeBase "www.horstmann.com/classes/" { . . . };
grant codeBase "www.horstmann.com/classes/MyApp.jar" { . . . };
```

The code base is a URL and should always contain forward slashes as file separators, even for file URLs in Windows. For example,

```
grant codeBase "file:C:/myapps/classes/" { . . . };
```

NOTE: Everyone knows that http URLs start with two slashes (http://). But there is enough confusion about file URLs, so the policy file reader accepts two forms of file URLs—namely, `file:///localFile` and `file:localFile`. Furthermore, a slash before a Windows drive letter is optional. That is, all of the following are acceptable:

```
file:C:/dir/filename.ext
file:/C:/dir/filename.ext
file://C:/dir/filename.ext
file:///C:/dir/filename.ext
```

Actually, in our tests, the `file:///C:/dir/filename.ext` is acceptable as well, and we have no explanation for that.

The permissions have the following structure:

```
permission className targetName, actionList;
```

The `className` is the fully qualified class name of the permission class (such as `java.io.FilePermission`). The `targetName` is a permission-specific value—for example, a file or directory name for the file permission, or a host and port for a socket permission. The `actionList` is also permission-specific. It is a list of actions, such as `read` or `connect`, separated by commas. Some permission classes don't need target names and action lists. Table 9.2 lists the commonly used permission classes and their actions.

Table 9.2 Permissions and Their Associated Targets and Actions

Permission	Target	Action
<code>java.io.FilePermission</code>	File target (see text)	<code>read, write, execute, delete</code>
<code>java.net.SocketPermission</code>	Socket target (see text)	<code>accept, connect, listen, resolve</code>
<code>java.util.PropertyPermission</code>	Property target (see text)	<code>read, write</code>

(Continues)

Table 9.2 (*Continued*)

Permission	Target	Action
java.lang.RuntimePermission	createClassLoader getClassLoader setContextClassLoader enableContextClassLoaderOverride createSecurityManager setSecurityManager exitVM getenv.variableName shutdownHooks setFactory setIO modifyThread stopThread modifyThreadGroup getProtectionDomain readFileDescriptor writeFileDescriptor loadLibrary.libraryName accessClassInPackage.packageName defineClassInPackage.packageName accessDeclaredMembers.className queuePrintJob getStackTrace setDefaultUncaughtExceptionHandler preferences usePolicy	None
java.awt.AWTPermission	showWindowWithoutWarningBanner accessClipboard accessEventQueue createRobot fullScreenExclusive listenToAllAWTEvents readDisplayPixels replaceKeyboardFocusManager watchMousePointer setWindowAlwaysOnTop setAppletStub	None

(Continues)

Table 9.2 (Continued)

Permission	Target	Action
java.net.NetPermission	setDefaultAuthenticator specifyStreamHandler requestPasswordAuthentication setProxySelector getProxySelector setCookieHandler getCookieHandler setResponseCache getResponseCache	None
java.lang.reflect.ReflectPermission	suppressAccessChecks	None
java.io.SerializablePermission	enableSubclassImplementation enableSubstitution	None
java.security.SecurityPermission	createAccessControlContext getDomainCombiner getPolicy setPolicy getProperty.keyName setProperty.keyName insertProvider.providerName removeProvider.providerName setSystemScope setIdentityPublicKey setIdentityInfo addIdentityCertificate removeIdentityCertificate printIdentity clearProviderProperties.providerName putProviderProperty.providerName removeProviderProperty.providerName getSignerPrivateKey setSignerKeyPair	None
java.security.AllPermission	None	None
javax.audio.AudioPermission	Play record	None

(Continues)

Table 9.2 (Continued)

Permission	Target	Action
javax.security.auth.AuthPermission	doAs doAsPrivileged getSubject getSubjectFromDomainCombiner setReadOnly modifyPrincipals modifyPublicCredentials modifyPrivateCredentials refreshCredential destroyCredential createLoginContext.contextName getLoginConfiguration setLoginConfiguration refreshLoginConfiguration	None
java.util.logging.LoggingPermission	control	None
java.sql.SQLPermission	setLog	None

As you can see from Table 9.2, most permissions simply permit a particular operation. You can think of the operation as the target with an implied action "permit". These permission classes all extend the `BasicPermission` class (see Figure 9.7 on p. 512). However, the targets for the file, socket, and property permissions are more complex, and we need to investigate them in detail.

File permission targets can have the following form:

<i>file</i>	A file
<i>directory/</i>	A directory
<i>directory/*</i>	All files in the directory
*	All files in the current directory
<i>directory/-</i>	All files in the directory or one of its subdirectories
-	All files in the current directory or one of its subdirectories
<<ALL FILES>>	All files in the file system

For example, the following permission entry gives access to all files in the directory `/myapp` and any of its subdirectories:

```
permission java.io.FilePermission "/myapp/-", "read,write,delete";
```

You must use the `\` escape sequence to denote a backslash in a Windows file name.

```
permission java.io.FilePermission "c:\\myapp\\-", "read,write,delete";
```

Socket permission targets consist of a host and a port range. Host specifications have the following form:

<i>hostname or IPAddress</i>	A single host
<i>localhost</i> or the empty string	The local host
<i>*.domainSuffix</i>	Any host whose domain ends with the given suffix
*	All hosts

Port ranges are optional and have the form:

<i>:n</i>	A single port
<i>:n-</i>	All ports numbered <i>n</i> and above
<i>:-n</i>	All ports numbered <i>n</i> and below
<i>:n1-n2</i>	All ports in the given range

Here is an example:

```
permission java.net.SocketPermission "*.*.horstmann.com:8000-8999", "connect";
```

Finally, property permission targets can have one of two forms:

<i>property</i>	A specific property
<i>propertyPrefix.*</i>	All properties with the given prefix

Examples are "java.home" and "java.vm.*".

For example, the following permission entry allows a program to read all properties that start with `java.vm`:

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

You can use system properties in policy files. The token `${property}` is replaced by the property value. For example, `${user.home}` is replaced by the home directory of the user. Here is a typical use of this system property in a permission entry:

```
permission java.io.FilePermission "${user.home}", "read,write";
```

To create platform-independent policy files, it is a good idea to use the `file.separator` property instead of explicit / or \\ separators. To make this simpler, the special notation `${/}` is a shortcut for `${file.separator}`. For example,

```
permission java.io.FilePermission "${user.home}${/}-", "read,write";
```

is a portable entry for granting permission to read and write in the user's home directory and any of its subdirectories.

NOTE: The JDK comes with a rudimentary tool, called `policytool`, that you can use to edit policy files (see Figure 9.8). Of course, this tool is not suitable for end users who would be completely mystified by most of the settings. We view it as a proof of concept for an administration tool that might be used by system administrators who prefer point-and-click over syntax. Still, what's missing in it is a sensible set of categories (such as low, medium, or high security) that would be meaningful to nonexperts. As a general observation, we believe that the Java platform certainly contains all the pieces for a fine-grained security model but it could benefit from some polish in delivering these pieces to end users and system administrators.

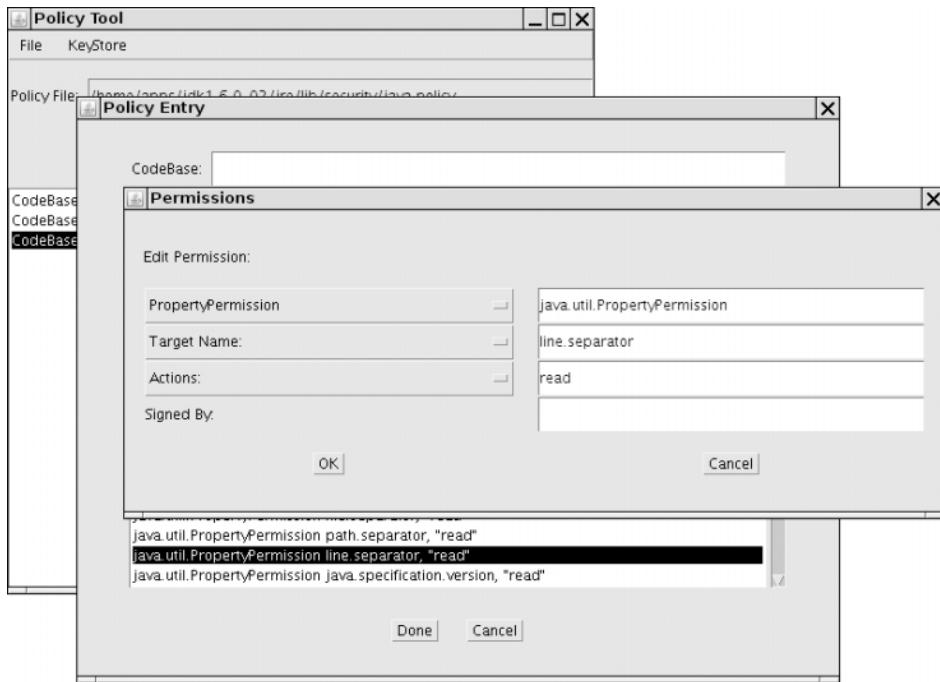


Figure 9.8 The policy tool

9.2.4 Custom Permissions

In this section, you'll see how you can supply your own permission class that users can refer to in their policy files.

To implement your permission class, extend the `Permission` class and supply the following methods:

- A constructor with two `String` parameters, for the target and the action list
- `String getActions()`
- `boolean equals(Object other)`
- `int hashCode()`
- `boolean implies(Permission other)`

The last method is the most important. Permissions have an *ordering*, in which more general permissions *imply* more specific ones. Consider the file permission

```
p1 = new FilePermission("/tmp/-", "read, write");
```

This permission allows reading and writing of any file in the `/tmp` directory and any of its subdirectories.

This permission implies other, more specific permissions:

```
p2 = new FilePermission("/tmp/-", "read");
p3 = new FilePermission("/tmp/aFile", "read, write");
p4 = new FilePermission("/tmp/aDirectory/-", "write");
```

In other words, a file permission `p1` implies another file permission `p2` if

1. The target file set of `p1` contains the target file set of `p2`.
2. The action set of `p1` contains the action set of `p2`.

Consider the following example of the use of the `implies` method. When the `FileInputStream` constructor wants to open a file for reading, it checks whether it has permission to do so. For that check, a *specific* file permission object is passed to the `checkPermission` method:

```
checkPermission(new FilePermission(fileName, "read"));
```

The security manager now asks all applicable permissions whether they imply this permission. If any one of them implies it, the check passes.

In particular, the `AllPermission` implies all other permissions.

If you define your own permission classes, you need to define a suitable notion of implication for your permission objects. Suppose, for example, that you define a `TVPermission` for a set-top box powered by Java technology. A permission

```
new TVPermission("Tommy:2-12:1900-2200", "watch, record")
```

might allow Tommy to watch and record television channels 2–12 between 19:00 and 22:00. You need to implement the `implies` method so that this permission implies a more specific one, such as

```
new TVPermission("Tommy:4:2000-2100", "watch")
```

9.2.5 Implementation of a Permission Class

In the next sample program, we implement a new permission for monitoring the insertion of text into a text area. The program ensures that you cannot add bad words such as *sex*, *drugs*, and *C++* into a text area. We use a custom permission class so that the list of bad words can be supplied in a policy file.

The following subclass of `JTextArea` asks the security manager whether it is okay to add new text:

```
class WordCheckTextArea extends JTextArea
{
    public void append(String text)
    {
        WordCheckPermission p = new WordCheckPermission(text, "insert");
        SecurityManager manager = System.getSecurityManager();
        if (manager != null) manager.checkPermission(p);
        super.append(text);
    }
}
```

If the security manager grants the `WordCheckPermission`, the text is appended. Otherwise, the `checkPermission` method throws an exception.

Word check permissions have two possible actions: `insert` (the permission to insert a specific text) and `avoid` (the permission to add any text that avoids certain bad words). You should run this program with the following policy file:

```
grant
{
    permission permissions.WordCheckPermission "sex,drugs,C++", "avoid";
};
```

This policy file grants the permission to insert any text that avoids the bad words *sex*, *drugs*, and *C++*.

When designing the `WordCheckPermission` class, we must pay particular attention to the `implies` method. Here are the rules that control whether permission `p1` implies permission `p2`:

- If `p1` has action `avoid` and `p2` has action `insert`, then the target of `p2` must avoid all words in `p1`. For example, the permission

```
permissions.WordCheckPermission "sex,drugs,C++", "avoid"
```

implies the permission

```
permissions.WordCheckPermission "Mary had a little lamb", "insert"
```

- If p_1 and p_2 both have action `avoid`, then the word set of p_2 must contain all words in the word set of p_1 . For example, the permission

```
permissions.WordCheckPermission "sex,drugs", "avoid"
```

implies the permission

```
permissions.WordCheckPermission "sex,drugs,C++", "avoid"
```

- If p_1 and p_2 both have action `insert`, then the text of p_1 must contain the text of p_2 . For example, the permission

```
permissions.WordCheckPermission "Mary had a little lamb", "insert"
```

implies the permission

```
permissions.WordCheckPermission "a little lamb", "insert"
```

You can find the implementation of this class in Listing 9.4.

Note that to retrieve the permission target, you need to use the confusingly named `getName` method of the `Permission` class.

Since permissions are described by a pair of strings in policy files, permission classes need to be prepared to parse these strings. In particular, we use the following method to transform the comma-separated list of bad words of an `avoid` permission into a genuine `Set`:

```
public Set<String> badWordSet()
{
    Set<String> set = new HashSet<String>();
    set.addAll(Arrays.asList(getName().split(",")));
    return set;
}
```

This code allows us to use the `equals` and `containsAll` methods to compare sets. As you saw in Chapter 3, the `equals` method of a set class finds two sets to be equal if they contain the same elements in any order. For example, the sets resulting from "`sex,drugs,C++`" and "`C++,drugs,sex`" are equal.



CAUTION: Make sure that your permission class is a public class. The policy file loader cannot load classes with package visibility outside the boot class path, and it silently ignores any classes that it cannot find.

The program in Listing 9.5 shows how the `WordCheckPermission` class works. Type any text into the text field and click the Insert button. If the security check passes, the text is appended to the text area. If not, an error message is displayed (see Figure 9.9).



Figure 9.9 The PermissionTest program



CAUTION: If you carefully look at Figure 9.9, you will see that the message window has a warning triangle, which is supposed to warn viewers that this window may have been popped up without permission. The warning started out as an ominous “Untrusted Java Applet Window” label, got watered down several times in successive JDK releases, and has now become essentially useless for alerting users. The warning is turned off by the `showWindowWithoutWarningBanner` target of the `java.awt.AWTPermission`. If you like, you can edit the policy file to grant that permission.

You have now seen how to configure Java platform security. Most commonly, you will simply tweak the standard permissions. For additional control, you can define custom permissions that can be configured in the same way as the standard permissions.

Listing 9.4 permissions/WordCheckPermission.java

```
1 package permissions;
2
3 import java.security.*;
4 import java.util.*;
```

```
6  /**
7   * A permission that checks for bad words.
8   */
9  public class WordCheckPermission extends Permission
10 {
11     private String action;
12
13     /**
14      * Constructs a word check permission.
15      * @param target a comma separated word list
16      * @param anAction "insert" or "avoid"
17      */
18     public WordCheckPermission(String target, String anAction)
19     {
20         super(target);
21         action = anAction;
22     }
23
24     public String getActions()
25     {
26         return action;
27     }
28
29     public boolean equals(Object other)
30     {
31         if (other == null) return false;
32         if (!getClass().equals(other.getClass())) return false;
33         WordCheckPermission b = (WordCheckPermission) other;
34         if (!Objects.equals(action, b.action)) return false;
35         if ("insert".equals(action)) return Objects.equals(getName(), b.getName());
36         else if ("avoid".equals(action)) return badWordSet().equals(b.badWordSet());
37         else return false;
38     }
39
40     public int hashCode()
41     {
42         return Objects.hash(getName(), action);
43     }
44
45     public boolean implies(Permission other)
46     {
47         if (!(other instanceof WordCheckPermission)) return false;
48         WordCheckPermission b = (WordCheckPermission) other;
49         if (action.equals("insert"))
50         {
51             return b.action.equals("insert") && getName().indexOf(b.getName()) >= 0;
52         }
```

(Continues)

Listing 9.4 (Continued)

```
53     else if (action.equals("avoid"))
54     {
55         if (b.action.equals("avoid")) return b.badWordSet().containsAll(badWordSet());
56         else if (b.action.equals("insert"))
57         {
58             for (String badWord : badWordSet())
59                 if (b.getName().indexOf(badWord) >= 0) return false;
60             return true;
61         }
62         else return false;
63     }
64     else return false;
65 }
66 /**
67 * Gets the bad words that this permission rule describes.
68 * @return a set of the bad words
69 */
70 public Set<String> badWordSet()
71 {
72     Set<String> set = new HashSet<>();
73     set.addAll(Arrays.asList(getName().split(",")));
74     return set;
75 }
76 }
77 }
```

Listing 9.5 permissions/PermissionTest.java

```
1 package permissions;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8 * This class demonstrates the custom WordCheckPermission.
9 * @version 1.04 2016-05-10
10 * @author Cay Horstmann
11 */
12 public class PermissionTest
13 {
14     public static void main(String[] args)
15     {
16         System.setProperty("java.security.policy", "permissions/PermissionTest.policy");
17         System.setSecurityManager(new SecurityManager());
```

```
18     EventQueue.invokeLater(() ->
19     {
20         JFrame frame = new PermissionTestFrame();
21         frame.setTitle("PermissionTest");
22         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23         frame.setVisible(true);
24     });
25 }
26 }
27
28 /**
29 * This frame contains a text field for inserting words into a text area that is protected from
30 * "bad words".
31 */
32 class PermissionTestFrame extends JFrame
33 {
34     private JTextField textField;
35     private WordCheckTextArea textArea;
36     private static final int TEXT_ROWS = 20;
37     private static final int TEXT_COLUMNS = 60;
38
39     public PermissionTestFrame()
40     {
41         textField = new JTextField(20);
42         JPanel panel = new JPanel();
43         panel.add(textField);
44         JButton openButton = new JButton("Insert");
45         panel.add(openButton);
46         openButton.addActionListener(event -> insertWords(textField.getText()));
47
48         add(panel, BorderLayout.NORTH);
49
50         textArea = new WordCheckTextArea();
51         textArea.setRows(TEXT_ROWS);
52         textArea.setColumns(TEXT_COLUMNS);
53         add(new JScrollPane(textArea), BorderLayout.CENTER);
54         pack();
55     }
56
57 /**
58 * Tries to insert words into the text area. Displays a dialog if the attempt fails.
59 * @param words the words to insert
60 */
61 public void insertWords(String words)
62 {
63     try
64     {
65         textArea.append(words + "\n");
66     }
```

(Continues)

Listing 9.5 (Continued)

```
67     catch (SecurityException ex)
68     {
69         JOptionPane.showMessageDialog(this, "I am sorry, but I cannot do that.");
70         ex.printStackTrace();
71     }
72 }
73 */
74 /**
75 * A text area whose append method makes a security check to see that no bad words are added.
76 */
77 class WordCheckTextArea extends JTextArea
78 {
79     public void append(String text)
80     {
81         WordCheckPermission p = new WordCheckPermission(text, "insert");
82         SecurityManager manager = System.getSecurityManager();
83         if (manager != null) manager.checkPermission(p);
84         super.append(text);
85     }
86 }
87 }
```

java.security.Permission 1.2

- `Permission(String name)`
constructs a permission with the given target name.
- `String getName()`
returns the target name of this permission.
- `boolean implies(Permission other)`
checks whether this permission implies the other permission. That is the case if the other permission describes a more specific condition that is a consequence of the condition described by this permission.

9.3 User Authentication

The Java API provides a framework, called the Java Authentication and Authorization Service (JAAS), that integrates platform-provided authentication with permission management. We'll discuss the JAAS framework in the following sections.

9.3.1 The JAAS Framework

As you can tell from its name, the JAAS framework has two components. The “authentication” part is concerned with ascertaining the identity of a program user. The “authorization” part maps users to permissions.

JAAS is a “pluggable” API that isolates Java applications from the particular technology used to implement authentication. It supports, among others, UNIX logins, NT logins, Kerberos authentication, and certificate-based authentication.

Once a user has been authenticated, you can attach a set of permissions. For example, here we grant Harry a particular set of permissions that other users do not have:

```
grant principal com.sun.security.auth.UnixPrincipal "harry"
{
    permission java.util.PropertyPermission "user.*", "read";
    ...
};
```

The `com.sun.security.auth.UnixPrincipal` class checks the name of the UNIX user who is running this program. Its `getName` method returns the UNIX login name, and we check whether that name equals "harry".

Use a `LoginContext` to allow the security manager to check such a grant statement. Here is the basic outline of the login code:

```
try
{
    System.setSecurityManager(new SecurityManager());
    LoginContext context = new LoginContext("Login1"); // defined in JAAS configuration file
    context.login();
    // get the authenticated Subject
    Subject subject = context.getSubject();
    ...
    context.logout();
}
catch (LoginException exception) // thrown if login was not successful
{
    exception.printStackTrace();
}
```

Now the `subject` denotes the individual who has been authenticated.

The string parameter "Login1" in the `LoginContext` constructor refers to an entry with the same name in the JAAS configuration file. Here is a sample configuration file:

```
>Login1
{
    com.sun.security.auth.module.UxLoginModule required;
    com.whizzbang.auth.module.RetinaScanModule sufficient;
};

>Login2
{
    ...
};
```

Of course, the JDK contains no biometric login modules. The following modules are supplied in the `com.sun.security.auth.module` package:

```
UnixLoginModule
NTLoginModule
Krb5LoginModule
JndiLoginModule
KeyStoreLoginModule
```

A login policy consists of a sequence of login modules, each of which is labeled `required`, `sufficient`, `requisite`, or `optional`. The meaning of these keywords is given by the following algorithm.

A login authenticates a *subject*, which can have multiple *principals*. A principal describes some property of the subject, such as the user name, group ID, or role. As you saw in the `grant` statement, principals govern permissions. The `com.sun.security.auth.UnixPrincipal` describes the UNIX login name, and the `UnixNumericGroupPrincipal` can test for membership in a UNIX group.

A `grant` clause can test for a principal, with the syntax

```
grant principalClass "principalName"
```

For example:

```
grant com.sun.security.auth.UnixPrincipal "harry"
```

When a user has logged in, you then run, in a separate access control context, the code that requires checking of principals. Use the static `doAs` or `doAsPrivileged` method to start a new `PrivilegedAction` whose `run` method executes the code.

Both of those methods execute an action by calling the `run` method of an object that implements the `PrivilegedAction` interface, using the permissions of the subject's principals:

```
PrivilegedAction<T> action = () ->
{
    // run with permissions of subject principals
    ...
}
```

```
};  
T result = Subject.doAs(subject, action); // or Subject.doAsPrivileged(subject, action, null)
```

If the actions can throw checked exceptions, you need to implement the `PrivilegedExceptionAction` interface instead.

The difference between the `doAs` and `doAsPrivileged` methods is subtle. The `doAs` method starts out with the current access control context, whereas the `doAsPrivileged` method starts out with a new context. The latter method allows you to separate the permissions for the login code and the “business logic.” In our example application, the login code has permissions

```
permission javax.security.auth.AuthPermission "createLoginContext.Login1";  
permission javax.security.auth.AuthPermission "doAsPrivileged";
```

The authenticated user has a permission

```
permission java.util.PropertyPermission "user.*", "read";
```

If we had used `doAs` instead of `doAsPrivileged`, then the login code would have also needed that permission!

The program in Listings 9.6 and 9.7 demonstrates how to restrict permissions to certain users. The `AuthTest` program authenticates a user and runs a simple action that retrieves a system property.

To make this example work, package the code for the login and the action into two separate JAR files:

```
javac auth/*.java  
jar cvf login.jar auth/AuthTest.class  
jar cvf action.jar auth/SysPropAction.class
```

If you look at the policy file in Listing 9.8, you will see that the UNIX user with the name `harry` has the permission to read all files. Change `harry` to your login name. Then run the command

```
java -classpath login.jar;action.jar  
-Djava.security.policy=auth/AuthTest.policy  
-Djava.security.auth.login.config=auth/jaas.config  
auth.AuthTest
```

Listing 9.9 shows the login configuration.

On Windows, change `UnixPrincipal` to `NTUserPrincipal` in `AuthTest.policy` and `UnixLoginModule` to `NTLoginModule` in `jaas.config`. When running the program, use a semicolon to separate the JAR files:

```
java -classpath login.jar;action.jar . . .
```

The AuthTest program should now display the value of the `user.home` property. However, if you log in with a different name, a security exception should be thrown because you no longer have the required permission.



CAUTION: Be careful to follow these instructions *exactly*. It is very easy to get the setup wrong by making seemingly innocuous changes.

Listing 9.6 auth/AuthTest.java

```
1 package auth;
2
3 import java.security.*;
4 import javax.security.auth.*;
5 import javax.security.auth.login.*;
6
7 /**
8 * This program authenticates a user via a custom login and then executes the SysPropAction with
9 * the user's privileges.
10 * @version 1.01 2007-10-06
11 * @author Cay Horstmann
12 */
13 public class AuthTest
14 {
15     public static void main(final String[] args)
16     {
17         System.setSecurityManager(new SecurityManager());
18         try
19         {
20             LoginContext context = new LoginContext("Login1");
21             context.login();
22             System.out.println("Authentication successful.");
23             Subject subject = context.getSubject();
24             System.out.println("subject=" + subject);
25             PrivilegedAction<String> action = new SysPropAction("user.home");
26             String result = Subject.doAsPrivileged(subject, action, null);
27             System.out.println(result);
28             context.logout();
29         }
30         catch (LoginException e)
31         {
32             e.printStackTrace();
33         }
34     }
35 }
```

Listing 9.7 auth/SysPropAction.java

```
1 package auth;
2
3 import java.security.*;
4
5 /**
6     This action looks up a system property.
7     * @version 1.01 2007-10-06
8     * @author Cay Horstmann
9 */
10 public class SysPropAction implements PrivilegedAction<String>
11 {
12     private String propertyName;
13
14     /**
15         Constructs an action for looking up a given property.
16         @param propertyName the property name (such as "user.home")
17     */
18     public SysPropAction(String propertyName) { this.propertyName = propertyName; }
19
20     public String run()
21     {
22         return System.getProperty(propertyName);
23     }
24 }
```

Listing 9.8 auth/AuthTest.policy

```
1 grant codebase "file:login.jar"
2 {
3     permission javax.security.auth.AuthPermission "createLoginContext.Login1";
4     permission javax.security.auth.AuthPermission "doAsPrivileged";
5 };
6
7 grant principal com.sun.security.auth.UnixPrincipal "harry"
8 {
9     permission java.util.PropertyPermission "user.*", "read";
10};
```

Listing 9.9 auth/jaas.config

```
1 Login1
2 {
3     com.sun.security.auth.module.UnixLoginModule required;
4 };
```

javax.security.auth.login.LoginContext 1.4

- `LoginContext(String name)`

constructs a login context. The `name` corresponds to the login descriptor in the JAAS configuration file.

- `void login()`

establishes a login or throws `LoginException` if the login failed. Invokes the `login` method on the managers in the JAAS configuration file.

- `void logout()`

logs out the subject. Invokes the `logout` method on the managers in the JAAS configuration file.

- `Subject getSubject()`

returns the authenticated subject.

javax.security.auth.Subject 1.4

- `Set<Principal> getPrincipals()`

gets the principals of this subject.

- `static Object doAs(Subject subject, PrivilegedAction action)`

- `static Object doAs(Subject subject, PrivilegedExceptionAction action)`

- `static Object doAsPrivileged(Subject subject, PrivilegedAction action, AccessControlContext context)`

- `static Object doAsPrivileged(Subject subject, PrivilegedExceptionAction action, AccessControlContext context)`

executes the privileged action on behalf of the subject. Returns the return value of the `run` method. The `doAsPrivileged` methods execute the action in the given access control context. You can supply a “context snapshot” that you obtained earlier by calling the static method `AccessController.getContext()`, or you can supply `null` to execute the code in a new context.

java.security.PrivilegedAction 1.4

- `Object run()`

You must define this method to execute the code that you want to have executed on behalf of a subject.

java.security.PrivilegedExceptionAction 1.4

- Object run()

You must define this method to execute the code that you want to have executed on behalf of a subject. This method may throw any checked exceptions.

java.security.Principal 1.1

- String getName()

returns the identifying name of this principal.

9.3.2 JAAS Login Modules

In this section, we'll look at a JAAS example that shows you

- How to implement your own login module
- How to implement *role-based* authentication

Supplying your own login module is useful if you store login information in a database. Even if you are happy with the default module, studying a custom module will help you understand the JAAS configuration file options.

Role-based authentication is essential if you manage a large number of users. It would be impractical to put the names of all legitimate users into a policy file. Instead, the login module should map users to roles such as "admin" or "HR", and the permissions should be based on these roles.

One job of the login module is to populate the principal set of the subject that is being authenticated. If a login module supports roles, it adds Principal objects that describe roles. The Java library does not provide a class for this purpose, so we wrote our own (see Listing 9.10). The class simply stores a description/value pair, such as role=admin. Its getName method returns that pair, so we can add role-based permissions into a policy file:

```
grant principal SimplePrincipal "role=admin" { . . . }
```

Our login module looks up users, passwords, and roles in a text file that contains lines like this:

```
harry|secret|admin  
carl|guessme|HR
```

Of course, in a realistic login module, you would store this information in a database or directory.

You can find the code for the `SimpleLoginModule` in Listing 9.11. The `checkLogin` method checks whether the user name and password match a record in the password file. If so, we add two `SimplePrincipal` objects to the subject's principal set:

```
Set<Principal> principals = subject.getPrincipals();
principals.add(new SimplePrincipal("username", username));
principals.add(new SimplePrincipal("role", role));
```

The remainder of `SimpleLoginModule` is straightforward plumbing. The `initialize` method receives

- The `Subject` that is being authenticated
- A handler to retrieve login information
- A `sharedState` map that can be used for communication between login modules
- An `options` map that contains the name/value pairs that are set in the login configuration

For example, we configure our module as follows:

```
SimpleLoginModule required pwfile="password.txt";
```

The login module retrieves the `pwfile` settings from the `options` map.

The login module does not gather the user name and password; that is the job of a separate handler. This separation allows you to use the same login module without worrying whether the login information comes from a GUI dialog box, a console prompt, or a configuration file.

The handler is specified when you construct the `LoginContext`, for example:

```
LoginContext context = new LoginContext("Login1",
    new com.sun.security.auth.callback.DialogCallbackHandler());
```

The `DialogCallbackHandler` pops up a simple GUI dialog box to retrieve the user name and password. The `com.sun.security.auth.callback.TextCallbackHandler` class gets the information from the console.

However, in our application, we have our own GUI for collecting the user name and password (see Figure 9.10). We produce a simple handler that merely stores and returns that information (see Listing 9.12).

The handler has a single method, `handle`, that processes an array of `Callback` objects. A number of predefined classes, such as `NameCallback` and `PasswordCallback`, implement the `Callback` interface. You could also add your own class, such as `RetinaScanCallback`. The handler code is a bit unsightly because it needs to analyze the types of the callback objects:

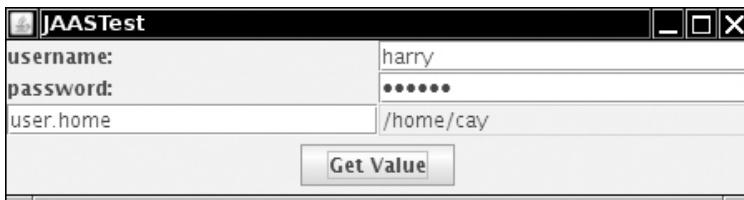


Figure 9.10 A custom login module

```
public void handle(Callback[] callbacks)
{
    for (Callback callback : callbacks)
    {
        if (callback instanceof NameCallback) . . .
        else if (callback instanceof PasswordCallback) . . .
        else . . .
    }
}
```

The login module prepares an array of the callbacks that it needs for authentication:

```
NameCallback nameCall = new NameCallback("username: ");
PasswordCallback passCall = new PasswordCallback("password: ", false);
callbackHandler.handle(new Callback[] { nameCall, passCall }));
```

Then it retrieves the information from the callbacks.

The program in Listing 9.13 displays a form for entering the login information and the name of a system property. If the user is authenticated, the property value is retrieved in a `PrivilegedAction`. As you can see from the policy file in Listing 9.14, only users with the `admin` role have permission to read properties.

As in the preceding section, you must separate the login and action code. Create two JAR files:

```
javac *.java
jar cvf login.jar JAAS*.class Simple*.class
jar cvf action.jar SysPropAction.class
```

Then run the program as

```
java -classpath login.jar:action.jar
-Djava.security.policy=JAASTest.policy
-Djava.security.auth.login.config=jaas.config
JAASTest
```

Listing 9.15 shows the policy file.

NOTE: It is possible to support a more complex two-phase protocol, whereby a login is *committed* if all modules in the login configuration were successful. For more information, see the login module developer's guide at <http://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASLMDevGuide.html>.

Listing 9.10 jaas/SimplePrincipal.java

```
1 package jaas;
2
3 import java.security.*;
4 import java.util.*;
5
6 /**
7 * A principal with a named value (such as "role=HR" or "username=harry").
8 */
9 public class SimplePrincipal implements Principal
10 {
11     private String descr;
12     private String value;
13
14     /**
15      * Constructs a SimplePrincipal to hold a description and a value.
16      * @param descr the description
17      * @param value the associated value
18      */
19     public SimplePrincipal(String descr, String value)
20     {
21         this.descr = descr;
22         this.value = value;
23     }
24
25     /**
26      * Returns the role name of this principal.
27      * @return the role name
28      */
29     public String getName()
30     {
31         return descr + "=" + value;
32     }
33
34     public boolean equals(Object otherObject)
35     {
36         if (this == otherObject) return true;
37         if (otherObject == null) return false;
38         if (getClass() != otherObject.getClass()) return false;
39         SimplePrincipal other = (SimplePrincipal) otherObject;
40         return Objects.equals(getName(), other.getName());
41     }
```

```
42
43     public int hashCode()
44     {
45         return Objects.hashCode(getName());
46     }
47 }
```

Listing 9.11 jaas/SimpleLoginModule.java

```
1 package jaas;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.security.*;
6 import java.util.*;
7 import javax.security.auth.*;
8 import javax.security.auth.callback.*;
9 import javax.security.auth.login.*;
10 import javax.security.auth.spi.*;
11
12 /**
13 * This login module authenticates users by reading usernames, passwords, and roles from a text
14 * file.
15 */
16 public class SimpleLoginModule implements LoginModule
17 {
18     private Subject subject;
19     private CallbackHandler callbackHandler;
20     private Map<String, ?> options;
21
22     public void initialize(Subject subject, CallbackHandler callbackHandler,
23                           Map<String, ?> sharedState, Map<String, ?> options)
24     {
25         this.subject = subject;
26         this.callbackHandler = callbackHandler;
27         this.options = options;
28     }
29
30     public boolean login() throws LoginException
31     {
32         if (callbackHandler == null) throw new LoginException("no handler");
33
34         NameCallback nameCall = new NameCallback("username: ");
35         PasswordCallback passCall = new PasswordCallback("password: ", false);
36         try
37         {
38             callbackHandler.handle(new Callback[] { nameCall, passCall });
39         }
```

(Continues)

Listing 9.11 (Continued)

```
40     catch (UnsupportedCallbackException e)
41     {
42         LoginException e2 = new LoginException("Unsupported callback");
43         e2.initCause(e);
44         throw e2;
45     }
46     catch (IOException e)
47     {
48         LoginException e2 = new LoginException("I/O exception in callback");
49         e2.initCause(e);
50         throw e2;
51     }
52
53     try
54     {
55         return checkLogin(nameCall.getName(), passCall.getPassword());
56     }
57     catch (IOException ex)
58     {
59         LoginException ex2 = new LoginException();
60         ex2.initCause(ex);
61         throw ex2;
62     }
63 }
64
65 /**
66 * Checks whether the authentication information is valid. If it is, the subject acquires
67 * principals for the user name and role.
68 * @param username the user name
69 * @param password a character array containing the password
70 * @return true if the authentication information is valid
71 */
72 private boolean checkLogin(String username, char[] password) throws LoginException, IOException
73 {
74     try (Scanner in = new Scanner(Paths.get("") + options.get("pwfile")), "UTF-8")
75     {
76         while (in.hasNextLine())
77         {
78             String[] inputs = in.nextLine().split("\\|");
79             if (inputs[0].equals(username) && Arrays.equals(inputs[1].toCharArray(), password))
80             {
81                 String role = inputs[2];
82                 Set<Principal> principals = subject.getPrincipals();
83                 principals.add(new SimplePrincipal("username", username));
84                 principals.add(new SimplePrincipal("role", role));
85                 return true;
86             }
87         }
88     }
```

```
88         return false;
89     }
90 }
91 public boolean logout()
92 {
93     return true;
94 }
95
96 public boolean abort()
97 {
98     return true;
99 }
100
101 public boolean commit()
102 {
103     return true;
104 }
105 }
106 }
```

Listing 9.12 jaas/SimpleCallbackHandler.java

```
1 package jaas;
2
3 import javax.security.auth.callback.*;
4
5 /**
6  * This simple callback handler presents the given user name and password.
7  */
8 public class SimpleCallbackHandler implements CallbackHandler
9 {
10    private String username;
11    private char[] password;
12
13    /**
14     * Constructs the callback handler.
15     * @param username the user name
16     * @param password a character array containing the password
17     */
18    public SimpleCallbackHandler(String username, char[] password)
19    {
20        this.username = username;
21        this.password = password;
22    }
23}
```

(Continues)

Listing 9.12 (Continued)

```
24     public void handle(Callback[] callbacks)
25     {
26         for (Callback callback : callbacks)
27         {
28             if (callback instanceof NameCallback)
29             {
30                 ((NameCallback) callback).setName(username);
31             }
32             else if (callback instanceof PasswordCallback)
33             {
34                 ((PasswordCallback) callback).setPassword(password);
35             }
36         }
37     }
38 }
```

Listing 9.13 jaas/JAASTest.java

```
1 package jaas;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7 * This program authenticates a user via a custom login and then looks up a system property with
8 * the user's privileges.
9 * @version 1.02 2016-05-10
10 * @author Cay Horstmann
11 */
12 public class JAASTest
13 {
14     public static void main(final String[] args)
15     {
16         System.setSecurityManager(new SecurityManager());
17         EventQueue.invokeLater(() ->
18         {
19             JFrame frame = new JAASFrame();
20             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21             frame.setTitle("JAASTest");
22             frame.setVisible(true);
23         });
24     }
25 }
```

Listing 9.14 jaas/JAASTest.policy

```
1 grant codebase "file:login.jar"
2 {
3     permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
4     permission java.awt.AWTPermission "accessEventQueue";
5     permission javax.security.auth.AuthPermission "createLoginContext.Login1";
6     permission javax.security.auth.AuthPermission "doAsPrivileged";
7     permission javax.security.auth.AuthPermission "modifyPrincipals";
8     permission java.io.FilePermission "jaas/password.txt", "read";
9 };
10
11 grant principal jaas.SimplePrincipal "role=admin"
12 {
13     permission java.util.PropertyPermission "*", "read";
14 };
```

Listing 9.15 jaas/jaas.config

```
1 Login1
2 {
3     jaas.SimpleLoginModule required pwfile="jaas/password.txt" debug=true;
4 };
```

javax.security.auth.callback.CallbackHandler 1.4

- void handle(Callback[] callbacks)

handles the given callbacks, interacting with the user if desired, and stores the security information in the callback objects.

javax.security.auth.callback.NameCallback 1.4

- NameCallback(String prompt)
- NameCallback(String prompt, String defaultValue)
constructs a NameCallback with the given prompt and default name.
- String getName()
- void setName(String name)
gets or sets the name gathered by this callback.
- String getPrompt()
gets the prompt to use when querying this name.
- String getDefaultName()
gets the default name to use when querying this name.

`javax.security.auth.callback.PasswordCallback 1.4`

- `PasswordCallback(String prompt, boolean echoOn)`
constructs a `PasswordCallback` with the given prompt and echo flag.
- `char[] getPassword()`
- `void setPassword(char[] password)`
gets or sets the password gathered by this callback.
- `String getPrompt()`
gets the prompt to use when querying this password.
- `boolean isEchoOn()`
gets the echo flag to use when querying this password.

`javax.security.auth.spi.LoginModule 1.4`

- `void initialize(Subject subject, CallbackHandler handler, Map<String,?> sharedState, Map<String,?> options)`
initializes this `LoginModule` for authenticating the given `subject`. During login processing, uses the given `handler` to gather login information. Use the `sharedState` map for communicating with other login modules. The `options` map contains the name/value pairs specified in the login configuration for this module instance.
- `boolean login()`
carries out the authentication process and populates the `subject`'s principals. Returns `true` if the login was successful.
- `boolean commit()`
is called after all login modules were successful, for login scenarios that require a two-phase commit. Returns `true` if the operation was successful.
- `boolean abort()`
is called if the failure of another login module caused the login process to abort. Returns `true` if the operation was successful.
- `boolean logout()`
logs out this `subject`. Returns `true` if the operation was successful.

9.4 Digital Signatures

As we said earlier, applets were what started the Java craze. In practice, people discovered that although they could write animated applets (like the famous

“nervous text”), applets could not do a whole lot of useful stuff in the JDK 1.0 security model. For example, since applets under JDK 1.0 were so closely supervised, they couldn’t do much good on a corporate intranet, even though relatively little risk attaches to executing an applet from your company’s secure intranet. It quickly became clear to Sun that for applets to become truly useful, users need to be able to assign *different* levels of security, depending on where the applet originated. If an applet comes from a trusted supplier and has not been tampered with, the user of that applet can decide whether to give the applet more privileges.

To give more trust to an applet, we need to know two things:

- Where did the applet come from?
- Was the code corrupted in transit?

In the past 50 years, mathematicians and computer scientists have developed sophisticated algorithms for ensuring the integrity of data and for electronic signatures. The `java.security` package contains implementations of many of these algorithms. Fortunately, you don’t need to understand the underlying mathematics to use the algorithms in the `java.security` package. In the next sections, we’ll show you how message digests can detect changes in data files and how digital signatures can prove the identity of the signer.

9.4.1 Message Digests

A message digest is a digital fingerprint of a block of data. For example, the so-called SHA-1 (Secure Hash Algorithm #1) condenses any data block, no matter how long, into a sequence of 160 bits (20 bytes). As with real fingerprints, one hopes that no two messages have the same SHA-1 fingerprint. Of course, that cannot be true—there are only 2^{160} SHA-1 fingerprints, so there must be some messages with the same fingerprint. But 2^{160} is so large that the probability of duplication occurring is negligible. How negligible? According to James Walsh in *True Odds: How Risks Affect Your Everyday Life* (Merritt Publishing, 1996), the chance that you will die from being struck by lightning is about one in 30,000. Now, think of nine other people—for example, your nine least favorite managers or professors. The chance that you and *all of them* will die from lightning strikes is higher than that of a forged message having the same SHA-1 fingerprint as the original. (Of course, more than ten people, none of whom you are likely to know, *will* die from lightning strikes. However, we are talking about the far slimmer chance that *your particular choice* of people will be wiped out.)

A message digest has two essential properties:

- If one bit or several bits of the data are changed, the message digest also changes.

- A forger who is in possession of a given message cannot construct a fake message that has the same message digest as the original.

The second property is, again, a matter of probabilities. Consider the following message by the billionaire father:

"Upon my death, my property shall be divided equally among my children; however, my son George shall receive nothing."

That message (with a final newline) has an SHA-1 fingerprint of

```
12 5F 09 03 E7 31 30 19 2E A6 E7 E4 90 43 84 B4 38 99 8F 67
```

The distrustful father has deposited the message with one attorney and the fingerprint with another. Now, suppose George bribes the lawyer holding the message. He wants to change the message so that Bill gets nothing. Of course, that changes the fingerprint to a completely different bit pattern:

```
7D F6 AB 08 EB 40 EC CD AB 74 ED E9 86 F9 ED 99 D1 45 B1 57
```

Can George find some other wording that matches the fingerprint? If he had been the proud owner of a billion computers from the time the Earth was formed, each computing a million messages a second, he would not yet have found a message he could substitute.

A number of algorithms have been designed to compute such message digests. Among them are SHA-1, the secure hash algorithm developed by the National Institute of Standards and Technology, and MD5, an algorithm invented by Ronald Rivest of MIT. Both algorithms scramble the bits of a message in ingenious ways. For details about these algorithms, see, for example, *Cryptography and Network Security, Fifth Edition*, by William Stallings (Prentice Hall, 2011). However, subtle regularities have been discovered in both algorithms, and NIST recommends to switch to stronger alternatives such as SHA-256, SHA-384, or SHA-512.

The Java programming language implements MD5, SHA-1, SHA-256, SHA-384, and SHA-512. The `MessageDigest` class is a *factory* for creating objects that encapsulate the fingerprinting algorithms. It has a static method, called `getInstance`, that returns an object of a class that extends the `MessageDigest` class. This means the `MessageDigest` class serves double duty:

- As a factory class
- As the superclass for all message digest algorithms

For example, here is how you obtain an object that can compute SHA fingerprints:

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

(To get an object that can compute MD5, use the string "MD5" as the argument to `getInstance`.)

After you have obtained a `MessageDigest` object, feed it all the bytes in the message by repeatedly calling the `update` method. For example, the following code passes all bytes in a file to the `alg` object just created to do the fingerprinting:

```
InputStream in = . . .
int ch;
while ((ch = in.read()) != -1)
    alg.update((byte) ch);
```

Alternatively, if you have the bytes in an array, you can update the entire array at once:

```
byte[] bytes = . . .;
alg.update(bytes);
```

When you are done, call the `digest` method. This method pads the input as required by the fingerprinting algorithm, does the computation, and returns the digest as an array of bytes.

```
byte[] hash = alg.digest();
```

The program in Listing 9.16 computes a message digest, using MD5, SHA-1, SHA-256, SHA-384, or SHA-512. Run it as

```
java hash.Digest hash/input.txt
```

or

```
java hash.Digest hash/input.txt MD5
```

Listing 9.16 hash/Digest.java

```
1 package hash;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.security.*;
6
7 /**
8  * This program computes the message digest of a file.
9  * @version 1.20 2012-06-16
10 * @author Cay Horstmann
11 */
12 public class Digest
13 {
14     /**
15      * @param args args[0] is the filename, args[1] is optionally the algorithm
16      * (SHA-1, SHA-256, or MD5)
17      */
18 }
```

(Continues)

Listing 9.16 (Continued)

```
18  public static void main(String[] args) throws IOException, GeneralSecurityException
19  {
20      String algname = args.length >= 2 ? args[1] : "SHA-1";
21      MessageDigest alg = MessageDigest.getInstance(algname);
22      byte[] input = Files.readAllBytes(Paths.get(args[0]));
23      byte[] hash = alg.digest(input);
24      String d = "";
25      for (int i = 0; i < hash.length; i++)
26      {
27          int v = hash[i] & 0xFF;
28          if (v < 16) d += "0";
29          d += Integer.toString(v, 16).toUpperCase() + " ";
30      }
31      System.out.println(d);
32  }
33 }
```

java.security.MessageDigest 1.1

- `static MessageDigest getInstance(String algorithmName)`
returns a `MessageDigest` object that implements the specified algorithm. Throws `NoSuchAlgorithmException` if the algorithm is not provided.
- `void update(byte input)`
- `void update(byte[] input)`
- `void update(byte[] input, int offset, int len)`
updates the digest, using the specified bytes.
- `byte[] digest()`
completes the hash computation, returns the computed digest, and resets the algorithm object.
- `void reset()`
resets the digest.

9.4.2 Message Signing

In the last section, you saw how to compute a message digest—a fingerprint for the original message. If the message is altered, the fingerprint of the altered message will not match the fingerprint of the original. If the message and its fingerprint are delivered separately, the recipient can check whether the message has been tampered with. However, if both the message and the fingerprint were intercepted, it is an easy matter to modify the message and then recompute the

fingerprint. After all, the message digest algorithms are publicly known, and they don't require secret keys. In that case, the recipient of the forged message and the recomputed fingerprint would never know that the message has been altered. Digital signatures solve this problem.

To help you understand how digital signatures work, we'll explain a few concepts from the field called *public key cryptography*. Public key cryptography is based on the notion of a *public key* and *private key*. The idea is that you tell everyone in the world your public key. However, only you hold the private key, and it is important that you safeguard it and don't release it to anyone else. The keys are matched by mathematical relationships, though the exact nature of these relationships is not important to us. (If you are interested, look it up in *The Handbook of Applied Cryptography* at www.cacr.math.uwaterloo.ca/hac.)

The keys are quite long and complex. For example, here is a matching pair of public and private Digital Signature Algorithm (DSA) keys.

Public key:

p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df63413c5e12ed089
9bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73da179069b32e293563
0e1c2062354d0da20a6c416e50be794ca4

y: c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a8123ce5a8018b8161a760480fadd040b92
7281ddb22cb9bc4df596d7de4d1b977d50

Private key:

p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df63413c5e12ed089
9bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73da179069b32e293563
0e1c2062354d0da20a6c416e50be794ca4

x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a

It is believed to be practically impossible to compute one key from the other. That is, even though everyone knows your public key, they can't compute your private key in your lifetime, no matter how many computing resources they have available.

It might seem difficult to believe that you can't compute the private key from the public key, but nobody has ever found an algorithm to do this for the encryption

algorithms in common use today. If the keys are sufficiently long, brute force—simply trying all possible keys—would require more computers than can be built from all the atoms in the solar system, crunching away for thousands of years. Of course it is possible that someone could come up with algorithms for computing keys that are much more clever than brute force. For example, the RSA algorithm (the encryption algorithm invented by Rivest, Shamir, and Adleman) depends on the difficulty of factoring large numbers. For the last 20 years, many of the best mathematicians have tried to come up with good factoring algorithms, but so far with no success. For that reason, most cryptographers believe that keys with a “modulus” of 2,000 bits or more are currently completely safe from any attack. DSA is believed to be similarly secure.

Figure 9.11 illustrates how the process works in practice.

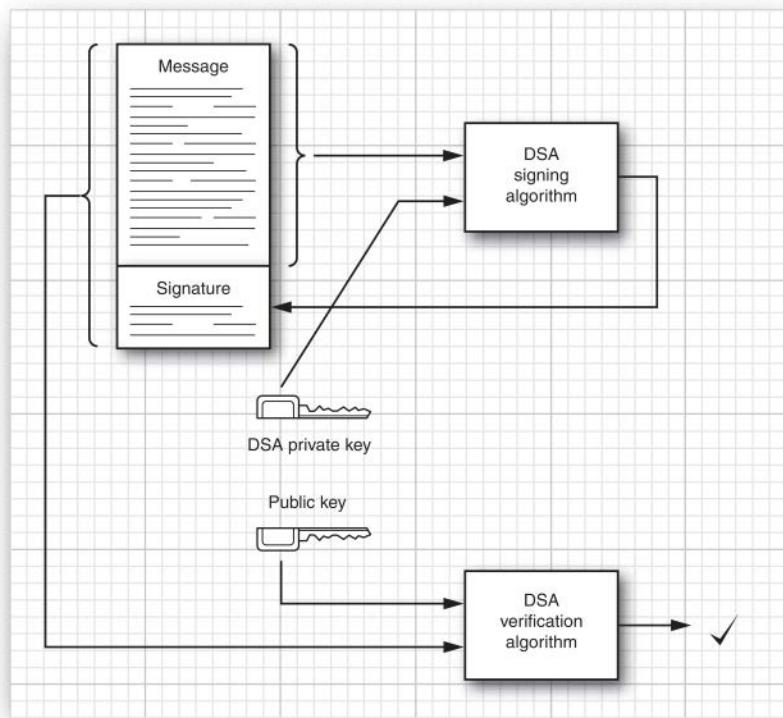


Figure 9.11 Public key signature exchange with DSA

Suppose Alice wants to send Bob a message, and Bob wants to know this message came from Alice and not an impostor. Alice writes the message and *signs* the message digest with her private key. Bob gets a copy of her public key. Bob then applies the public key to *verify* the signature. If the verification passes, Bob can be assured of two facts:

- The original message has not been altered.
- The message was signed by Alice, the holder of the private key that matches the public key that Bob used for verification.

You can see why the security of private keys is so important. If someone steals Alice's private key, or if a government can require her to turn it over, then she is in trouble. The thief or a government agent can now impersonate her by sending messages, such as money transfer instructions, that others will believe came from Alice.

9.4.3 Verifying a Signature

The JDK comes with the `keytool` program, which is a command-line tool to generate and manage a set of certificates. We expect that ultimately the functionality of this tool will be embedded in other, more user-friendly programs. But right now, we'll use `keytool` to show how Alice can sign a document and send it to Bob, and how Bob can verify that the document really was signed by Alice and not an impostor.

The `keytool` program manages *keystores*, databases of certificates and private/public key pairs. Each entry in the keystore has an *alias*. Here is how Alice creates a keystore, `alice.certs`, and generates a key pair with alias `alice`:

```
keytool -genkeypair -keystore alice.certs -alias alice
```

When creating or opening a keystore, you are prompted for a keystore password. For this example, just use `secret`. If you were to use the `keytool`-generated keystore for any serious purpose, you would need to choose a good password and safeguard this file.

When generating a key, you are prompted for the following information:

```
Enter keystore password: secret
Reenter new password: secret
What is your first and last name?
[Unknown]: Alice Lee
```

```
What is the name of your organizational unit?  
[Unknown]: Engineering  
What is the name of your organization?  
[Unknown]: ACME Software  
What is the name of your City or Locality?  
[Unknown]: San Francisco  
What is the name of your State or Province?  
[Unknown]: CA  
What is the two-letter country code for this unit?  
[Unknown]: US  
Is <CN=Alice Lee, OU=Engineering, O=ACME Software, L=San Francisco, ST=CA, C=US> correct?  
[no]: yes
```

The keytool uses names in the X.500 format, whose components are Common Name (CN), Organizational Unit (OU), Organization (O), Location (L), State (ST), and Country (C), to identify key owners and certificate issuers.

Finally, specify a key password, or press Enter to use the keystore password as the key password.

Suppose Alice wants to give her public key to Bob. She needs to export a certificate file:

```
keytool -exportcert -keystore alice.certs -alias alice -file alice.cer
```

Now Alice can send the certificate to Bob. When Bob receives the certificate, he can print it:

```
keytool -printcert -file alice.cer
```

The printout looks like this:

```
Owner: CN=Alice Lee, OU=Engineering, O=ACME Software, L=San Francisco, ST=CA, C=US  
Issuer: CN=Alice Lee, OU=Engineering, O=ACME Software, L=San Francisco, ST=CA, C=US  
Serial number: 470835ce  
Valid from: Sat Oct 06 18:26:38 PDT 2007 until: Fri Jan 04 17:26:38 PST 2008  
Certificate fingerprints:  
MD5: BC:18:15:27:85:69:48:B1:5A:C3:0B:1C:C6:11:B7:81  
SHA1: 31:0A:A0:B8:C2:8B:3B:B6:85:7C:EF:C0:57:E5:94:95:61:47:6D:34  
Signature algorithm name: SHA1withDSA  
Version: 3
```

If Bob wants to check that he got the right certificate, he can call Alice and verify the certificate fingerprint over the phone.

NOTE: Some certificate issuers publish certificate fingerprints on their web sites.

For example, to check the VeriSign certificate in the keystore

jre/lib/security/cacerts directory, use the *-list* option:

```
keytool -list -v -keystore jre/lib/security/cacerts
```

The password for this keystore is *changeit*. One of the certificates in this keystore is

```
Owner: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorized use only",  
OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign, Inc.", C=US  
Issuer: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorized  
use only", OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign, Inc.",  
C=US  
Serial number: 4cc7eaaa983e71d39310f83d3a899192  
Valid from: Sun May 17 17:00:00 PDT 1998 until: Tue Aug 01 16:59:59 PDT 2028  
Certificate fingerprints:  
MD5: DB:23:3D:F9:69:FA:4B:B9:95:80:44:73:5E:7D:41:83  
SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
```

You can check that your certificate is valid by visiting the web site
www.verisign.com/repository/root.html.

Once Bob trusts the certificate, he can import it into his keystore.

```
keytool -importcert -keystore bob.certs -alias alice -file alice.cer
```



CAUTION: Never import into a keystore a certificate that you don't fully trust.

Once a certificate is added to the keystore, any program that uses the keystore assumes that the certificate can be used to verify signatures.

Now Alice can start sending signed documents to Bob. The *jarsigner* tool signs and verifies JAR files. Alice simply adds the document to be signed into a JAR file.

```
jar cvf document.jar document.txt
```

She then uses the *jarsigner* tool to add the signature to the file. She needs to specify the keystore, the JAR file, and the alias of the key to use.

```
jarsigner -keystore alice.certs document.jar alice
```

When Bob receives the file, he uses the *-verify* option of the *jarsigner* program.

```
jarsigner -verify -keystore bob.certs document.jar
```

Bob does not need to specify the key alias. The `jarsigner` program finds the X.500 name of the key owner in the digital signature and looks for a matching certificate in the keystore.

If the JAR file is not corrupted and the signature matches, the `jarsigner` program prints

```
jar verified.
```

Otherwise, the program displays an error message.

9.4.4 The Authentication Problem

Suppose you get a message from your friend Alice, signed with her private key, using the method we just showed you. You might already have her public key, or you can easily get it by asking her for a copy or by getting it from her web page. Then, you can verify that the message was in fact authored by Alice and has not been tampered with. Now, suppose you get a message from a stranger who claims to represent a famous software company, urging you to run a program attached to the message. The stranger even sends you a copy of his public key so you can verify that he authored the message. You check that the signature is valid. This proves that the message was signed with the matching private key and has not been corrupted.

Be careful: *You still have no idea who wrote the message.* Anyone could have generated a pair of public and private keys, signed the message with the private key, and sent the signed message and the public key to you. The problem of determining the identity of the sender is called the *authentication problem*.

The usual way to solve the authentication problem is simple. Suppose the stranger and you have a common acquaintance you both trust. Suppose the stranger meets your acquaintance in person and hands over a disk with the public key. Your acquaintance later meets you, assures you that he met the stranger and that the stranger indeed works for the famous software company, and then gives you the disk (see Figure 9.12). That way, your acquaintance vouches for the authenticity of the stranger.

In fact, your acquaintance does not actually need to meet you. Instead, he can use his private key to sign the stranger's public key file (see Figure 9.13).

When you get the public key file, you verify the signature of your friend, and because you trust him, you are confident that he did check the stranger's credentials before applying his signature.

However, you might not have a common acquaintance. Some trust models assume that there is always a "chain of trust"—a chain of mutual acquaintances—so that you trust every member of that chain. In practice, of course, that isn't always

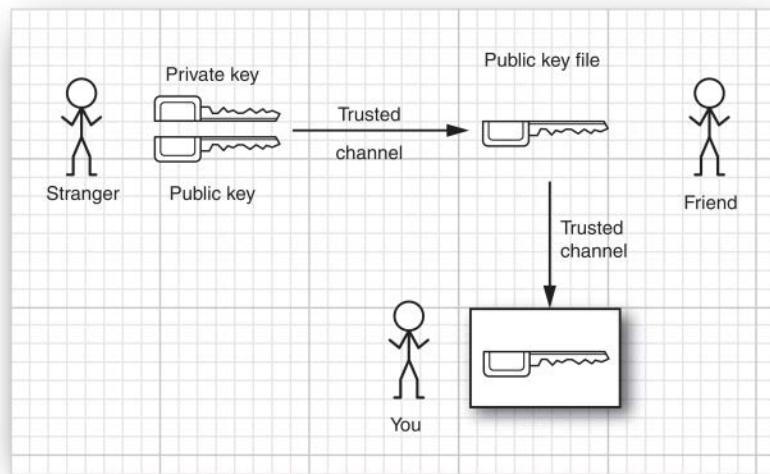


Figure 9.12 Authentication through a trusted intermediary

true. You might trust your friend, Alice, and you know that Alice trusts Bob, but you don't know Bob and aren't sure that you trust him. Other trust models assume that there is a benevolent big brother—a company in which we all trust. The best known of such companies is VeriSign, Inc. (www.verisign.com).

You will often encounter digital signatures signed by one or more entities who will vouch for the authenticity, and you will need to evaluate to what degree you trust the authenticators. You might place a great deal of trust in VeriSign, perhaps because you saw their logo on many web pages or because you heard that they require multiple people with black attaché cases to come together into a secure chamber whenever new master keys are to be minted.

However, you should have realistic expectations about what is actually being authenticated. The CEO of VeriSign does not personally meet every individual or company representative when authenticating a public key. You can get a "class 1" ID simply by filling out a web form and paying a small fee. The key is mailed to the e-mail address included in the certificate. Thus, you can be reasonably assured that the e-mail address is genuine, but the requestor could have filled in *any* name and organization. There are more stringent classes of IDs. For example, with a "class 3" ID, VeriSign will require an individual requestor to appear before a notary public, and it will check the financial rating of a corporate requestor. Other authenticators will have different procedures. Thus, when you receive an authenticated message, it is important that you understand what, in fact, is being authenticated.

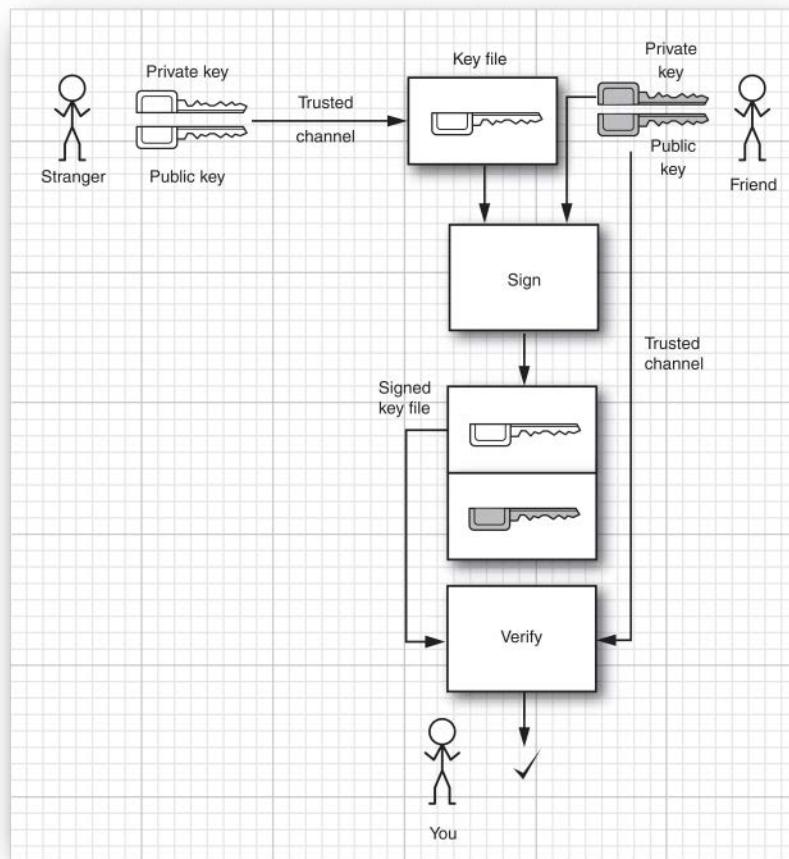


Figure 9.13 Authentication through a trusted intermediary's signature

9.4.5 Certificate Signing

In Section 9.4.3, “Verifying a Signature,” on p. 553 you saw how Alice used a self-signed certificate to distribute a public key to Bob. However, Bob needed to ensure that the certificate was valid by verifying the fingerprint with Alice.

Suppose Alice wants to send her colleague Cindy a signed message, but Cindy doesn’t want to bother with verifying lots of signature fingerprints. Now suppose there is an entity that Cindy trusts to verify signatures. In this example, Cindy trusts the Information Resources Department at ACME Software.

That department operates a *certificate authority* (CA). Everyone at ACME has the CA's public key in their keystore, installed by a system administrator who carefully checked the key fingerprint. The CA signs the keys of ACME employees. When they install each other's keys, the keystore will trust them implicitly because they are signed by a trusted key.

Here is how you can simulate this process. Create a keystore `acmesoft.certs`. Generate a key pair and export the public key:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot  
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer
```

The public key is exported into a "self-signed" certificate. Then add it to every employee's keystore.

```
keytool -importcert -keystore cindy.certs -alias acmeroot -file acmeroot.cer
```

For Alice to send messages to Cindy and to everyone else at ACME Software, she needs to bring her certificate to the Information Resources Department and have it signed. Unfortunately, this functionality is missing in the `keytool` program. In the book's companion code, we supply a `CertificateSigner` class to fill the gap. An authorized staff member at ACME Software would verify Alice's identity and generate a signed certificate as follows:

```
java CertificateSigner -keystore acmesoft.certs -alias acmeroot  
-infile alice.cer -outfile alice_signedby_acmeroot.cer
```

The certificate signer program must have access to the ACME Software keystore, and the staff member must know the keystore password. Clearly, this is a sensitive operation.

Alice gives the file `alice_signedby_acmeroot.cer` file to Cindy and to anyone else in ACME Software. Alternatively, ACME Software can simply store the file in a company directory. Remember, this file contains Alice's public key and an assertion by ACME Software that this key really belongs to Alice.

Now Cindy imports the signed certificate into her keystore:

```
keytool -importcert -keystore cindy.certs -alias alice -file alice_signedby_acmeroot.cer
```

The keystore verifies that the key was signed by a trusted root key that is already present in the keystore. Cindy is *not* asked to verify the certificate fingerprint.

Once Cindy has added the root certificate and the certificates of the people who regularly send her documents, she never has to worry about the keystore again.

9.4.6 Certificate Requests

In the preceding section, we simulated a CA with a keystore and the `CertificateSigner` tool. However, most CAs run more sophisticated software to manage certificates, and they use slightly different formats for certificates. This section shows the added steps required to interact with those software packages.

We will use the OpenSSL software package as an example. The software is preinstalled on many Linux systems and Mac OS X, and a Cygwin port is also available. You can also download the software at www.openssl.org.

To create a CA, run the `CA` script. The exact location depends on your operating system. On Ubuntu, run

```
/usr/lib/ssl/misc/CA.pl -newca
```

This script creates a subdirectory called `demoCA` in the current directory. The directory contains a root key pair and storage for certificates and certificate revocation lists.

You will want to import the public key into the Java keystores of all employees, but it is in the Privacy Enhanced Mail (PEM) format, not the DER format that the keystore accepts easily. Copy the file `demoCA/cacert.pem` to a file `acmeroot.pem` and open that file in a text editor. Remove everything before the line

```
-----BEGIN CERTIFICATE-----
```

and after the line

```
-----END CERTIFICATE-----
```

Now you can import `acmeroot.pem` into each keystore in the usual way:

```
keytool -importcert -keystore cindy.certs -alias alice -file acmeroot.pem
```

It seems quite incredible that the `keytool` cannot carry out this editing operation itself.

To sign Alice's public key, you start by generating a *certificate request* that contains the certificate in the PEM format:

```
keytool -certreq -keystore alice.store -alias alice -file alice.pem
```

To sign the certificate, run

```
openssl ca -in alice.pem -out alice_signedby_acmeroot.pem
```

As before, cut out everything outside the BEGIN CERTIFICATE / END CERTIFICATE markers from `alice_signedby_acmeroot.pem`. Then import it into the keystore:

```
keytool -importcert -keystore cindy.certs -alias alice -file alice_signedby_acmeroot.pem
```

You can use the same steps to have a certificate signed by a public certificate authority such as VeriSign.

9.4.7 Code Signing

A common use of authentication technology is signing executable programs. If you download a program, you are naturally concerned about the damage it can do. For example, the program could have been infected by a virus. If you know where the code comes from *and* that it has not been tampered with since it left its origin, your comfort level will be a lot higher than without this knowledge.

In this section, we'll show you how to sign JAR files, and how you can configure Java to verify the signature. This capability was designed for the Java Plug-in, the launching point for applets and Java Web Start applications. These are no longer commonly used technologies, but you may still need to support them in legacy products.

When Java was first released, applets ran in the "sandbox," with limited permissions, as soon as they were loaded. If users wanted to use applets that can access the local file system, make network connections, and so on, they had to explicitly agree. To ensure that the applet code was not tampered with in transit, it had to be digitally signed.

Here is a specific example. Suppose that while surfing the Internet, you encounter a web site that offers to run an applet from an unfamiliar vendor, provided you grant it the permission to do so (see Figure 9.14). Such a program is signed with a *software developer* certificate issued by a certificate authority that the Java runtime trusts. The pop-up dialog box identifies the software developer and the certificate issuer. Now you need to decide whether to authorize the program.

What facts do you have at your disposal that might influence your decision? Here is what you know:

- Thawte sold a certificate to the software developer.
- The program really was signed with that certificate, and it hasn't been modified in transit.
- The certificate really was signed by Thawte—it was verified by the public key in the local cacerts file.

Does that tell you whether the code is safe to run? Can you trust a vendor if all you know is the vendor's name and the fact that Thawte sold them a software developer certificate? One would like to think that Thawte went to some degree of trouble to assure itself that ChemAxon Kft. is not an outright cracker. However, no certificate issuer carries out a comprehensive audit of the honesty and

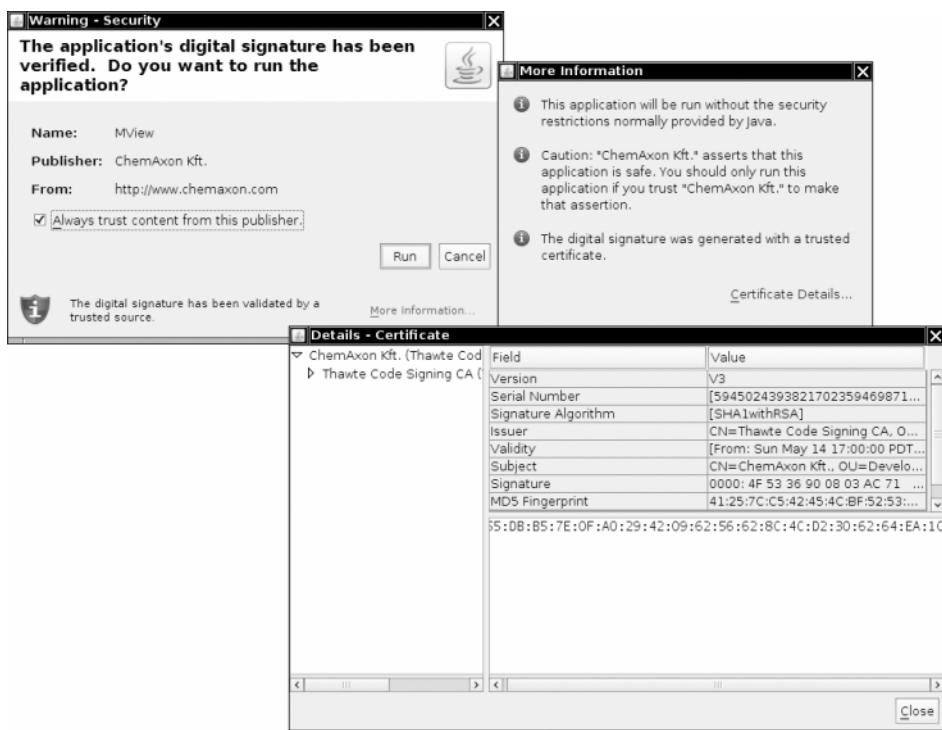


Figure 9.14 Launching a signed applet

competence of software vendors. They merely verify the identity, typically by inspecting a scanned copy of a business license or passport.

As you can see, this is not a satisfactory solution. A better way might have been to expand the functionality of the sandbox. When the Java Web Start technology was first released, it went beyond the sandbox and enabled users to agree to limited file and printer access. However, that concept was never further developed. Instead, the opposite happened. When the sandbox was under attack by hackers, Oracle found it too difficult to keep up and discontinued support for unsigned applets altogether.

Nowadays, applets are quite uncommon and mostly used for legacy purposes. If you need to support an applet served to the public, sign it with a certificate from a vendor who is trusted by the Java runtime environment.

For intranet application, one can do a bit better. One can install policy files and certificates on local machines so that no user interaction is required for launching

code from trusted sources. Whenever the Java Plug-in tool loads signed code, it consults the policy file for the permissions and the keystore for signatures.

For the remainder of this section, we will describe how you can build policy files that grant specific permissions to code from known sources. Building and deploying these policy files is not for casual end users. However, system administrators can carry out these tasks in preparation for distributing intranet programs.

Suppose ACME Software wants its employees to run certain programs that require local file access, and it wants to deploy these programs through a browser as applets or Web Start applications.

As you saw earlier in this chapter, ACME could identify the programs by their code base. But that means ACME would need to update the policy files each time the programs are moved to a different web server. Instead, ACME decides to *sign* the JAR files that contain the program code.

First, ACME generates a root certificate:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
```

Of course, the keystore containing the private root key must be kept in a safe place. Therefore, we create a second keystore client.certs for the public certificates and add the public acmeroot certificate into it.

```
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer  
keytool -importcert -keystore client.certs -alias acmeroot -file acmeroot.cer
```

To make a signed JAR file, programmers add their class files to a JAR file in the usual way. For example,

```
javac FileReadApplet.java  
jar cvf FileReadApplet.jar *.class
```

Then a trusted person at ACME runs the jarsigner tool, specifying the JAR file and the alias of the private key:

```
jarsigner -keystore acmesoft.certs FileReadApplet.jar acmeroot
```

The signed applet is now ready to be deployed on a web server.

Next, let us turn to the client machine configuration. A policy file must be distributed to each client machine.

To reference a keystore, a policy file starts with the line

```
keystore "keystoreURL", "keystoreType";
```

The URL can be absolute or relative. Relative URLs are relative to the location of the policy file. The type is JKS if the keystore was generated by keytool. For example,

```
keystore "client.certs", "JKS";
```

Then grant clauses can have suffixes `signedBy "alias"`, such as this one:

```
grant signedBy "acmeroot"  
{  
    . . .  
};
```

Any signed code that can be verified with the public key associated with the alias is now granted the permissions inside the grant clause.

You can try out the code signing process with the applet in Listing 9.17. The applet tries to read from a local file. The default security policy only lets the applet read files from its code base and any subdirectories. Use `appletviewer` to run the applet and verify that you can view files from the code base directory, but not from other directories.

We provide a policy file `applet.policy` with the contents:

```
keystore "client.certs", "JKS";  
grant signedBy "acmeroot"  
{  
    permission java.lang.RuntimePermission "usePolicy";  
    permission java.io.FilePermission "/etc/*", "read";  
};
```

The `usePolicy` permission overrides the default “all or nothing” permission for signed applets. Here, we say that any applets signed by `acmeroot` are allowed to read files in the `/etc` directory. (Windows users: Substitute another directory such as `C:\Windows`.)

Tell the applet viewer to use the policy file:

```
appletviewer -J-Djava.security.policy=applet.policy FileReadApplet.html
```

Now the applet can read files from the `/etc` directory, thus demonstrating that the signing mechanism works.



TIP: If you have trouble getting this step to work, add the option `-J-Djava.security.debug=policy`, and you will be rewarded with detailed messages that trace how the program establishes the security policy.

As a final test, you can run your applet inside the browser (see Figure 9.15). You need to copy the permission file and the keystore inside the Java deployment directory. If you run UNIX or Linux, that directory is the `.java/deployment` subdirectory

of your home directory. In Windows, it is the `C:\Users\yourLoginName\AppData\Sun\Java\Deployment` directory. In the following, we'll refer to that directory as *deploydir*.

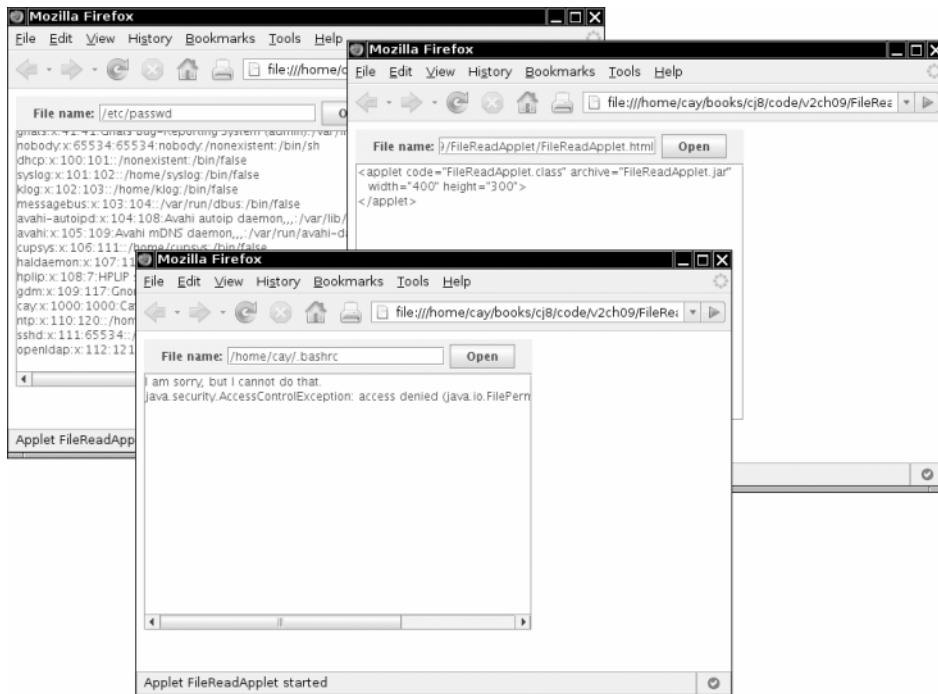


Figure 9.15 A signed applet can read local files.

Copy `applet.policy` and `client.certs` to the *deploydir/security* directory. In that directory, rename `applet.policy` to `java.policy`. (Double-check that you are not wiping out an existing `java.policy` file. If there is one, add the `applet.policy` contents to it.)

Restart your browser and load the `FileReadApplet.html`. You should *not* be prompted to accept any certificate. Check that you can load any file from the `/etc` directory and the directory from which the applet was loaded, but not from other directories.

When you are done, remember to clean up your *deploydir/security* directory. Remove the files `java.policy` and `client.certs`. Restart your browser. If you load the applet again after cleaning up, you should no longer be able to read files from the local file system. Instead, you will be prompted for a certificate. We'll discuss security certificates in the next section.



TIP: For more details on configuring client Java security, we refer you to the Java Rich Internet Applications Guide at <http://docs.oracle.com/javase/8/docs/technotes/guides/jweb/index.html>.

Listing 9.17 signed/FileReadApplet.java

```
1 package signed;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.nio.file.*;
7 import javax.swing.*;
8
9 /**
10 * This applet can run "outside the sandbox" and read local files when it is given the right
11 * permissions.
12 * @version 1.13 2016-05-10
13 * @author Cay Horstmann
14 */
15 public class FileReadApplet extends JApplet
16 {
17     private JTextField fileNameField;
18     private JTextArea fileText;
19
20     public void init()
21     {
22         EventQueue.invokeLater(() ->
23         {
24             fileNameField = new JTextField(20);
25             JPanel panel = new JPanel();
26             panel.add(new JLabel("File name:"));
27             panel.add(fileNameField);
28             JButton openButton = new JButton("Open");
29             panel.add(openButton);
30             ActionListener listener = event -> loadFile(fileNameField.getText());
31             fileNameField.addActionListener(listener);
32             openButton.addActionListener(listener);
33             add(panel, "North");
34             fileText = new JTextArea();
35             add(new JScrollPane(fileText), "Center");
36         });
37     }
38
39 /**
40 * Loads the contents of a file into the text area.
41 * @param filename the file name
42 */
```

```
43 public void loadFile(String filename)
44 {
45     fileText.setText("");
46     try
47     {
48         fileText.append(new String(Files.readAllBytes(Paths.get(filename))));
49     }
50     catch (IOException ex)
51     {
52         fileText.append(ex + "\n");
53     }
54     catch (SecurityException ex)
55     {
56         fileText.append("I am sorry, but I cannot do that.\n");
57         fileText.append(ex + "\n");
58         ex.printStackTrace();
59     }
60 }
61 }
```

9.5 Encryption

So far, we have discussed one important cryptographic technique implemented in the Java security API—namely, authentication through digital signatures. A second important aspect of security is *encryption*. Even when authenticated, the information itself is plainly visible. The digital signature merely verifies that the information has not been changed. In contrast, when information is encrypted, it is not visible. It can only be decrypted with a matching key.

Authentication is sufficient for code signing—there is no need to hide the code. However, encryption is necessary when applets or applications transfer confidential information, such as credit card numbers and other personal data.

In the past, patents and export controls prevented many companies from offering strong encryption. Fortunately, export controls are now much less stringent, and the patents for important algorithms have expired. Nowadays, Java SE has excellent encryption support as a part of the standard library.

9.5.1 Symmetric Ciphers

The Java cryptographic extensions contain a class `Cipher` that is the superclass of all encryption algorithms. To get a cipher object, call the `getInstance` method:

```
Cipher cipher = Cipher.getInstance(algorithmName);
```

or

```
Cipher cipher = Cipher.getInstance(algorithmName, providerName);
```

The JDK comes with ciphers by the provider named "SunJCE". It is the default provider used if you don't specify another provider name. You might want another provider if you need specialized algorithms that Oracle does not support.

The algorithm name is a string such as "AES" or "DES/CBC/PKCS5Padding".

The Data Encryption Standard (DES) is a venerable block cipher with a key length of 56 bits. Nowadays, the DES algorithm is considered obsolete because it can be cracked with brute force (see, for example, http://w2.eff.org/Privacy/Crypto/Crypto_mis/DESCracker). A far better alternative is its successor, the Advanced Encryption Standard (AES). See www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf for a detailed description of the AES algorithm. We use AES for our example.

Once you have a cipher object, initialize it by setting the mode and the key:

```
int mode = . . .;
Key key = . . .;
cipher.init(mode, key);
```

The mode is one of

```
Cipher.ENCRYPT_MODE
Cipher.DECRYPT_MODE
Cipher.WRAP_MODE
Cipher.UNWRAP_MODE
```

The wrap and unwrap modes encrypt one key with another—see the next section for an example.

Now you can repeatedly call the `update` method to encrypt blocks of data:

```
int blockSize = cipher.getBlockSize();
byte[] inBytes = new byte[blockSize];
. . . // read inBytes
int outputSize= cipher.getOutputSize(blockSize);
byte[] outBytes = new byte[outputSize];
int outLength = cipher.update(inBytes, 0, outputSize, outBytes);
. . . // write outBytes
```

When you are done, you must call the `doFinal` method once. If a final block of input data is available (with fewer than `blockSize` bytes), call

```
outBytes = cipher.doFinal(inBytes, 0, inLength);
```

If all input data have been encrypted, instead call

```
outBytes = cipher.doFinal();
```

The call to `doFinal` is necessary to carry out *padding* of the final block. Consider the DES cipher. It has a block size of eight bytes. Suppose the last block of the input

data has fewer than eight bytes. Of course, we can fill the remaining bytes with 0, to obtain one final block of eight bytes, and encrypt it. But when the blocks are decrypted, the result will have several trailing 0 bytes appended to it, and therefore will be slightly different from the original input file. To avoid this problem, we need a *padding scheme*. A commonly used padding scheme is the one described in the Public Key Cryptography Standard (PKCS) #5 by RSA Security, Inc. (<https://tools.ietf.org/html/rfc2898>).

In this scheme, the last block is not padded with a pad value of zero, but with a pad value that equals the number of pad bytes. In other words, if L is the last (incomplete) block, it is padded as follows:

L 01	if length(L) = 7
L 02 02	if length(L) = 6
L 03 03 03	if length(L) = 5
...	
L 07 07 07 07 07 07 07	if length(L) = 1

Finally, if the length of the input is actually divisible by 8, then one block

08 08 08 08 08 08 08 08

is appended to the input and encrypted. After decryption, the very last byte of the plaintext is a count of the padding characters to discard.

9.5.2 Key Generation

To encrypt, you need to generate a key. Each cipher has a different format for keys, and you need to make sure that the key generation is random. Follow these steps:

1. Get a `KeyGenerator` for your algorithm.
2. Initialize the generator with a source for randomness. If the block length of the cipher is variable, also specify the desired block length.
3. Call the `generateKey` method.

For example, here is how you generate an AES key:

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecureRandom random = new SecureRandom(); // see below
keygen.init(random);
Key key = keygen.generateKey();
```

Alternatively, you can produce a key from a fixed set of raw data (perhaps derived from a password or the timing of keystrokes). Construct a `SecretKeySpec` (which implements the `SecretKey` interface) like this:

```
byte[] keyData = . . .; // 16 bytes for AES  
SecretKey key = new SecretKeySpec(keyData, "AES");
```

When generating keys, make sure you use *truly random* numbers. For example, the regular random number generator in the `Random` class, seeded by the current date and time, is not random enough. Suppose the computer clock is accurate to 1/10 of a second. Then there are at most 864,000 seeds per day. If an attacker knows the day a key was issued (which can often be deduced from a message date or certificate expiration date), it is an easy matter to generate all possible seeds for that day.

The `SecureRandom` class generates random numbers that are far more secure than those produced by the `Random` class. You still need to provide a seed to start the number sequence at a random spot. The best method for doing this is to obtain random input from a hardware device such as a white-noise generator. Another reasonable source for random input is to ask the user to type away aimlessly on the keyboard, with each keystroke contributing only one or two bits to the random seed. Once you gather such random bits in an array of bytes, pass it to the `setSeed` method:

```
SecureRandom secrand = new SecureRandom();  
byte[] b = new byte[20];  
// fill with truly random bits  
secrand.setSeed(b);
```

If you don't seed the random number generator, it will compute its own 20-byte seed by launching threads, putting them to sleep, and measuring the exact time when they are awakened.

NOTE: This algorithm is *not* known to be safe. In the past, algorithms that relied on the timing of some components of the computer, such as hard disk access time, were shown not to be completely random.

The sample program at the end of this section puts the AES cipher to work (see Listing 9.18). The `crypt` utility method in Listing 9.19 will be reused in other examples. To use the program, you first need to generate a secret key. Run

```
java aes.AESTest -genkey secret.key
```

The secret key is saved in the file `secret.key`.

Now you can encrypt with the command

```
java aes.AESTest -encrypt plaintextFile encryptedFile secret.key
```

Decrypt with the command

```
java aes.AESTest -decrypt encryptedFile decryptedFile secret.key
```

The program is straightforward. The `-genkey` option produces a new secret key and serializes it in the given file. That operation takes a long time because the initialization of the secure random generator is time-consuming. The `-encrypt` and `-decrypt` options both call into the same `crypt` method that calls the `update` and `doFinal` methods of the cipher. Note how the `update` method is called as long as the input blocks have the full length, and the `doFinal` method is either called with a partial input block (which is then padded) or with no additional data (to generate one pad block).

Listing 9.18 aes/AESTest.java

```
1 package aes;
2
3 import java.io.*;
4 import java.security.*;
5 import javax.crypto.*;
6
7 /**
8  * This program tests the AES cipher. Usage:<br>
9  * java aes.AESTest -genkey keyfile<br>
10 * java aes.AESTest -encrypt plaintext encrypted keyfile<br>
11 * java aes.AESTest -decrypt encrypted decrypted keyfile<br>
12 * @author Cay Horstmann
13 * @version 1.01 2012-06-10
14 */
15 public class AESTest
16 {
17     public static void main(String[] args)
18         throws IOException, GeneralSecurityException, ClassNotFoundException
19     {
20         if (args[0].equals("-genkey"))
21         {
22             KeyGenerator keygen = KeyGenerator.getInstance("AES");
23             SecureRandom random = new SecureRandom();
24             keygen.init(random);
25             SecretKey key = keygen.generateKey();
26             try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(args[1])))
27             {
28                 out.writeObject(key);
29             }
30         }
31         else
32         {
```

(Continues)

Listing 9.18 (Continued)

```
33     int mode;
34     if (args[0].equals("-encrypt")) mode = Cipher.ENCRYPT_MODE;
35     else mode = Cipher.DECRYPT_MODE;
36
37     try (ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(args[3]));
38          InputStream in = new FileInputStream(args[1]);
39          OutputStream out = new FileOutputStream(args[2]))
40     {
41         Key key = (Key) keyIn.readObject();
42         Cipher cipher = Cipher.getInstance("AES");
43         cipher.init(mode, key);
44         Util.crypt(in, out, cipher);
45     }
46 }
47 }
48 }
```

Listing 9.19 aes/Util.java

```
1 package aes;
2
3 import java.io.*;
4 import java.security.*;
5 import javax.crypto.*;
6
7 public class Util
8 {
9     /**
10      * Uses a cipher to transform the bytes in an input stream and sends the transformed bytes to
11      * an output stream.
12      * @param in the input stream
13      * @param out the output stream
14      * @param cipher the cipher that transforms the bytes
15     */
16    public static void crypt(InputStream in, OutputStream out, Cipher cipher)
17        throws IOException, GeneralSecurityException
18    {
19        int blockSize = cipher.getBlockSize();
20        int outputSize = cipher.getOutputSize(blockSize);
21        byte[] inBytes = new byte[blockSize];
22        byte[] outBytes = new byte[outputSize];
23
24        int inLength = 0;
25        boolean more = true;
26        while (more)
27        {
```

```
28         inLength = in.read(inBytes);
29         if (inLength == blockSize)
30         {
31             int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
32             out.write(outBytes, 0, outLength);
33         }
34     else more = false;
35 }
36 if (inLength > 0) outBytes = cipher.doFinal(inBytes, 0, inLength);
37 else outBytes = cipher.doFinal();
38 out.write(outBytes);
39 }
40 }
```

javax.crypto.Cipher 1.4

- static Cipher getInstance(String algorithmName)
- static Cipher getInstance(String algorithmName, String providerName)
returns a Cipher object that implements the specified algorithm. Throws a NoSuchAlgorithmException if the algorithm is not provided.
- int getBlockSize()
returns the size (in bytes) of a cipher block, or 0 if the cipher is not a block cipher.
- int getOutputSize(int inputLength)
returns the size of an output buffer that is needed if the next input has the given number of bytes. This method takes into account any buffered bytes in the cipher object.
- void init(int mode, Key key)
initializes the cipher algorithm object. The mode is one of ENCRYPT_MODE, DECRYPT_MODE, WRAP_MODE, or UNWRAP_MODE.
- byte[] update(byte[] in)
- byte[] update(byte[] in, int offset, int length)
- int update(byte[] in, int offset, int length, byte[] out)
transforms one block of input data. The first two methods return the output. The third method returns the number of bytes placed into out.
- byte[] doFinal()
- byte[] doFinal(byte[] in)
- byte[] doFinal(byte[] in, int offset, int length)
- int doFinal(byte[] in, int offset, int length, byte[] out)
transforms the last block of input data and flushes the buffer of this algorithm object. The first three methods return the output. The fourth method returns the number of bytes placed into out.

javax.crypto.KeyGenerator 1.4

- `static KeyGenerator getInstance(String algorithmName)`
returns a `KeyGenerator` object that implements the specified algorithm. Throws a `NoSuchAlgorithmException` if the algorithm is not provided.
- `void init(SecureRandom random)`
- `void init(int keySize, SecureRandom random)`
initializes the key generator.
- `SecretKey generateKey()`
generates a new key.

javax.crypto.spec.SecretKeySpec 1.4

- `SecretKeySpec(byte[] key, String algorithmName)`
constructs a key specification.

9.5.3 Cipher Streams

The JCE library provides a convenient set of stream classes that automatically encrypt or decrypt stream data. For example, here is how you can encrypt data to a file:

```
Cipher cipher = . . .;
cipher.init(Cipher.ENCRYPT_MODE, key);
CipherOutputStream out = new CipherOutputStream(new FileOutputStream(outputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = getData(bytes); // get data from data source
while (inLength != -1)
{
    out.write(bytes, 0, inLength);
    inLength = getData(bytes); // get more data from data source
}
out.flush();
```

Similarly, you can use a `CipherInputStream` to read and decrypt data from a file:

```
Cipher cipher = . . .;
cipher.init(Cipher.DECRYPT_MODE, key);
CipherInputStream in = new CipherInputStream(new FileInputStream(inputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = in.read(bytes);
while (inLength != -1)
{
```

```
    putData(bytes, inLength); // put data to destination  
    inLength = in.read(bytes);  
}
```

The cipher stream classes transparently handle the calls to `update` and `doFinal`, which is clearly a convenience.

javax.crypto.CipherInputStream 1.4

- `CipherInputStream(InputStream in, Cipher cipher)`
constructs an input stream that reads data from `in` and decrypts or encrypts them by using the given cipher.
- `int read()`
- `int read(byte[] b, int off, int len)`
reads data from the input stream, which is automatically decrypted or encrypted.

javax.crypto.CipherOutputStream 1.4

- `CipherOutputStream(OutputStream out, Cipher cipher)`
constructs an output stream that writes data to `out` and encrypts or decrypts them using the given cipher.
- `void write(int ch)`
- `void write(byte[] b, int off, int len)`
writes data to the output stream, which is automatically encrypted or decrypted.
- `void flush()`
flushes the cipher buffer and carries out padding if necessary.

9.5.4 Public Key Ciphers

The AES cipher that you have seen in the preceding section is a *symmetric* cipher. The same key is used for both encryption and decryption. The Achilles heel of symmetric ciphers is key distribution. If Alice sends Bob an encrypted method, Bob needs the same key that Alice used. If Alice changes the key, she needs to send Bob both the message and, through a secure channel, the new key. But perhaps she has no secure channel to Bob—which is why she encrypts her messages to him in the first place.

Public key cryptography solves that problem. In a public key cipher, Bob has a key pair consisting of a public key and a matching private key. Bob can publish

the public key anywhere, but he must closely guard the private key. Alice simply uses the public key to encrypt her messages to Bob.

Actually, it's not quite that simple. All known public key algorithms are *much* slower than symmetric key algorithms such as DES or AES. It would not be practical to use a public key algorithm to encrypt large amounts of information. However, that problem can easily be overcome by combining a public key cipher with a fast symmetric cipher, like this:

1. Alice generates a random symmetric encryption key. She uses it to encrypt her plaintext.
2. Alice encrypts the symmetric key with Bob's public key.
3. Alice sends Bob both the encrypted symmetric key and the encrypted plaintext.
4. Bob uses his private key to decrypt the symmetric key.
5. Bob uses the decrypted symmetric key to decrypt the message.

Nobody but Bob can decrypt the symmetric key because only Bob has the private key for decryption. Thus, the expensive public key encryption is only applied to a small amount of key data.

The most commonly used public key algorithm is the RSA algorithm invented by Rivest, Shamir, and Adleman. Until October 2000, the algorithm was protected by a patent assigned to RSA Security, Inc. Licenses were not cheap—typically a 3% royalty, with a minimum payment of \$50,000 per year. Now the algorithm is in the public domain.

To use the RSA algorithm, you need a public/private key pair. Use a `KeyPairGenerator` like this:

```
KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
SecureRandom random = new SecureRandom();
pairgen.initialize(KEYSIZE, random);
KeyPair keyPair = pairgen.generateKeyPair();
Key publicKey = keyPair.getPublic();
Key privateKey = keyPair.getPrivate();
```

The program in Listing 9.20 has three options. The `-genkey` option produces a key pair. The `-encrypt` option generates an AES key and *wraps* it with the public key.

```
Key key = . . .; // an AES key
Key publicKey = . . .; // a public RSA key
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.WRAP_MODE, publicKey);
byte[] wrappedKey = cipher.wrap(key);
```

It then produces a file that contains

- The length of the wrapped key
- The wrapped key bytes
- The plaintext encrypted with the AES key

The `-decrypt` option decrypts such a file. To try the program, first generate the RSA keys:

```
java rsa.RSATest -genkey public.key private.key
```

Then encrypt a file:

```
java rsa.RSATest -encrypt plaintextFile encryptedFile public.key
```

Finally, decrypt it and verify that the decrypted file matches the plaintext:

```
java rsa.RSATest -decrypt encryptedFile decryptedFile private.key
```

Listing 9.20 rsa/RSA.java

```
1 package rsa;
2
3 import java.io.*;
4 import java.security.*;
5 import javax.crypto.*;
6
7 /**
8 * This program tests the RSA cipher. Usage:<br>
9 * java rsa.RSATest -genkey public private<br>
10 * java rsa.RSATest -encrypt plaintext encrypted public<br>
11 * java rsa.RSATest -decrypt encrypted decrypted private<br>
12 * @author Cay Horstmann
13 * @version 1.01 2012-06-10
14 */
15 public class RSATest
16 {
17     private static final int KEYSIZE = 512;
18
19     public static void main(String[] args)
20         throws IOException, GeneralSecurityException, ClassNotFoundException
21     {
22         if (args[0].equals("-genkey"))
23         {
24             KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
25             SecureRandom random = new SecureRandom();
26             pairgen.initialize(KEYSIZE, random);
27             KeyPair keyPair = pairgen.generateKeyPair();
```

(Continues)

Listing 9.20 (Continued)

```
28     try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(args[1])))
29     {
30         out.writeObject(keyPair.getPublic());
31     }
32     try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(args[2])))
33     {
34         out.writeObject(keyPair.getPrivate());
35     }
36 }
37 else if (args[0].equals("-encrypt"))
38 {
39     KeyGenerator keygen = KeyGenerator.getInstance("AES");
40     SecureRandom random = new SecureRandom();
41     keygen.init(random);
42     SecretKey key = keygen.generateKey();
43
44     // wrap with RSA public key
45     try (ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(args[3]));
46          DataOutputStream out = new DataOutputStream(new FileOutputStream(args[2]));
47          InputStream in = new FileInputStream(args[1]) )
48     {
49         Key publicKey = (Key) keyIn.readObject();
50         Cipher cipher = Cipher.getInstance("RSA");
51         cipher.init(Cipher.WRAP_MODE, publicKey);
52         byte[] wrappedKey = cipher.wrap(key);
53         out.writeInt(wrappedKey.length);
54         out.write(wrappedKey);
55
56         cipher = Cipher.getInstance("AES");
57         cipher.init(Cipher.ENCRYPT_MODE, key);
58         Util.crypt(in, out, cipher);
59     }
60 }
61 else
62 {
63     try (DataInputStream in = new DataInputStream(new FileInputStream(args[1]));
64          ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(args[3]));
65          OutputStream out = new FileOutputStream(args[2]))
66     {
67         int length = in.readInt();
68         byte[] wrappedKey = new byte[length];
69         in.read(wrappedKey, 0, length);
70
71         // unwrap with RSA private key
72         Key privateKey = (Key) keyIn.readObject();
73
74         Cipher cipher = Cipher.getInstance("RSA");
```

```
75         cipher.init(Cipher.UNWRAP_MODE, privateKey);
76         Key key = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
77
78         cipher = Cipher.getInstance("AES");
79         cipher.init(Cipher.DECRYPT_MODE, key);
80
81         Util.crypt(in, out, cipher);
82     }
83 }
84 }
85 }
```

You have now seen how the Java security model allows controlled execution of code, which is a unique and increasingly important aspect of the Java platform. You have also seen the services for authentication and encryption that the Java library provides. We did not cover a number of advanced and specialized issues, among them:

- The GSS-API for “generic security services” that provides support for the Kerberos protocol (and, in principle, other protocols for secure message exchange). There is a tutorial at <http://docs.oracle.com/javase/7/docs/technotes/guides/security/jgss/tutorials>.
- Support for the Simple Authentication and Security Layer (SASL), used by the Lightweight Directory Access Protocol (LDAP) and Internet Message Access Protocol (IMAP). If you need to implement SASL in your application, look at <http://docs.oracle.com/javase/7/docs/technotes/guides/security/sasl/sasl-refguide.html>.
- Support for SSL. Using SSL over HTTP is transparent to application programmers; simply use URLs that start with https. If you want to add SSL to your application, see the Java Secure Socket Extension (JSSE) reference at <http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html>.

In the next chapter, we will delve into advanced Swing programming.

This page intentionally left blank

Advanced Swing

In this chapter

- 10.1 Lists, page 582
- 10.2 Tables, page 599
- 10.3 Trees, page 639
- 10.4 Text Components, page 681
- 10.5 Progress Indicators, page 719
- 10.6 Component Organizers and Decorators, page 731

In this chapter, we continue our discussion of the Swing user interface toolkit from Volume I. Swing is a rich toolkit, and Volume I covered only the basic and commonly used components. That leaves us with three significantly more complex components for lists, tables, and trees, the exploration of which occupies a large part of this chapter. We will then turn to text components and go beyond the simple text fields and text areas that you have seen in Volume I. We will show you how to add validations and spinners to text fields and how you can display structured text such as HTML. Next, you will see a number of components for displaying progress of a slow activity. We will finish the chapter by covering component organizers, such as tabbed panes and desktop panes with internal frames.

10.1 Lists

If you want to present a set of choices to a user, and a radio button or checkbox set consumes too much space, you can use a combo box or a list. Combo boxes were covered in Volume I because they are relatively simple. The `JList` component has many more features, and its design is similar to that of the tree and table components. For that reason, it is our starting point for the discussion of complex Swing components.

You can have lists of strings, of course, but you can also have lists of arbitrary objects, with full control of how they appear. The internal architecture of the list component that makes this generality possible is rather elegant. Unfortunately, the designers at Sun felt that they needed to show off that elegance, instead of hiding it from the programmer who just wants to use the component. You will find that the list control is somewhat awkward to use for common cases because you need to manipulate some of the machinery that makes the general cases possible. We will walk you through the simple and most common case—a list box of strings—and then give a more complex example that shows off the flexibility of the list component.

10.1.1 The `JList` Component

The `JList` component shows a number of items inside a single box. Figure 10.1 shows an admittedly silly example. The user can select the attributes for the fox, such as “quick,” “brown,” “hungry,” “wild,” and, because we ran out of attributes, “static,” “private,” and “final.” You can thus have the *private static final* fox jump over the lazy dog.

As of Java SE 7, `JList` is a generic type. The `type` parameter is the type of the values the user can select. In this example, we use a `JList<String>`.

To construct this list component, start out with an array of strings and pass that array to the `JList` constructor:

```
String[] words= { "quick", "brown", "hungry", "wild", . . . };
JList<String> wordList = new JList<>(words);
```

List boxes do not scroll automatically. To make a list box scroll, you must insert it into a scroll pane:

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

Then, add the scroll pane, not the list, into the surrounding panel.

We have to admit that the separation of the list display and the scrolling mechanism is elegant in theory, but a pain in practice. Essentially all lists that we ever



Figure 10.1 A list box

encountered needed scrolling. It seems cruel to force programmers to go through the hoops in the default case just so they can appreciate that elegance.

By default, the list component displays eight items; use the `setVisibleRowCount` method to change that value:

```
wordList.setVisibleRowCount(4); // display 4 items
```

You can set the *layout orientation* to one of three values:

- `JList.VERTICAL` (*the default*): Arrange all items vertically.
- `JList.VERTICAL_WRAP`: Start new columns if there are more items than the visible row count (see Figure 10.2).
- `JList.HORIZONTAL_WRAP`: Start new columns if there are more items than the visible row count, but fill them horizontally. Look at the placement of the words “quick,” “brown,” and “hungry” in Figure 10.2 to see the difference between vertical and horizontal wrap.

By default, a user can select multiple items. To add more items to a selection, press the Ctrl key while clicking on each item. To select a contiguous range of items, click on the first one, then hold down the Shift key and click on the last one.

You can also restrict the user to a more limited selection mode with the `setSelectionMode` method:

```
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    // select one item at a time
wordList.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
    // select one item or one range of items
```

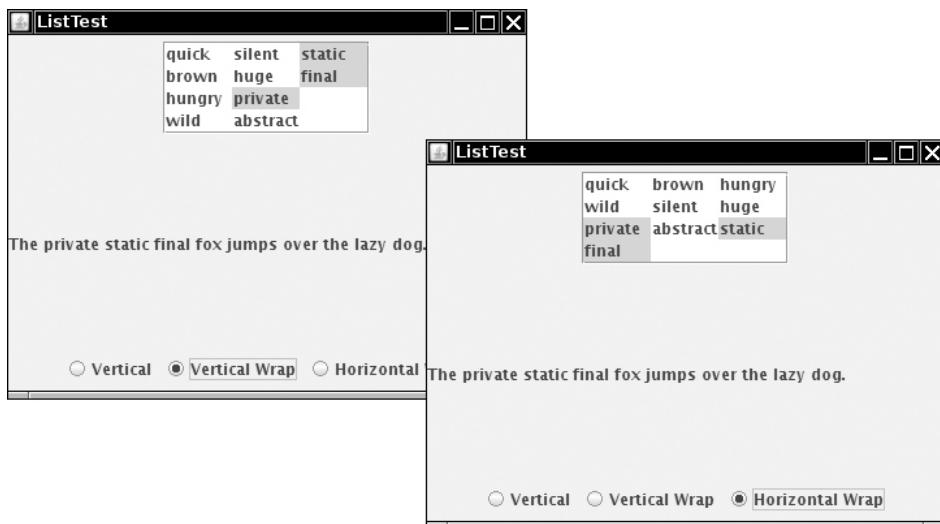


Figure 10.2 Lists with vertical and horizontal wrap

You might recall from Volume I that the basic user interface components send out action events when the user activates them. List boxes use a different notification mechanism. Instead of listening to action events, you need to listen to list selection events. Add a list selection listener to the list component, and implement the method

```
public void valueChanged(ListSelectionEvent evt)
```

in the listener.

When the user selects items, a flurry of list selection events is generated. For example, suppose the user clicks on a new item. When the mouse button goes down, an event reports a change in selection. This is a transitional event—the call

```
event.getValueIsAdjusting()
```

returns `true` if the selection is not yet final. Then, when the mouse button goes up, there is another event, this time with `getValueIsAdjusting` returning `false`. If you are not interested in the transitional events, you can wait for the event for which `getValueIsAdjusting` is `false`. However, if you want to give the user instant feedback as soon as the mouse button is clicked, you need to process all events.

Once you are notified that an event has happened, you will want to find out what items are currently selected. If your list is in single-selection mode, call `getSelectedValue` to get the value as the list element type. Otherwise, call the

`getSelectedValuesList` method which returns a list containing all selected items. You can process it in the usual way:

```
for (String value : wordList.getSelectedValuesList())
    // do something with value
```

NOTE: List components do not react to double clicks from a mouse. As envisioned by the designers of Swing, you use a list to select an item, then click a button to make something happen. However, some interfaces allow a user to double-click on a list item as a shortcut for selecting the item and invoking the default action. If you want to implement this behavior, you have to add a mouse listener to the list box, then trap the mouse event as follows:

```
public void mouseClicked(MouseEvent evt)
{
    if (evt.getClickCount() == 2)
    {
        JList source = (JList) evt.getSource();
        Object[] selection = source.getSelectedValuesList();
        doAction(selection);
    }
}
```

Listing 10.1 is the listing of the frame containing a list box filled with strings. Notice how the `valueChanged` method builds up the message string from the selected items.

Listing 10.1 list/ListFrame.java

```
1 package list;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8 * This frame contains a word list and a label that shows a sentence made up from the chosen
9 * words. Note that you can select multiple words with Ctrl+click and Shift+click.
10 */
11 class ListFrame extends JFrame
12 {
13     private static final int DEFAULT_WIDTH = 400;
14     private static final int DEFAULT_HEIGHT = 300;
15
16     private JPanel listPanel;
```

(Continues)

Listing 10.1 (Continued)

```
17  private JList<String> wordList;
18  private JLabel label;
19  private JPanel buttonPanel;
20  private ButtonGroup group;
21  private String prefix = "The ";
22  private String suffix = "fox jumps over the lazy dog.";
23
24  public ListFrame()
25  {
26      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
27
28      String[] words = { "quick", "brown", "hungry", "wild", "silent", "huge", "private",
29                         "abstract", "static", "final" };
30
31      wordList = new JList<>(words);
32      wordList.setVisibleRowCount(4);
33      JScrollPane scrollPane = new JScrollPane(wordList);
34
35      listPanel = new JPanel();
36      listPanel.add(scrollPane);
37      wordList.addListSelectionListener(event ->
38      {
39          StringBuilder text = new StringBuilder(prefix);
40          for (String value : wordList.getSelectedValuesList())
41          {
42              text.append(value);
43              text.append(" ");
44          }
45          text.append(suffix);
46
47          label.setText(text.toString());
48      });
49
50      buttonPanel = new JPanel();
51      group = new ButtonGroup();
52      makeButton("Vertical", JList.VERTICAL);
53      makeButton("Vertical Wrap", JList.VERTICAL_WRAP);
54      makeButton("Horizontal Wrap", JList.HORIZONTAL_WRAP);
55
56      add(listPanel, BorderLayout.NORTH);
57      label = new JLabel(prefix + suffix);
58      add(label, BorderLayout.CENTER);
59      add(buttonPanel, BorderLayout.SOUTH);
60  }
61
62 /**
63 * Makes a radio button to set the layout orientation.
64 * @param label the button label
```

```
65     * @param orientation the orientation for the list
66     */
67     private void makeButton(String label, final int orientation)
68     {
69         JRadioButton button = new JRadioButton(label);
70         buttonPanel.add(button);
71         if (group.getButtonCount() == 0) button.setSelected(true);
72         group.add(button);
73         button.addActionListener(event ->
74             {
75                 wordList.setLayoutOrientation(orientation);
76                 listPanel.revalidate();
77             });
78     }
79 }
```

javax.swing.JList<E> 1.2

- `JList(E[] items)`

constructs a list that displays these items.

- `int getVisibleRowCount()`
- `void setVisibleRowCount(int c)`

gets or sets the preferred number of rows in the list that can be displayed without a scroll bar.

- `int getLayoutOrientation() 1.4`
- `void setLayoutOrientation(int orientation) 1.4`

gets or sets the layout orientation.

Parameters: orientation One of VERTICAL, VERTICAL_WRAP, HORIZONTAL_WRAP

- `int getSelectionMode()`
- `void setSelectionMode(int mode)`

gets or sets the mode that determines whether single-item or multiple-item selections are allowed.

Parameters: mode One of SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION,
 MULTIPLE_INTERVAL_SELECTION

- `void addListSelectionListener(ListSelectionListener listener)`

adds to the list a listener that's notified each time a change to the selection occurs.

- `List<E> getSelectedValuesList() 7`

returns the selected values or an empty list if the selection is empty.

- `E getSelectedValue()`

returns the first selected value or null if the selection is empty.

```
javax.swing.event.ListSelectionListener 1.2
```

- void valueChanged(ListSelectionEvent e)
is called whenever the list selection changes.

10.1.2 List Models

In the preceding section, you saw the most common method for using a list component:

1. Specify a fixed set of strings for display in the list.
2. Place the list inside a scroll pane.
3. Trap the list selection events.

In the remainder of this section on lists, we cover more complex situations that require a bit more finesse:

- Very long lists
- Lists with changing contents
- Lists that don't contain strings

In the first example, we constructed a `JList` component that held a fixed collection of strings. However, the collection of choices in a list box is not always fixed. How do we add or remove items in the list box? Somewhat surprisingly, there are no methods in the `JList` class to achieve this. Instead, you have to understand a little more about the internal design of the list component. The list component uses the model-view-controller design pattern to separate the visual appearance (a column of items that are rendered in some way) from the underlying data (a collection of objects).

The `JList` class is responsible for the visual appearance of the data. It actually knows very little about how the data are stored—all it knows is that it can retrieve the data through some object that implements the `ListModel` interface:

```
public interface ListModel<E>
{
    int getSize();
    E getElementAt(int i);
    void addListDataListener(ListDataListener l);
    void removeListDataListener(ListDataListener l);
}
```

Through this interface, the `JList` can get a count of elements and retrieve any of them. Also, the `JList` object can add itself as a `ListDataListener`. That way, if the collection of elements changes, the `JList` gets notified so it can repaint itself.

Why is this generality useful? Why doesn't the `JList` object simply store an array of objects?

Note that the interface doesn't specify how the objects are stored. In particular, it doesn't force them to be stored at all! The `getElementAt` method is free to recompute each value whenever it is called. This is potentially useful if you want to show a very large collection without having to store the values.

Here is a somewhat silly example: We let the user choose among *all three-letter words* in a list box (see Figure 10.3).

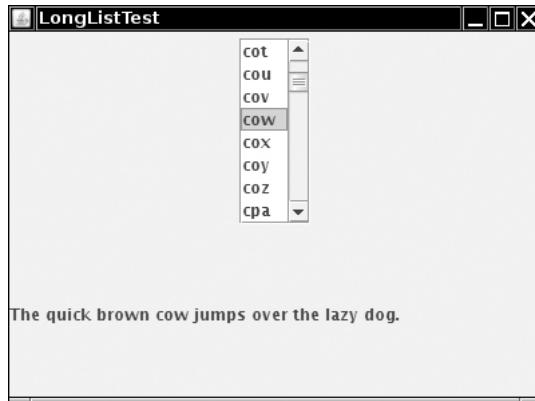


Figure 10.3 Choosing from a very long list of selections

There are $26 \times 26 \times 26 = 17,576$ three-letter combinations. Instead of storing all these combinations, we recompute them as requested when the user scrolls through them.

This turns out to be easy to implement. The tedious part, adding and removing listeners, has been done for us in the `AbstractListModel` class, which we extend. We only need to supply the `getSize` and `getElementAt` methods:

```
class WordListModel extends AbstractListModel<String>
{
    public WordListModel(int n) { length = n; }
    public int getSize() { return (int) Math.pow(26, length); }
```

```
public String getElementAt(int n)
{
    // compute nth string
    . .
}
. .
```

The computation of the *n*th string is a bit technical—you’ll find the details in Listing 10.3.

Now that we have a model, we can simply build a list that lets the user scroll through the elements supplied by the model:

```
JList<String> wordList = new JList<>(new WordListModel(3));
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane scrollPane = new JScrollPane(wordList);
```

The point is that the strings are never *stored*. Only those strings that the user actually requests to see are generated.

We must make one other setting: tell the list component that all items have a fixed width and height. The easiest way to set the cell dimensions is to specify a *prototype cell value*:

```
wordList.setPrototypeCellValue("www");
```

The prototype cell value is used to determine the size for all cells. (We use the string “www” because, in most fonts, “w” is the widest lowercase letter.)

Alternatively, you can set a fixed cell size:

```
wordList.setFixedCellWidth(50);
wordList.setFixedCellHeight(15);
```

If you don’t set a prototype value or a fixed cell size, the list component computes the width and height of each item. That can take a long time.

Listing 10.2 shows the frame class of the example program.

As a practical matter, very long lists are rarely useful. It is extremely cumbersome for a user to scroll through a huge selection. For that reason, we believe that the list control has been overengineered. A selection that can be comfortably managed on the screen is certainly small enough to be stored directly in the list component. That arrangement would have saved programmers the pain of dealing with the list model as a separate entity. On the other hand, the `JList` class is consistent with the `JTree` and `JTable` classes where this generality is useful.

Listing 10.2 longList/LongListFrame.java

```
1 package longList;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8 * This frame contains a long word list and a label that shows a sentence made up from the chosen
9 * word.
10 */
11 public class LongListFrame extends JFrame
12 {
13     private JList<String> wordList;
14     private JLabel label;
15     private String prefix = "The quick brown ";
16     private String suffix = " jumps over the lazy dog.";
17
18     public LonglistFrame()
19     {
20         wordList = new JList<String>(new WordListModel(3));
21         wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
22         wordList.setPrototypeCellValue("www");
23         JScrollPane scrollPane = new JScrollPane(wordList);
24
25         JPanel p = new JPanel();
26         p.add(scrollPane);
27         wordList.addListSelectionListener(event -> setSubject(wordList.getSelectedValue()));
28
29         Container contentPane = getContentPane();
30         contentPane.add(p, BorderLayout.NORTH);
31         label = new JLabel(prefix + suffix);
32         contentPane.add(label, BorderLayout.CENTER);
33         setSubject("fox");
34         pack();
35     }
36
37 /**
38 * Sets the subject in the label.
39 * @param word the new subject that jumps over the lazy dog
40 */
41 public void setSubject(String word)
42 {
43     StringBuilder text = new StringBuilder(prefix);
44     text.append(word);
45     text.append(suffix);
46     label.setText(text.toString());
47 }
48 }
```

Listing 10.3 longList/WordListModel.java

```
1 package longList;
2
3 import javax.swing.*;
4
5 /**
6  * A model that dynamically generates n-letter words.
7  */
8 public class WordListModel extends AbstractListModel<String>
9 {
10     private int length;
11     public static final char FIRST = 'a';
12     public static final char LAST = 'z';
13
14     /**
15      * Constructs the model.
16      * @param n the word length
17      */
18     public WordListModel(int n)
19     {
20         length = n;
21     }
22
23     public int getSize()
24     {
25         return (int) Math.pow(LAST - FIRST + 1, length);
26     }
27
28     public String getElementAt(int n)
29     {
30         StringBuilder r = new StringBuilder();
31
32         for (int i = 0; i < length; i++)
33         {
34             char c = (char) (FIRST + n % (LAST - FIRST + 1));
35             r.insert(0, c);
36             n = n / (LAST - FIRST + 1);
37         }
38         return r.toString();
39     }
40 }
```

javax.swing.JList<E> 1.2

- `JList(ListModel<E> dataModel)`
constructs a list that displays the elements in the specified model.
- `E getPrototypeCellValue()`
- `void setPrototypeCellValue(E newValue)`

gets or sets the prototype cell value used to determine the width and height of each cell in the list. The default is `null`, which forces the size of each cell to be measured.

- `void setFixedCellWidth(int width)`
- `void setFixedCellHeight(int height)`

if the width or height is greater than zero, specifies the width or height (in pixels) of every cell in the list. The default value is -1, which forces the size of each cell to be measured.

javax.swing.ListModel<E> 1.2

- `int getSize()`
returns the number of elements of the model.
- `E getElementAt(int position)`
returns an element of the model at the given position.

10.1.3 Inserting and Removing Values

You cannot directly edit the collection of list values. Instead, you must access the *model* and then add or remove elements. That, too, is easier said than done. Suppose you want to add more values to a list. You can obtain a reference to the model:

```
ListModel<String> model = list.getModel();
```

But that does you no good—as you saw in the preceding section, the `ListModel` interface has no methods to insert or remove elements because, after all, the whole point of having a list model is that it does not need to *store* the elements.

Let's try it the other way around. One of the constructors of `JList` takes a vector of objects:

```
Vector<String> values = new Vector<String>();
values.addElement("quick");
values.addElement("brown");
...
JList<String> list = new JList<>(values);
```

You can now edit the vector and add or remove elements, but the list does not know that this is happening, so it cannot react to the changes. In particular, the list does not update its view when you add the values. Therefore, this constructor is not very useful.

Instead, you should construct a `DefaultListModel` object, fill it with the initial values, and associate it with the list. The `DefaultListModel` class implements the `ListModel` interface and manages a collection of objects.

```
DefaultListModel<String> model = new DefaultListModel<>();
model.addElement("quick");
model.addElement("brown");
...
JList<String> list = new JList<>(model);
```

Now you can add or remove values from the `model` object. The `model` object then notifies the list of the changes, and the list repaints itself.

```
model.removeElement("quick");
model.addElement("slow");
```

For historical reasons, the `DefaultListModel` class doesn't use the same method names as the collection classes.

Internally, the default list model uses a vector to store the values.



CAUTION: There are `JList` constructors that construct a list from an array or vector of objects or strings. You might think that these constructors use a `DefaultListModel` to store these values. That is not the case—the constructors build a trivial model that can access the values without any provisions for notification if the content changes. For example, here is the code for the constructor that constructs a `JList` from a `Vector`:

```
public JList(final Vector<? extends E> listData)
{
    this (new AbstractListModel<E>()
    {
        public int getSize() { return listData.size(); }
        public E getElementAt(int i) { return listData.elementAt(i); }
    });
}
```

That means, if you change the contents of the vector after the list is constructed, the list might show a confusing mix of old and new values until it is completely repainted. (The keyword `final` in the preceding constructor does not prevent you from changing the vector elsewhere—it only means that the constructor itself won't modify the value of the `listData` reference; the keyword is required because the `listData` object is used in the inner class.)

`javax.swing.JList<E>` 1.2

- `ListModel<E> getModel()`
gets the model of this list.

`javax.swing.DefaultListModel<E>` 1.2

- `void addElement(E obj)`
adds the object to the end of the model.
- `boolean removeElement(Object obj)`
removes the first occurrence of the object from the model. Returns `true` if the object was contained in the model, `false` otherwise.

10.1.4 Rendering Values

So far, all lists you have seen in this chapter contained strings. It is actually just as easy to show a list of icons—simply pass an array or vector filled with `Icon` objects. More interestingly, you can easily represent your list values with any drawing whatsoever.

Although the `JList` class can display strings and icons automatically, you need to install a *list cell renderer* into the `JList` object for all custom drawing. A list cell renderer is any class that implements the following interface:

```
interface ListCellRenderer<E>
{
    Component getListCellRendererComponent(JList<? extends E> list,
                                           E value, int index, boolean isSelected, boolean cellHasFocus);
}
```

This method is called for each cell. It returns a component that paints the cell contents. The component is placed at the appropriate location whenever a cell needs to be rendered.

One way to implement a cell renderer is to create a class that extends `JComponent`, like this:

```
class MyCellRenderer extends JComponent implements ListCellRenderer<Type>
{
    public Component getListCellRendererComponent(JList<? extends Type> list,
                                                   Type value, int index, boolean isSelected, boolean cellHasFocus)
    {
        stash away information needed for painting and size measurement
        return this;
    }
    public void paintComponent(Graphics g)
    {
        paint code
    }
    public Dimension getPreferredSize()
    {
        size measurement code
    }
    instance fields
}
```

In Listing 10.4, we display the font choices graphically by showing the actual appearance of each font (see Figure 10.4). In the `paintComponent` method, we display each name in its own font. We also need to make sure to match the usual colors of the look-and-feel of the `JList` class. We obtain these colors by calling the `getForeground/getBackground` and `getSelectionForeground/getSelectionBackground` methods of the `JList` class. In the `getPreferredSize` method, we need to measure the size of the string, using the techniques that you saw in Volume I, Chapter 10.



Figure 10.4 A list box with rendered cells

To install the cell renderer, simply call the `setCellRenderer` method:

```
fontList.setCellRenderer(new FontCellRenderer());
```

Now all list cells are drawn with the custom renderer.

Actually, a simpler method for writing custom renderers works in many cases. If the rendered image just contains text, an icon, and possibly a change of color, you can get by with configuring a `JLabel`. For example, to show the font name in its own font, we can use the following renderer:

```
class FontCellRenderer extends JLabel implements ListCellRenderer<Font>
{
    public Component getListCellRendererComponent(JList<? extends Font> list,
                                                Font value, int index, boolean isSelected, boolean cellHasFocus)
    {
        Font font = (Font) value;
        setText(font.getFamily());
        setFont(font);
        setOpaque(true);
        setBackground(isSelected ? list.getSelectionBackground() : list.getBackground());
        setForeground(isSelected ? list.getSelectionForeground() : list.getForeground());
        return this;
    }
}
```

Note that here we don't write any `paintComponent` or `getPreferredSize` methods; the `JLabel` class already implements these methods to our satisfaction. All we do is configure the label appropriately by setting its text, font, and color.

This code is a convenient shortcut for those cases where an existing component—in this case, `JLabel`—already provides all functionality needed to render a cell value.

We could have used a `JLabel` in our sample program, but we gave you the more general code so you can modify it if you need to do arbitrary drawings in list cells.



CAUTION: It is not a good idea to construct a new component in each call to `getListCellRendererComponent`. If the user scrolls through many list entries, a new component would be constructed every time. Reconfiguring an existing component is safe and much more efficient.

Listing 10.4 listRendering/FontCellRenderer.java

```
1 package listRendering;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * A cell renderer for Font objects that renders the font name in its own font.
8  */
9 public class FontCellRenderer extends JComponent implements ListCellRenderer<Font>
10 {
11     private Font font;
12     private Color background;
13     private Color foreground;
14
15     public Component getListCellRendererComponent(JList<? extends Font> list,
16             Font value, int index, boolean isSelected, boolean cellHasFocus)
17     {
18         font = value;
19         background = isSelected ? list.getSelectionBackground() : list.getBackground();
20         foreground = isSelected ? list.getSelectionForeground() : list.getForeground();
21         return this;
22     }
23
24     public void paintComponent(Graphics g)
25     {
26         String text = font.getFamily();
27         FontMetrics fm = g.getFontMetrics(font);
28         g.setColor(background);
29         g.fillRect(0, 0, getWidth(), getHeight());
30         g.setColor(foreground);
31         g.setFont(font);
32         g.drawString(text, 0, fm.getAscent());
33     }
34
35     public Dimension getPreferredSize()
36     {
37         String text = font.getFamily();
38         Graphics g = getGraphics();
39         FontMetrics fm = g.getFontMetrics(font);
40         return new Dimension(fm.stringWidth(text), fm.getHeight());
41     }
42 }
```

javax.swing.JList<E> 1.2

- `Color getBackground()`
returns the background color for unselected cells.
- `Color getSelectionBackground()`
returns the background color for selected cells.
- `Color getForeground()`
returns the foreground color for unselected cells.
- `Color getSelectionForeground()`
returns the foreground color for selected cells.
- `void setCellRenderer(ListCellRenderer<? super E> cellRenderer)`
sets the renderer that paints the cells in the list.

javax.swing.ListCellRenderer<E> 1.2

- `Component getListCellRendererComponent(JList<? extends E> list, E item, int index, boolean isSelected, boolean hasFocus)`
returns a component whose `paint` method draws the cell contents. If the list cells do not have fixed size, that component must also implement `getPreferredSize`.

<i>Parameters:</i>	<code>list</code>	The list whose cell is being drawn
	<code>item</code>	The item to be drawn
	<code>index</code>	The index where the item is stored in the model
	<code>isSelected</code>	true if the specified cell was selected
	<code>hasFocus</code>	true if the specified cell has the focus

10.2 Tables

The `JTable` component displays a two-dimensional grid of objects. Tables are common in user interfaces, and the Swing team has put a lot of effort into the table control. Tables are inherently complex, but—perhaps more successfully than other Swing classes—the `JTable` component hides much of that complexity. You can produce fully functional tables with rich behavior by writing a few lines of code. You can also write more code and customize the display and behavior for your specific applications.

In the following sections, we will explain how to make simple tables, how the user interacts with them, and how to make some of the most common adjustments. As with the other complex Swing controls, it is impossible to cover all aspects in complete detail. For more information, look in *Graphic Java™, Third Edition*, by David M. Geary (Prentice Hall, 1999), or *Core Swing* by Kim Topley (Prentice Hall, 1999).

10.2.1 A Simple Table

Similar to the `JList` component, a `JTable` does not store its own data but obtains them from a *table model*. The `JTable` class has a constructor that wraps a two-dimensional array of objects into a default model. That is the strategy that we use in our first example; later in this chapter, we will turn to table models.

Figure 10.5 shows a typical table, describing the properties of the planets of the solar system. (A planet is *gaseous* if it consists mostly of hydrogen and helium. You should take the “Color” entries with a grain of salt—that column was added because it will be useful in later code examples.)

The screenshot shows a Java application window titled "TableTest". Inside the window is a `JTable` displaying data about the planets. The table has five columns: "Planet", "Radius", "Moons", "Gaseous", and "Color". The data rows are as follows:

Planet	Radius	Moons	Gaseous	Color
Mercury	2440.0	0	false	java.awt.C...
Venus	6052.0	0	false	java.awt.C...
Earth	6378.0	1	false	java.awt.C...
Mars	3397.0	2	false	java.awt.C...
Jupiter	71492.0	16	true	java.awt.C...
Saturn	60268.0	18	true	java.awt.C...
Uranus	25559.0	17	true	java.awt.C...
Neptune	171766.0	8	true	java.awt.C...

At the bottom of the window is a "Print" button.

Figure 10.5 A simple table

As you can see from the code in Listing 10.5, the data of the table is stored as a two-dimensional array of `Object` values:

```
Object[][] cells =
{
    { "Mercury", 2440.0, 0, false, Color.YELLOW },
    { "Venus", 6052.0, 0, false, Color.YELLOW },
    . . .
}
```

NOTE: Here, we take advantage of autoboxing. The entries in the second, third, and fourth columns are automatically converted into objects of type Double, Integer, and Boolean.

The table simply invokes the `toString` method on each object to display it. That's why the colors show up as `java.awt.Color[r=...,g=...,b=...]`.

Supply the column names in a separate array of strings:

```
String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color" };
```

Then, construct a table from the cell and column name arrays:

```
JTable table = new JTable(cells, columnNames);
```

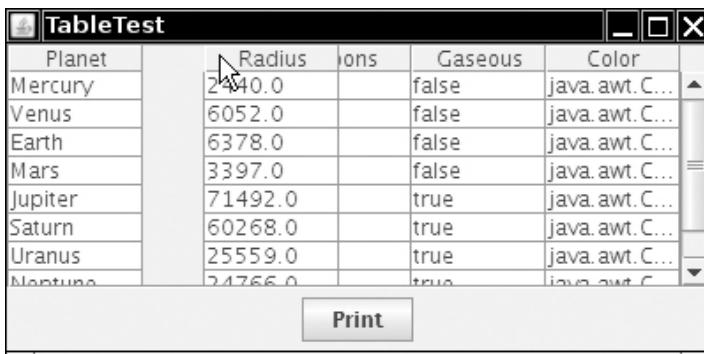
NOTE: Note that a `JTable`, unlike a `JList`, is not a generic type. There is a good reason for that. Elements in a list are expected to be of a uniform type—but, in general, there is no single element type for the entire table. In our example, the planet name is a string, the color is a `java.awt.Color`, and so on.

You can add scroll bars in the usual way—by wrapping the table in a `JScrollPane`:

```
JScrollPane pane = new JScrollPane(table);
```

When you scroll the table, the table header doesn't scroll out of view.

Next, click on one of the column headers and drag it to the left or right. See how the entire column becomes detached (see Figure 10.6). You can drop it in a different location. This rearranges the columns *in the view only*. The data model is not affected.



Planet	Radius	Moons	Gaseous	Color
Mercury	2440.0		false	java.awt.C...
Venus	6052.0		false	java.awt.C...
Earth	6378.0		false	java.awt.C...
Mars	3397.0		false	java.awt.C...
Jupiter	71492.0		true	java.awt.C...
Saturn	60268.0		true	java.awt.C...
Uranus	25559.0		true	java.awt.C...
Nepptune	24766.0		true	java.awt.C...

Figure 10.6 Moving a column

To resize columns, simply place the cursor between two columns until the cursor shape changes to an arrow. Then, drag the column boundary to the desired place (see Figure 10.7).

Planet	Radius	Moons	Gas	Color
Mercury	2440.0	0	false	java.awt.Color[r=...
Venus	6052.0	0	false	java.awt.Color[r=...
Earth	6378.0	1	false	java.awt.Color[r=...
Mars	3397.0	2	false	java.awt.Color[r=...
Jupiter	71492.0	16	true	java.awt.Color[r=...
Saturn	60268.0	18	true	java.awt.Color[r=...
Uranus	25559.0	17	true	java.awt.Color[r=...
Neptune	24766.0	8	true	java.awt.Color[r=...

Figure 10.7 Resizing columns

Users can select rows by clicking anywhere in a row. The selected rows are highlighted; you will see later how to get selection events. Users can also edit the table entries by clicking on a cell and typing into it. However, in this code example, the edits do not change the underlying data. In your programs, you should either make cells uneditable or handle cell editing events and update your model. We will discuss those topics later in this section.

Finally, click on a column header. The rows are automatically sorted. Click again, and the sort order is reversed. This behavior is activated by the call

```
table.setAutoCreateRowSorter(true);
```

You can print a table with the call

```
table.print();
```

A print dialog box appears, and the table is sent to the printer. We will discuss custom printing options in Chapter 11.

NOTE: If you resize the TableTest frame so that its height is taller than the table height, you will see a gray area below the table. Unlike JList and JTree components, the table does not fill the scroll pane's viewport. This can be a problem if you want to support drag and drop. (For more information on drag and drop, see Chapter 11.) In that case, call

```
table.setFillViewport(true);
```



CAUTION: If you don't wrap a table into a scroll pane, you need to explicitly add the header:

```
add(table.getTableHeader(), BorderLayout.NORTH);
```

Listing 10.5 table/TableTest.java

```
1 package table;
2
3 import java.awt.*;
4 import java.awt.print.*;
5
6 import javax.swing.*;
7
8 /**
9  * This program demonstrates how to show a simple table.
10 * @version 1.13 2016-05-10
11 * @author Cay Horstmann
12 */
13 public class TableTest
14 {
15     public static void main(String[] args)
16     {
17         EventQueue.invokeLater(() ->
18         {
19             JFrame frame = new PlanetTableFrame();
20             frame.setTitle("TableTest");
21             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22             frame.setVisible(true);
23         });
24     }
25 }
26
27 /**
28 * This frame contains a table of planet data.
29 */
30 class PlanetTableFrame extends JFrame
31 {
32     private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color" };
33     private Object[][] cells = { { "Mercury", 2440.0, 0, false, Color.YELLOW },
34         { "Venus", 6052.0, 0, false, Color.YELLOW }, { "Earth", 6378.0, 1, false, Color.BLUE },
35         { "Mars", 3397.0, 2, false, Color.RED }, { "Jupiter", 71492.0, 16, true, Color.ORANGE },
36         { "Saturn", 60268.0, 18, true, Color.ORANGE },
37         { "Uranus", 25559.0, 17, true, Color.BLUE }, { "Neptune", 24766.0, 8, true, Color.BLUE },
38         { "Pluto", 1137.0, 1, false, Color.BLACK } };
39 }
```

(Continues)

Listing 10.5 (Continued)

```
40     public PlanetTableFrame()
41     {
42         final JTable table = new JTable(cells, columnNames);
43         table.setAutoCreateRowSorter(true);
44         add(new JScrollPane(table), BorderLayout.CENTER);
45         JButton printButton = new JButton("Print");
46         printButton.addActionListener(event ->
47             {
48                 try { table.print(); }
49                 catch (SecurityException | PrinterException ex) { ex.printStackTrace(); }
50             });
51         JPanel buttonPanel = new JPanel();
52         buttonPanel.add(printButton);
53         add(buttonPanel, BorderLayout.SOUTH);
54         pack();
55     }
56 }
```

javax.swing.JTable 1.2

- **JTable(Object[][] entries, Object[] columnNames)**
constructs a table with a default table model.
- **void print() 5.0**
displays a print dialog box and prints the table.
- **boolean getAutoCreateRowSorter() 6**
- **void setAutoCreateRowSorter(boolean newValue) 6**
gets or sets the autoCreateRowSorter property. The default is false. When set, a default row sorter is automatically set whenever the model changes.
- **boolean getFillsViewportHeight() 6**
- **void setFillsViewportHeight(boolean newValue) 6**
gets or sets the fillsViewportHeight property. The default is false. When set, the table always fills the enclosing viewport.

10.2.2 Table Models

In the preceding example, the table data were stored in a two-dimensional array. However, you should generally not use that strategy in your own code. Instead of dumping data into an array to display it as a table, consider implementing your own table model.

Table models are particularly simple to implement because you can take advantage of the `AbstractTableModel` class that implements most of the required methods. You only need to supply three methods:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

There are many ways of implementing the `getValueAt` method. For example, if you want to display the contents of a `RowSet` that contains the result of a database query, simply provide this method:

```
public Object getValueAt(int r, int c)
{
    try
    {
        rowSet.absolute(r + 1);
        return rowSet.getObject(c + 1);
    }
    catch (SQLException e)
    {
        e.printStackTrace();
        return null;
    }
}
```

Our sample program is even simpler. We construct a table that shows some computed values—namely, the growth of an investment under different interest rate scenarios (see Figure 10.8).



The screenshot shows a Java Swing application window with a title bar. Inside, there is a `JTable` component with a header row labeled "InvestmentTable". The table has six columns representing interest rates: 5%, 6%, 7%, 8%, 9%, and 10%. The data rows show the growth of an investment starting from \$1,000,000.00. The table uses a light gray background with alternating row colors. The right side of the table includes standard scroll bars.

	5%	6%	7%	8%	9%	10%
1000000.00	1000000.00	1000000.00	1000000.00	1000000.00	1000000.00	1000000.00
1050000.00	1060000.00	1070000.00	1080000.00	1090000.00	1100000.00	
110250.00	112360.00	114490.00	116640.00	118810.00	121000.00	
115762.50	119101.60	122504.30	125971.20	129502.90	133100.00	
121550.63	126247.70	131079.60	136048.90	141158.16	146410.00	
127628.16	133822.56	140255.17	146932.81	153862.40	161051.00	
134009.56	141851.91	150073.04	158687.43	167710.01	177156.10	
140710.04	150363.03	160578.15	171382.43	182803.91	194871.71	
147745.54	159384.81	171818.62	185093.02	199256.26	214358.88	
155132.82	168947.90	183845.92	199900.46	217189.33	235794.77	
162889.46	179084.77	196715.14	215892.50	236736.37	259374.25	
171033.94	189829.86	210485.20	233163.90	258042.64	285311.67	
179585.63	201219.65	225219.16	251817.01	281266.48	313842.84	
188564.91	213292.83	240984.50	271962.37	306580.46	345227.12	
197993.16	226090.40	257853.42	293719.36	334172.70	379749.83	
207892.82	239655.82	275903.15	317216.91	364248.25	417724.82	

Figure 10.8 Growth of an investment

The `getValueAt` method computes the appropriate value and formats it:

```
public Object getValueAt(int r, int c)
{
    double rate = (c + minRate) / 100.0;
    int nperiods = r;
    double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
    return String.format("%.2f", futureBalance);
}
```

The `getRowCount` and `getColumnCount` methods simply return the number of rows and columns:

```
public int getRowCount() { return years; }
public int getColumnCount() { return maxRate - minRate + 1; }
```

If you don't supply column names, the `getColumnName` method of the `AbstractTableModel` names the columns A, B, C, and so on. To change the default column names, override the `getColumnName` method. In this example, we simply label each column with the interest rate.

```
public String getColumnName(int c) { return (c + minRate) + "%"; }
```

You can find the complete source code in Listing 10.6.

Listing 10.6 `tableModel/InvestmentTable.java`

```
1 package tableModel;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6 import javax.swing.table.*;
7
8 /**
9  * This program shows how to build a table from a table model.
10 * @version 1.03 2006-05-10
11 * @author Cay Horstmann
12 */
13 public class InvestmentTable
14 {
15     public static void main(String[] args)
16     {
17         EventQueue.invokeLater(() ->
18         {
19             JFrame frame = new InvestmentTableFrame();
20             frame.setTitle("InvestmentTable");
21             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22             frame.setVisible(true);
23         });
24     }
25 }
```

```
24     }
25 }
26
27 /**
28 * This frame contains the investment table.
29 */
30 class InvestmentTableFrame extends JFrame
31 {
32     public InvestmentTableFrame()
33     {
34         TableModel model = new InvestmentTableModel(30, 5, 10);
35         JTable table = new JTable(model);
36         add(new JScrollPane(table));
37         pack();
38     }
39
40 }
41
42 /**
43 * This table model computes the cell entries each time they are requested. The table contents
44 * shows the growth of an investment for a number of years under different interest rates.
45 */
46 class InvestmentTableModel extends AbstractTableModel
47 {
48     private static double INITIAL_BALANCE = 100000.0;
49
50     private int years;
51     private int minRate;
52     private int maxRate;
53
54 /**
55 * Constructs an investment table model.
56 * @param y the number of years
57 * @param r1 the lowest interest rate to tabulate
58 * @param r2 the highest interest rate to tabulate
59 */
60     public InvestmentTableModel(int y, int r1, int r2)
61     {
62         years = y;
63         minRate = r1;
64         maxRate = r2;
65     }
66
67     public int getRowCount()
68     {
69         return years;
70     }
71 }
```

(Continues)

Listing 10.6 (Continued)

```
72     public int getColumnCount()
73     {
74         return maxRate - minRate + 1;
75     }
76
77     public Object getValueAt(int r, int c)
78     {
79         double rate = (c + minRate) / 100.0;
80         int nperiods = r;
81         double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
82         return String.format("%.2f", futureBalance);
83     }
84
85     public String getColumnName(int c)
86     {
87         return (c + minRate) + "%";
88     }
89 }
```

javax.swing.table.TableModel 1.2

- `int getRowCount()`
- `int getColumnCount()`
gets the number of rows and columns in the table model.
- `Object getValueAt(int row, int column)`
gets the value at the given row and column.
- `void setValueAt(Object newValue, int row, int column)`
sets a new value at the given row and column.
- `boolean isCellEditable(int row, int column)`
returns true if the cell at the given row and column is editable.
- `String getColumnName(int column)`
gets the column title.

10.2.3 Working with Rows and Columns

In this subsection, you will see how to manipulate the rows and columns in a table. As you read through this material, keep in mind that a Swing table is quite asymmetric—the operations that you can carry out on rows and columns are different. The table component was optimized to display rows of information with the same structure, such as the result of a database query, not an arbitrary

two-dimensional grid of objects. You will see this asymmetry throughout this subsection.

10.2.3.1 Column Classes

In the next example, we again display our planet data, but this time we want to give the table more information about the column types. This is achieved by defining the method

```
Class<?> getColumnClass(int columnIndex)
```

of the table model to return the class that describes the column type.

The `JTable` class uses this information to pick an appropriate renderer for the class. Table 10.1 shows the default rendering actions.

You can see the checkboxes and images in Figure 10.9. (Thanks to Jim Evins, www.snaught.com/JimsCoolIcons/Planets, for providing the planet images!)

The screenshot shows a Java application window titled "TableRowColumnTest". The window has a menu bar with "Selection" and "Edit" options. A toolbar on the left contains three checkboxes: "Rows", "Columns", and "Cells". The main area is a JTable with the following data:

	Planets	Moons	Gaseous	Color	Image
	52	0	<input type="checkbox"/>	java.awt.Color[r=255,g=255,b=0]	
Earth	6,378	1	<input type="checkbox"/>	java.awt.Color[r=0,g=0,b=255]	
Mars	3,397	2	<input type="checkbox"/>	java.awt.Color[r=255,g=0,b=0]	
Jupiter	71,492	16	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	
Saturn	60,268	18	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	

Figure 10.9 A table with planet data

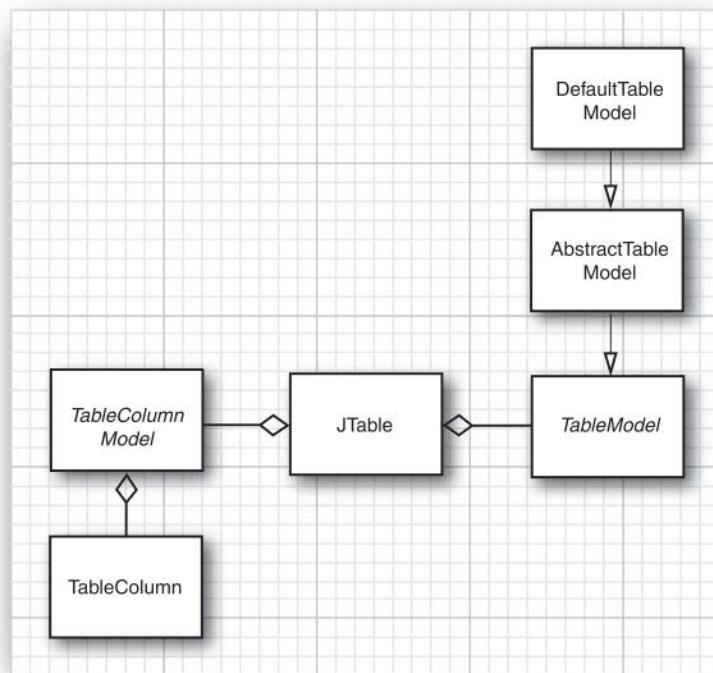
Table 10.1 Default Rendering Actions

Type	Rendered As
Boolean	Checkbox
Icon	Image
Object	String

To render other types, you can install a custom renderer—see Section 10.2.4, “Cell Rendering and Editing,” on p. 626.

10.2.3.2 Accessing Table Columns

The `JTable` class stores information about table columns in objects of type `TableColumn`. A `TableColumnModel` object manages the columns. (Figure 10.10 shows the relationships

**Figure 10.10** Relationship between table classes

among the most important table classes.) If you don't want to insert or remove columns dynamically, you won't use the column model much. The most common use for the column model is simply to get a `TableColumn` object:

```
int columnIndex = . . .;
TableColumn column = table.getColumnModel().getColumn(columnIndex);
```

10.2.3.3 Resizing Columns

The `TableColumn` class gives you control over the resizing behavior of columns. You can set the preferred, minimum, and maximum width with the methods

```
void setPreferredWidth(int width)
void setMinWidth(int width)
void setMaxWidth(int width)
```

This information is used by the table component to lay out the columns.

Use the method

```
void setResizable(boolean resizable)
```

to control whether the user is allowed to resize the column.

You can programmatically resize a column with the method

```
void setWidth(int width)
```

When a column is resized, the default is to leave the total size of the table unchanged. Of course, the width increase or decrease of the resized column must then be distributed over other columns. The default behavior is to change the size of all columns to the right of the resized column. That's a good default because it allows a user to adjust all columns to a desired width, moving from left to right.

You can set another behavior from Table 10.2 by using the method

```
void setAutoResizeMode(int mode)
```

of the `JTable` class.

Table 10.2 Resize Modes

Mode	Behavior
<code>AUTO_RESIZE_OFF</code>	Don't resize other columns; change the table width.
<code>AUTO_RESIZE_NEXT_COLUMN</code>	Resize the next column only.

(Continues)

Table 10.2 (Continued)

Mode	Behavior
AUTO_RESIZE_SUBSEQUENT_COLUMNS	Resize all subsequent columns equally; this is the default behavior.
AUTO_RESIZE_LAST_COLUMN	Resize the last column only.
AUTO_RESIZE_ALL_COLUMNS	Resize all columns in the table; this is not a good choice because it prevents the user from adjusting multiple columns to a desired size.

10.2.3.4 Resizing Rows

Row heights are managed directly by the `JTable` class. If your cells are taller than the default, you may want to set the row height:

```
table.setRowHeight(height);
```

By default, all rows of the table have the same height. You can set the heights of individual rows with the call

```
table.setRowHeight(row, height);
```

The actual row height equals the row height set with these methods, reduced by the row margin. The default row margin is 1 pixel, but you can change it with the call

```
table.setRowMargin(margin);
```

10.2.3.5 Selecting Rows, Columns, and Cells

Depending on the selection mode, the user can select rows, columns, or individual cells in the table. By default, row selection is enabled. Clicking inside a cell selects the entire row (see Figure 10.9 on p. 609). Call

```
table.setRowSelectionAllowed(false)
```

to disable row selection.

When row selection is enabled, you can control whether the user is allowed to select a single row, a contiguous set of rows, or any set of rows. You need to retrieve the *selection model* and use its `setSelectionMode` method:

```
table.getSelectionModel().setSelectionMode(mode);
```

Here, `mode` is one of the three values:

```
ListSelectionModel.SINGLE_SELECTION  
ListSelectionModel.SINGLE_INTERVAL_SELECTION  
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
```

Column selection is disabled by default. You can turn it on with the call

```
table.setColumnSelectionAllowed(true)
```

Enabling both row and column selection is equivalent to enabling cell selection. The user then selects ranges of cells (see Figure 10.11). You can also enable that setting with the call

```
table.setCellSelectionEnabled(true)
```

Planet	Radius	Moons	Gaseous	Color	Image
Mars	3,397	2	<input type="checkbox"/>	java.awt.Color[r=255,g=0,b=0]	
Jupiter	71,492	16	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	
Saturn	60,268	18	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	
Uranus	25,559	17	<input checked="" type="checkbox"/>	java.awt.Color[r=0,g=0,b=255]	

Figure 10.11 Selecting a range of cells

Run the program in Listing 10.7 to watch cell selection in action. Enable row, column, or cell selection in the Selection menu and watch how the selection behavior changes.

You can find out which rows and columns are selected by calling the `getSelectedRows` and `getSelectedColumns` methods. Both return an `int[]` array of the indexes of the

selected items. Note that the index values are those of the table view, not the underlying table model. Try selecting rows and columns, then drag columns to different places and sort the rows by clicking on column headers. Use the Print Selection menu item to see which rows and columns are reported as selected.

If you need to translate the table index values to table model index values, use the `JTable` methods `convertRowIndexToModel` and `convertColumnIndexToModel`.

10.2.3.6 Sorting Rows

As you have seen in our first table example, it is easy to add row sorting to a `JTable` simply by calling the `setAutoCreateRowSorter` method. However, to have finer-grained control over the sorting behavior, install a `TableRowSorter<M>` object into a `JTable` and customize it. The type parameter `M` denotes the table model; it needs to be a subtype of the `TableModel` interface.

```
TableRowSorter<TableModel> sorter = new TableRowSorter<TableModel>(model);
table.setRowSorter(sorter);
```

Some columns should not be sortable, such as the image column in our planet data. Turn sorting off by calling

```
sorter.setSortable(IMAGE_COLUMN, false);
```

You can install a custom comparator for each column. In our example, we will sort the colors in the Color column by preferring blue and green over red. When you click on the Color column, you will see that the blue planets go to the bottom of the table. This is achieved with the following call:

```
sorter.setComparator(COLOR_COLUMN, new Comparator<Color>()
{
    public int compare(Color c1, Color c2)
    {
        int d = c1.getBlue() - c2.getBlue();
        if (d != 0) return d;
        d = c1.getGreen() - c2.getGreen();
        if (d != 0) return d;
        return c1.getRed() - c2.getRed();
    }
});
```

If you do not specify a comparator for a column, the sort order is determined as follows:

1. If the column class is `String`, use the default collator returned by `Collator.getInstance()`. It sorts strings in a way that is appropriate for the current locale. (See Chapter 7 for more information about locales and collators.)

2. If the column class implements Comparable, use its compareTo method.
3. If a TableStringConverter has been set for the sorter, sort the strings returned by the converter's toString method with the default collator. If you want to use this approach, define a converter as follows:

```
sorter.setStringConverter(new TableStringConverter()
{
    public String toString(TableModel model, int row, int column)
    {
        Object value = model.getValueAt(row, column);
        convert value to a string and return it
    }
});
```

4. Otherwise, call the toString method on the cell values and sort them with the default collator.

10.2.3.7 Filtering Rows

In addition to sorting rows, the TableRowSorter can also selectively hide rows—a process called *filtering*. To activate filtering, set a RowFilter. For example, to include all rows that contain at least one moon, call

```
sorter.setRowFilter(RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN));
```

Here, we use a predefined number filter. To construct a number filter, supply

- The comparison type (one of EQUAL, NOT_EQUAL, AFTER, or BEFORE).
- An object of a subclass of Number (such as an Integer or Double). Only objects that have the same class as the given Number object are considered.
- Zero or more column index values. If no index values are supplied, all columns are searched.

The static RowFilter.dateFilter method constructs a date filter in the same way; you need to supply a Date object instead of the Number object.

Finally, the static RowFilter.regexFilter method constructs a filter that looks for strings matching a regular expression. For example,

```
sorter.setRowFilter(RowFilter.regexFilter(".*[s]$", PLANET_COLUMN));
```

only displays those planets whose name doesn't end with an "s". (See Chapter 2 for more information on regular expressions.)

You can also combine filters with the andFilter, orFilter, and notFilter methods. To filter for planets not ending in an "s" with at least one moon, you can use this filter combination:

```
sorter.setRowFilter( RowFilter.andFilter( Arrays.asList(
    RowFilter.regexFilter( ".*[\\s]", PLANET_COLUMN ),
    RowFilter.numberFilter( ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN ) ) );
```



CAUTION: Annoyingly, the `andFilter` and `orFilter` methods don't use variable arguments but a single parameter of type `Iterable`.

To implement your own filter, provide a subclass of `RowFilter` and implement an `include` method to indicate which rows should be displayed. This is easy to do, but the glorious generality of the `RowFilter` class makes it a bit scary.

The `RowFilter<M, I>` class has two type parameters—the types for the model and for the row identifier. When dealing with tables, the model is always a subtype of `TableModel` and the identifier type is `Integer`. (At some point in the future, other components might also support row filtering. For example, to filter rows in a `JTree`, one might use a `RowFilter<TreeModel, TreePath>`.)

A row filter must implement the method

```
public boolean include(RowFilter.Entry<? extends M, ? extends I> entry)
```

The `RowFilter.Entry` class supplies methods to obtain the model, the row identifier, and the value at a given index. Therefore, you can filter both by row identifier and by the contents of the row.

For example, this filter displays every other row:

```
RowFilter<TableModel, Integer> filter = new RowFilter<TableModel, Integer>()
{
    public boolean include(Entry<? extends TableModel, ? extends Integer> entry)
    {
        return entry.getIdentifier() % 2 == 0;
    }
};
```

If you wanted to include only those planets with an even number of moons, you would instead test for

```
((Integer) entry.getValue(MOONS_COLUMN)) % 2 == 0
```

In our sample program, we allow the user to hide arbitrary rows. We store the hidden row indexes in a set. The row filter includes all rows whose indexes are not in that set.

The filtering mechanism wasn't designed for filters with criteria changing over time. In our sample program, we keep calling

```
sorter.setRowFilter(filter);
```

whenever the set of hidden rows changes. Setting a filter causes it to be applied immediately.

10.2.3.8 Hiding and Displaying Columns

As you saw in the preceding section, you can filter table rows by either their contents or their row identifier. Hiding table columns uses a completely different mechanism.

The `removeColumn` method of the `JTable` class removes a column from the table view. The column data are not actually removed from the model—they are just hidden from view. The `removeColumn` method takes a `TableColumn` argument. If you have the column number (for example, from a call to `getSelectedColumns`), you need to ask the table model for the actual table column object:

```
TableColumnModel columnModel = table.getColumnModel();
TableColumn column = columnModel.getColumn(i);
table.removeColumn(column);
```

If you remember the column, you can later add it back in:

```
table.addColumn(column);
```

This method adds the column to the end. If you want it to appear elsewhere, call the `moveColumn` method.

You can also add a new column that corresponds to a column index in the table model, by adding a new `TableColumn` object:

```
table.addColumn(new TableColumn(modelColumnIndex));
```

You can have multiple table columns that view the same column of the model.

The program in Listing 10.7 demonstrates selection and filtering of rows and columns.

Listing 10.7 `tableRowColumn/PlanetTableFrame.java`

```
1 package tableRowColumn;
2
3 import java.awt.*;
4 import java.util.*;
5
6 import javax.swing.*;
7 import javax.swing.table.*;
8
9 /**
10 * This frame contains a table of planet data.
```

(Continues)

Listing 10.7 (*Continued*)

```
58     }
59 };
60
61 table = new JTable(model);
62
63 table.setRowHeight(100);
64 table.getColumnModel().getColumn(COLOR_COLUMN).setMinWidth(250);
65 table.getColumnModel().getColumn(IMAGE_COLUMN).setMinWidth(100);
66
67 final TableRowSorter<TableModel> sorter = new TableRowSorter<>(model);
68 table.setRowSorter(sorter);
69 sorter.setComparator(COLOR_COLUMN, Comparator.comparing(Color::getBlue)
70         .thenComparing(Color::getGreen).thenComparing(Color::getRed));
71 sorter.setSortable(IMAGE_COLUMN, false);
72 add(new JScrollPane(table), BorderLayout.CENTER);
73
74 removedRowIndices = new HashSet<>();
75 removedColumns = new ArrayList<>();
76
77 final RowFilter<TableModel, Integer> filter = new RowFilter<TableModel, Integer>()
78 {
79     public boolean include(Entry<? extends TableModel, ? extends Integer> entry)
80     {
81         return !removedRowIndices.contains(entry.getIdentifier());
82     }
83 };
84
85 // create menu
86
87 JMenuBar menuBar = new JMenuBar();
88 setJMenuBar(menuBar);
89
90 JMenu selectionMenu = new JMenu("Selection");
91 menuBar.add(selectionMenu);
92
93 rowsItem = new JCheckBoxMenuItem("Rows");
94 columnsItem = new JCheckBoxMenuItem("Columns");
95 cellsItem = new JCheckBoxMenuItem("Cells");
96
97 rowsItem.setSelected(table.getRowSelectionAllowed());
98 columnsItem.setSelected(table.getColumnSelectionAllowed());
99 cellsItem.setSelected(table.getCellSelectionEnabled());
100
101 rowsItem.addActionListener(event ->
102 {
103     table.clearSelection();
104     table.setRowSelectionAllowed(rowsItem.isSelected());
105     updateCheckboxMenuItems();
106});
```

(Continues)

Listing 10.7 (Continued)

```
107     columnsItem.addActionListener(event ->
108     {
109         table.clearSelection();
110         table.setColumnSelectionAllowed(columnsItem.isSelected());
111         updateCheckboxMenuItems();
112     });
113     selectionMenu.add(columnsItem);
114
115     cellsItem.addActionListener(event ->
116     {
117         table.clearSelection();
118         table.setCellSelectionEnabled(cellsItem.isSelected());
119         updateCheckboxMenuItems();
120     });
121     selectionMenu.add(cellsItem);
122
123     JMenu tableMenu = new JMenu("Edit");
124     menuBar.add(tableMenu);
125
126
127     JMenuItem hideColumnsItem = new JMenuItem("Hide Columns");
128     hideColumnsItem.addActionListener(event ->
129     {
130         int[] selected = table.getSelectedColumns();
131         TableColumnModel columnModel = table.getColumnModel();
132
133         // remove columns from view, starting at the last
134         // index so that column numbers aren't affected
135
136         for (int i = selected.length - 1; i >= 0; i--)
137         {
138             TableColumn column = columnModel.getColumn(selected[i]);
139             table.removeColumn(column);
140
141             // store removed columns for "show columns" command
142
143             removedColumns.add(column);
144         }
145     });
146     tableMenu.add(hideColumnsItem);
147
148     JMenuItem showColumnsItem = new JMenuItem("Show Columns");
149     showColumnsItem.addActionListener(event ->
150     {
151         // restore all removed columns
152         for (TableColumn tc : removedColumns)
153             table.addColumn(tc);
```

```
154     removedColumns.clear();
155 });
156 tableMenu.add(showColumnsItem);
157
158 JMenuItem hideRowsItem = new JMenuItem("Hide Rows");
159 hideRowsItem.addActionListener(event ->
160 {
161     int[] selected = table.getSelectedRows();
162     for (int i : selected)
163         removedRowIndices.add(table.convertRowIndexToModel(i));
164     sorter.setRowFilter(filter);
165 });
166 tableMenu.add(hideRowsItem);
167
168 JMenuItem showRowsItem = new JMenuItem("Show Rows");
169 showRowsItem.addActionListener(event ->
170 {
171     removedRowIndices.clear();
172     sorter.setRowFilter(filter);
173 });
174 tableMenu.add(showRowsItem);
175
176 JMenuItem printSelectionItem = new JMenuItem("Print Selection");
177 printSelectionItem.addActionListener(event ->
178 {
179     int[] selected = table.getSelectedRows();
180     System.out.println("Selected rows: " + Arrays.toString(selected));
181     selected = table.getSelectedColumns();
182     System.out.println("Selected columns: " + Arrays.toString(selected));
183 });
184 tableMenu.add(printSelectionItem);
185 }
186
187 private void updateCheckboxMenuItems()
188 {
189     rowsItem.setSelected(table.getRowSelectionAllowed());
190     columnsItem.setSelected(table.getColumnSelectionAllowed());
191     cellsItem.setSelected(table.getCellSelectionEnabled());
192 }
193 }
```

javax.swing.table.TableModel 1.2

- `Class getColumnClass(int columnIndex)`

gets the class for the values in this column. This information is used for sorting and rendering.

javax.swing.JTable 1.2

- `TableColumnModel getColumnModel()`
gets the “column model” that describes the arrangement of the table columns.
- `void setAutoResizeMode(int mode)`
sets the mode for automatic resizing of table columns.

Parameters: mode One of AUTO_RESIZE_OFF, AUTO_RESIZE_NEXT_COLUMN, AUTO_RESIZE_SUBSEQUENT_COLUMNS, AUTO_RESIZE_LAST_COLUMN, and AUTO_RESIZE_ALL_COLUMNS

- `int getRowMargin()`
- `void setRowMargin(int margin)`
gets or sets the amount of empty space between cells in adjacent rows.
- `int getRowHeight()`
- `void setRowHeight(int height)`
gets or sets the default height of all rows of the table.
- `int getRowHeight(int row)`
- `void setRowHeight(int row, int height)`
gets or sets the height of the given row of the table.
- `ListSelectionModel getSelectionModel()`
returns the list selection model. You need that model to choose between row, column, and cell selection.
- `boolean getRowSelectionAllowed()`
- `void setRowSelectionAllowed(boolean b)`
gets or sets the `rowSelectionAllowed` property. If true, rows are selected when the user clicks on cells.
- `boolean getColumnSelectionAllowed()`
- `void setColumnSelectionAllowed(boolean b)`
gets or sets the `columnSelectionAllowed` property. If true, columns are selected when the user clicks on cells.
- `boolean getCellSelectionEnabled()`
returns true if both `rowSelectionAllowed` and `columnSelectionAllowed` are true.
- `void setCellSelectionEnabled(boolean b)`
sets both `rowSelectionAllowed` and `columnSelectionAllowed` to b.
- `void addColumn(TableColumn column)`
adds a column as the last column of the table view.

(Continues)

javax.swing.JTable 1.2 (Continued)

- `void moveColumn(int from, int to)`
moves the column whose table index is `from` so that its index becomes `to`. Only the view is affected.
- `void removeColumn(TableColumn column)`
removes the given column from the view.
- `int convertRowIndexToModel(int index) 6`
- `int convertColumnIndexToModel(int index)`
returns the model index of the row or column with the given index. This value is different from `index` when rows are sorted or filtered, or when columns are moved or removed.
- `void setRowSorter(RowSorter<? extends TableModel> sorter)`
sets the row sorter.

javax.swing.table.TableColumnModel 1.2

- `TableColumn getColumn(int index)`
gets the table column object that describes the column with the given view index.

javax.swing.table.TableColumn 1.2

- `TableColumn(int modelColumnIndex)`
constructs a table column for viewing the model column with the given index.
- `void setPreferredWidth(int width)`
- `void setMinWidth(int width)`
- `void setMaxWidth(int width)`
sets the preferred, minimum, and maximum width of this table column to `width`.
- `void setWidth(int width)`
sets the actual width of this column to `width`.
- `void setResizable(boolean b)`
If `b` is true, this column is resizable.

javax.swing.ListSelectionModel 1.2

- void setSelectionMode(int mode)

Parameters: mode One of SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, and MULTIPLE_INTERVAL_SELECTION

javax.swing.DefaultRowSorter<M, I> 6

- void setComparator(int column, Comparator<?> comparator)
sets the comparator to be used with the given column.
- void setSortable(int column, boolean enabled)
enables or disables sorting for the given column.
- void setRowFilter(RowFilter<? super M, ? super I> filter)
sets the row filter.

javax.swing.table.TableRowSorter<M extends TableModel> 6

- void setStringConverter(TableStringConverter stringConverter)
sets the string converter used for sorting and filtering.

javax.swing.table.TableStringConverter 6

- abstract String toString(TableModel model, int row, int column)
converts the model value at the given location to a string; you can override this method.

javax.swing.RowFilter<M, I> 6

- boolean include(RowFilter.Entry<? extends M, ? extends I> entry)
specifies the rows that are retained; you can override this method.

(Continues)

javax.swing.RowFilter<M, I> 6 (Continued)

- static <M,I> RowFilter<M,I> numberFilter(RowFilter.ComparisonType type, Number number, int... indices)
- static <M,I> RowFilter<M,I> dateFilter(RowFilter.ComparisonType type, Date date, int... indices)
returns a filter that includes rows containing values that match the given comparison to the given number or date. The comparison type is one of EQUAL, NOT_EQUAL, AFTER, or BEFORE. If any column model indexes are given, only those columns are searched; otherwise, all columns are searched. For the number filter, the class of the cell value must match the class of number.
- static <M,I> RowFilter<M,I> regexFilter(String regex, int... indices)
returns a filter that includes rows that have a string value matching the given regular expression. If any column model indexes are given, only those columns are searched; otherwise, all columns are searched. Note that the string returned by the getStringValue method of RowFilter.Entry is matched.
- static <M,I> RowFilter<M,I> andFilter(Iterable<? extends RowFilter<? super M, ? super I>> filters)
- static <M,I> RowFilter<M,I> orFilter(Iterable<? extends RowFilter<? super M, ? super I>> filters)
returns a filter that includes the entries included by all filters or at least one of the filters.
- static <M,I> RowFilter<M,I> notFilter(RowFilter<M,I> filter)
returns a filter that includes the entries not included by the given filter.

javax.swing.RowFilter.Entry<M, I> 6

- I getIdentifier()
returns the identifier of this row entry.
- M getModel()
returns the model of this row entry.
- Object getValue(int index)
returns the value stored at the given index of this row.
- int getCount()
returns the number of values stored in this row.
- String getStringValue()
returns the value stored at the given index of this row, converted to a string. The TableRowSorter produces entries whose getStringValue calls the sorter's string converter.

10.2.4 Cell Rendering and Editing

As you saw in Section 10.2.3.2, “Accessing Table Columns,” on p. 610, the column type determines how the cells are rendered. There are default renderers for the types `Boolean` and `Icon` that render a checkbox or icon. For all other types, you need to install a custom renderer.

10.2.4.1 Rendering Cells

Table cell renderers are similar to the list cell renderers that you saw earlier. They implement the `TableCellRenderer` interface which has a single method:

```
Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected,
    boolean hasFocus, int row, int column)
```

That method is called when the table needs to draw a cell. You return a component whose `paint` method is then invoked to fill the cell area.

The table in Figure 10.12 contains cells of type `Color`. The renderer simply returns a panel with a background color that is the `color` object stored in the cell. The `color` is passed as the `value` parameter.

```
class ColorTableCellRenderer extends JPanel implements TableCellRenderer
{
    public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int column)
    {
        setBackground((Color) value);
        if (hasFocus)
            setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
        else
            setBorder(null);
        return this;
    }
}
```

As you can see, the renderer draws a border when the cell has focus. (We ask the `UIManager` for the correct border. To find the lookup key, we peeked into the source code of the `DefaultTableCellRenderer` class.)

Generally, you will also want to set the background color of the cell to indicate whether it is currently selected. We skip this step because it would interfere with the displayed color. The `ListRenderingTest` example in Listing 10.4 shows how to indicate the selection status in a renderer.

You need to tell the table to use this renderer with all objects of type `Color`. The `setDefaultRenderer` method of the `JTable` class lets you establish this association. Supply a `Class` object and the renderer:

Planet	Radius	Gaseous	Color	Image
Mars	3,397	2	<input type="checkbox"/>	
Jupiter	71,492	16	<input checked="" type="checkbox"/>	
Saturn	60,268	18	<input checked="" type="checkbox"/>	

Figure 10.12 A table with cell renderers

```
table.setDefaultRenderer(Color.class, new ColorTableCellRenderer());
```

That renderer is now used for all objects of the given type in this table.

If you want to select a renderer based on some other criterion, you need to subclass the `JTable` class and override the `getCellRenderer` method.



TIP: If your renderer simply draws a text string or an icon, you can extend the `DefaultTableCellRenderer` class. It takes care of rendering the focus and selection status for you.

10.2.4.2 Rendering the Header

To display an icon in the header, set the header value:

```
moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
```

However, the table header isn't smart enough to choose an appropriate renderer for the header value. You have to install the renderer manually. For example, to show an image icon in a column header, call

```
moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIcon.class));
```

10.2.4.3 Editing Cells

To enable cell editing, the table model must indicate which cells are editable by defining the `isCellEditable` method. Most commonly, you will want to make certain columns editable. In the example program, we allow editing in four columns.

```
public boolean isCellEditable(int r, int c)
{
    return c == PLANET_COLUMN || c == MOONS_COLUMN || c == GASEOUS_COLUMN || c == COLOR_COLUMN;
}
```

NOTE: The `AbstractTableModel` defines the `isCellEditable` method to always return `false`. The `DefaultTableModel` overrides the method to always return `true`.

If you run the program (Listings 10.8 to 10.11), note that you can click the checkboxes in the Gaseous column and turn the check marks on and off. If you click a cell in the Moons column, a combo box appears (see Figure 10.13). You will shortly see how to install such a combo box as a cell editor.

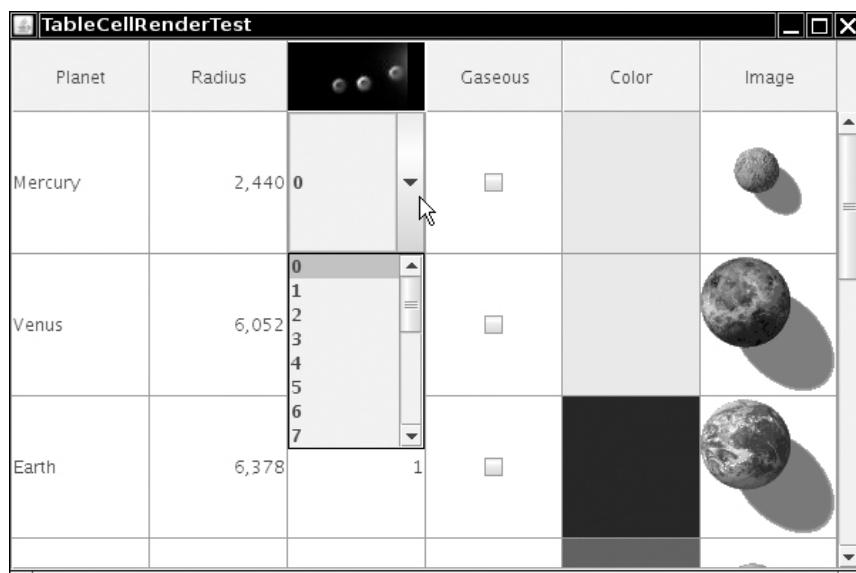


Figure 10.13 A cell editor

Finally, click a cell in the first column. The cell gains focus. You can start typing, and the cell contents change.

What you just saw in action are the three variations of the `DefaultCellEditor` class. A `DefaultCellEditor` can be constructed with a `JTextField`, a `JCheckBox`, or a `JComboBox`. The `JTable` class automatically installs a checkbox editor for `Boolean` cells and a text field editor for all editable cells that don't supply their own renderer. The text fields let the user edit the strings that result from applying `toString` to the return value of the `getValueAt` method of the table model.

When the edit is complete, the edited value is retrieved by calling the `getCellEditorValue` method of your editor. That method should return a value of the correct type (that is, the type returned by the `getColumnType` method of the model).

To get a combo box editor, set a cell editor manually—the `JTable` component has no idea what values might be appropriate for a particular type. For the Moons column, we wanted to enable the user to pick any value between 0 and 20. Here is the code for initializing the combo box:

```
JComboBox moonCombo = new JComboBox();
for (int i = 0; i <= 20; i++)
    moonCombo.addItem(i);
```

To construct a `DefaultCellEditor`, supply the combo box in the constructor:

```
TableCellEditor moonEditor = new DefaultCellEditor(moonCombo);
```

Next, we need to install the editor. Unlike the color cell renderer, this editor does not depend on the object *type*—we don't necessarily want to use it for all objects of type `Integer`. Instead, we need to install it into a particular column:

```
moonColumn.setCellEditor(moonEditor);
```

10.2.4.4 Custom Editors

Run the example program again and click a color. A *color chooser* pops up and lets you pick a new color for the planet. Select a color and click OK. The cell color is updated (see Figure 10.14).

The color cell editor is not a standard table cell editor but a custom implementation. To create a custom cell editor, implement the `TableCellEditor` interface. That interface is a bit tedious, and as of Java SE 1.3, an `AbstractCellEditor` class is provided to take care of the event handling details.

The `getTableCellEditorComponent` method of the `TableCellEditor` interface requests a component to render the cell. It is exactly the same as the `getTableCellRendererComponent` method of the `TableCellRenderer` interface, except that there is no `focus` parameter. When the cell is being edited, it is presumed to have focus. The editor component temporarily *replaces* the renderer when the editing is in progress. In our example, we return a blank panel that is not colored. This is an indication to the user that the cell is currently being edited.

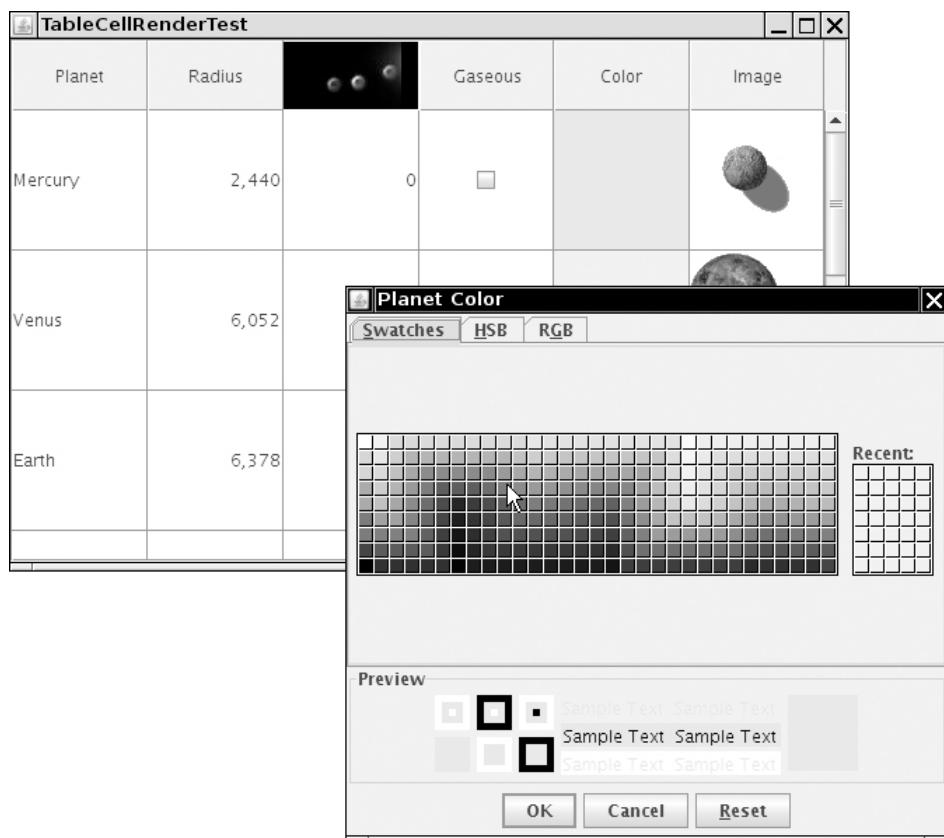


Figure 10.14 Editing the cell color with a color chooser

Next, you want to have your editor pop up when the user clicks on the cell.

The `JTable` class calls your editor with an event (such as a mouse click) to find out if that event is acceptable to initiate the editing process. The `AbstractCellEditor` class defines the method to accept all events.

```
public boolean isCellEditable(EventObject anEvent)
{
    return true;
}
```

However, if you override this method to return `false`, the table would not go through the trouble of inserting the editor component.

Once the editor component is installed, the `shouldSelectCell` method is called, presumably with the same event. You should initiate editing in this method—for example, by popping up an external edit dialog box.

```
public boolean shouldSelectCell(EventObject anEvent)
{
    colorDialog.setVisible(true);
    return true;
}
```

If the user cancels the edit, the table calls the `cancelCellEditing` method. If the user has clicked on another table cell, the table calls the `stopCellEditing` method. In both cases, you should hide the dialog box. When your `stopCellEditing` method is called, the table would like to use the partially edited value. You should return `true` if the current value is valid. In the color chooser, any value is valid. But if you edit other data, you can ensure that only valid data are retrieved from the editor.

Also, you should call the superclass methods that take care of event firing—otherwise, the editing won't be properly canceled.

```
public void cancelCellEditing()
{
    colorDialog.setVisible(false);
    super.cancelCellEditing();
}
```

Finally, you need a method that yields the value that the user supplied in the editing process:

```
public Object getCellEditorValue()
{
    return colorChooser.getColor();
}
```

To summarize, your custom editor should do the following:

1. Extend the `AbstractCellEditor` class and implement the `TableCellEditor` interface.
2. Define the `getTableCellEditorComponent` method to supply a component. This can either be a dummy component (if you pop up a dialog box) or a component for in-place editing such as a combo box or text field.
3. Define the `shouldSelectCell`, `stopCellEditing`, and `cancelCellEditing` methods to handle the start, completion, and cancellation of the editing process. The `stopCellEditing` and `cancelCellEditing` methods should call the superclass methods to ensure that listeners are notified.
4. Define the `getCellEditorValue` method to return the value that is the result of the editing process.

Finally, indicate when the user is finished editing by calling the `stopCellEditing` and `cancelCellEditing` methods. When constructing the color dialog box, we install the `accept` and `cancel` callbacks that fire these events.

```
colorDialog = JColorChooser.createDialog(null, "Planet Color", false, colorChooser,
    EventHandler.create(ActionListener.class, this, "stopCellEditing"),
    EventHandler.create(ActionListener.class, this, "cancelCellEditing"));
```

This completes the implementation of the custom editor.

You now know how to make a cell editable and how to install an editor. There is one remaining issue—how to update the model with the value that the user edited. When editing is complete, the `JTable` class calls the following method of the table model:

```
void setValueAt(Object value, int r, int c)
```

You need to override the method to store the new value. The `value` parameter is the object that was returned by the cell editor. If you implemented the cell editor, you know the type of the object you return from the `getCellEditorValue` method. In the case of the `DefaultCellEditor`, there are three possibilities for that value. It is a `Boolean` if the cell editor is a checkbox, a string if it is a text field, and, if the value comes from a combo box, it is the object that the user selected.

If the `value` object does not have the appropriate type, you need to convert it. That happens most commonly when a number is edited in a text field. In our example, we populated the combo box with `Integer` objects so that no conversion is necessary.

Listing 10.8 `tableCellRender/TableCellRenderFrame.java`

```
1 package tableCellRender;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import javax.swing.table.*;
6
7 /**
8  * This frame contains a table of planet data.
9 */
10 public class TableCellRenderFrame extends JFrame
11 {
12     private static final int DEFAULT_WIDTH = 600;
13     private static final int DEFAULT_HEIGHT = 400;
14
15     public TableCellRenderFrame()
16     {
```

```
17    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);  
18  
19    TableModel model = new PlanetTableModel();  
20    JTable table = new JTable(model);  
21    table.setRowSelectionAllowed(false);  
22  
23    // set up renderers and editors  
24  
25    table.setDefaultRenderer(Color.class, new ColorTableCellRenderer());  
26    table.setDefaultEditor(Color.class, new ColorTableCellEditor());  
27  
28    JComboBox<Integer> moonCombo = new JComboBox<>();  
29    for (int i = 0; i <= 20; i++)  
30        moonCombo.addItem(i);  
31  
32    TableColumnModel columnModel = table.getColumnModel();  
33    TableColumn moonColumn = columnModel.getColumn(PlanetTableModel.MOONS_COLUMN);  
34    moonColumn.setCellEditor(new DefaultCellEditor(moonCombo));  
35    moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIcon.class));  
36    moonColumn.setHeaderValue(new ImageIcon(getClass().getResource("Moons.gif")));  
37  
38    // show table  
39  
40    table.setRowHeight(100);  
41    add(new JScrollPane(table), BorderLayout.CENTER);  
42 }  
43 }
```

Listing 10.9 tableCellRender/PlanetTableModel.java

```
1 package tableCellRender;  
2  
3 import java.awt.*;  
4 import javax.swing.*;  
5 import javax.swing.table.*;  
6  
7 /**  
8  * The planet table model specifies the values, rendering and editing properties for the planet  
9  * data.  
10 */  
11 public class PlanetTableModel extends AbstractTableModel  
12 {  
13     public static final int PLANET_COLUMN = 0;  
14     public static final int MOONS_COLUMN = 2;  
15     public static final int GASEOUS_COLUMN = 3;  
16     public static final int COLOR_COLUMN = 4;  
17 }
```

(Continues)

Listing 10.9 (Continued)

```
18  private Object[][] cells = {  
19      { "Mercury", 2440.0, 0, false, Color.YELLOW,  
20          new ImageIcon(getClass().getResource("Mercury.gif")) },  
21      { "Venus", 6052.0, 0, false, Color.YELLOW,  
22          new ImageIcon(getClass().getResource("Venus.gif")) },  
23      { "Earth", 6378.0, 1, false, Color.BLUE,  
24          new ImageIcon(getClass().getResource("Earth.gif")) },  
25      { "Mars", 3397.0, 2, false, Color.RED,  
26          new ImageIcon(getClass().getResource("Mars.gif")) },  
27      { "Jupiter", 71492.0, 16, true, Color.ORANGE,  
28          new ImageIcon(getClass().getResource("Jupiter.gif")) },  
29      { "Saturn", 60268.0, 18, true, Color.ORANGE,  
30          new ImageIcon(getClass().getResource("Saturn.gif")) },  
31      { "Uranus", 25559.0, 17, true, Color.BLUE,  
32          new ImageIcon(getClass().getResource("Uranus.gif")) },  
33      { "Neptune", 24766.0, 8, true, Color.BLUE,  
34          new ImageIcon(getClass().getResource("Neptune.gif")) },  
35      { "Pluto", 1137.0, 1, false, Color.BLACK,  
36          new ImageIcon(getClass().getResource("Pluto.gif")) } };  
37  
38  private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color", "Image" };  
39  
40  public String getColumnName(int c)  
41  {  
42      return columnNames[c];  
43  }  
44  
45  public Class<?> getColumnClass(int c)  
46  {  
47      return cells[0][c].getClass();  
48  }  
49  
50  public int getColumnCount()  
51  {  
52      return cells[0].length;  
53  }  
54  
55  public int getRowCount()  
56  {  
57      return cells.length;  
58  }  
59  
60  public Object getValueAt(int r, int c)  
61  {  
62      return cells[r][c];  
63  }  
64
```

```
65     public void setValueAt(Object obj, int r, int c)
66     {
67         cells[r][c] = obj;
68     }
69
70     public boolean isCellEditable(int r, int c)
71     {
72         return c == PLANET_COLUMN || c == MOONS_COLUMN || c == GASEOUS_COLUMN || c == COLOR_COLUMN;
73     }
74 }
```

Listing 10.10 tableCellRender/ColorTableCellRenderer.java

```
1 package tableCellRender;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import javax.swing.table.*;
6
7 /**
8  * This renderer renders a color value as a panel with the given color.
9  */
10 public class ColorTableCellRenderer extends JPanel implements TableCellRenderer
11 {
12     public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected,
13             boolean hasFocus, int row, int column)
14     {
15         setBackground((Color) value);
16         if (hasFocus) setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
17         else setBorder(null);
18         return this;
19     }
20 }
```

Listing 10.11 tableCellRender/ColorTableCellEditor.java

```
1 package tableCellRender;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.beans.*;
6 import java.util.*;
7 import javax.swing.*;
8 import javax.swing.table.*;
9
10 /**
11  * This editor pops up a color dialog to edit a cell value.
12  */
```

(Continues)

Listing 10.11 (Continued)

```
13 public class ColorTableCellEditor extends AbstractCellEditor implements TableCellEditor
14 {
15     private JColorChooser colorChooser;
16     private JDialog colorDialog;
17     private JPanel panel;
18
19     public ColorTableCellEditor()
20     {
21         panel = new JPanel();
22         // prepare color dialog
23
24         colorChooser = new JColorChooser();
25         colorDialog = JColorChooser.createDialog(null, "Planet Color", false, colorChooser,
26             EventHandler.create(ActionListener.class, this, "stopCellEditing"),
27             EventHandler.create(ActionListener.class, this, "cancelCellEditing"));
28     }
29
30     public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected,
31         int row, int column)
32     {
33         // this is where we get the current Color value. We store it in the dialog in case the user
34         // starts editing
35         colorChooser.setColor((Color) value);
36         return panel;
37     }
38
39     public boolean shouldSelectCell(EventObject anEvent)
40     {
41         // start editing
42         colorDialog.setVisible(true);
43
44         // tell caller it is ok to select this cell
45         return true;
46     }
47
48     public void cancelCellEditing()
49     {
50         // editing is canceled--hide dialog
51         colorDialog.setVisible(false);
52         super.cancelCellEditing();
53     }
54
55     public boolean stopCellEditing()
56     {
57         // editing is complete--hide dialog
58         colorDialog.setVisible(false);
59         super.stopCellEditing();
```

```
60      // tell caller it is ok to use color value
61      return true;
62  }
63
64  public Object getCellEditorValue()
65  {
66      return colorChooser.getColor();
67  }
68
69 }
```

javax.swing.JTable 1.2

- **TableCellRenderer getDefaultRenderer(Class<?> type)**
gets the default renderer for the given type.
- **TableCellEditor getDefaultEditor(Class<?> type)**
gets the default editor for the given type.

javax.swing.table.TableCellRenderer 1.2

- **Component getTableCellRendererComponent(JTable table, Object value, boolean selected, boolean hasFocus, int row, int column)**

returns a component whose paint method is invoked to render a table cell.

Parameters:

table	The table containing the cell to be rendered
value	The cell to be rendered
selected	true if the cell is currently selected
hasFocus	true if the cell currently has focus
row, column	The row and column of the cell

javax.swing.table.TableColumn 1.2

- **void setCellEditor(TableCellEditor editor)**
- **void setCellRenderer(TableCellRenderer renderer)**
sets the cell editor or renderer for all cells in this column.
- **void setHeaderRenderer(TableCellRenderer renderer)**
sets the cell renderer for the header cell in this column.
- **void setHeaderValue(Object value)**
sets the value to be displayed for the header in this column.

javax.swing.DefaultCellEditor 1.2

- `DefaultCellEditor(JComboBox comboBox)`

constructs a cell editor that presents the combo box for selecting cell values.

javax.swing.table.TableCellEditor 1.2

- `Component getTableCellEditorComponent(JTable table, Object value, boolean selected, int row, int column)`

returns a component whose `paint` method renders a table cell.

Parameters: `table` The table containing the cell to be rendered
 `value` The cell to be rendered
 `selected` true if the cell is currently selected
 `row, column` The row and column of the cell

javax.swing.CellEditor 1.2

- `boolean isCellEditable(EventObject event)`

returns true if the event is suitable for initiating the editing process for this cell.

- `boolean shouldSelectCell(EventObject anEvent)`

starts the editing process. Returns true if the edited cell should be *selected*. Normally, you want to return true, but you can return false if you don't want the editing process to change the cell selection.

- `void cancelCellEditing()`

cancels the editing process. You can abandon partial edits.

- `boolean stopCellEditing()`

stops the editing process, with the intent of using the result. Returns true if the edited value is in a proper state for retrieval.

- `Object getCellEditorValue()`

returns the edited result.

- `void addCellEditorListener(CellEditorListener l)`

- `void removeCellEditorListener(CellEditorListener l)`

adds or removes the obligatory cell editor listener.

10.3 Trees

Every computer user who has worked with a hierarchical file system has seen tree displays. Of course, directories and files form only one of the many examples of tree-like organizations. Many tree structures arise in everyday life, such as the hierarchy of countries, states, and cities shown in Figure 10.15.

As programmers, we often need to display tree structures. Fortunately, the Swing library has a `JTree` class for this purpose. The `JTree` class (together with its helper classes) takes care of laying out the tree and processing user requests for expanding and collapsing nodes. In this section, you will learn how to put the `JTree` class to use.

As with the other complex Swing components, we must focus on the common and useful cases and cannot cover every nuance. If you want to achieve something unusual, we recommend that you consult *Graphic Java™, Third Edition*, by David M. Geary or *Core Swing* by Kim Topley.

Before going any further, let's settle on some terminology (see Figure 10.16). A *tree* is composed of *nodes*. Every node is either a *leaf* or it has *child nodes*. Every node, with the exception of the root node, has exactly one *parent*. A tree has exactly one root node. Sometimes you have a collection of trees, each with its own root node. Such a collection is called a *forest*.

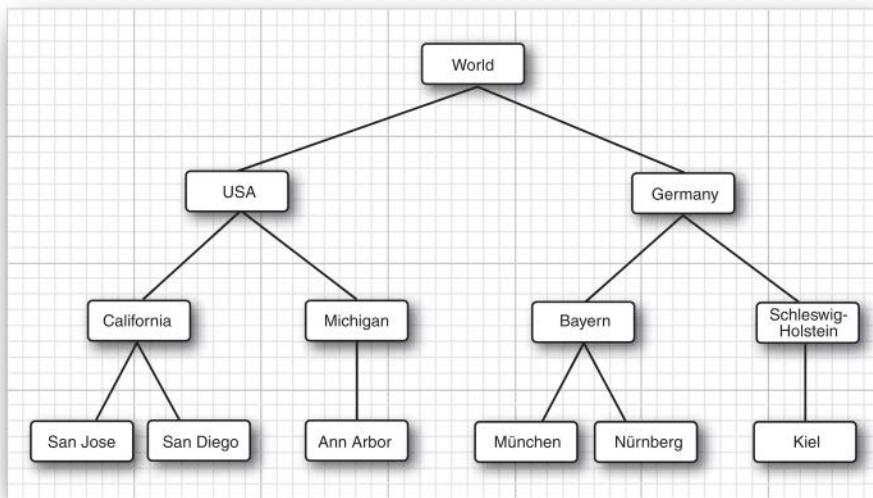


Figure 10.15 A hierarchy of countries, states, and cities

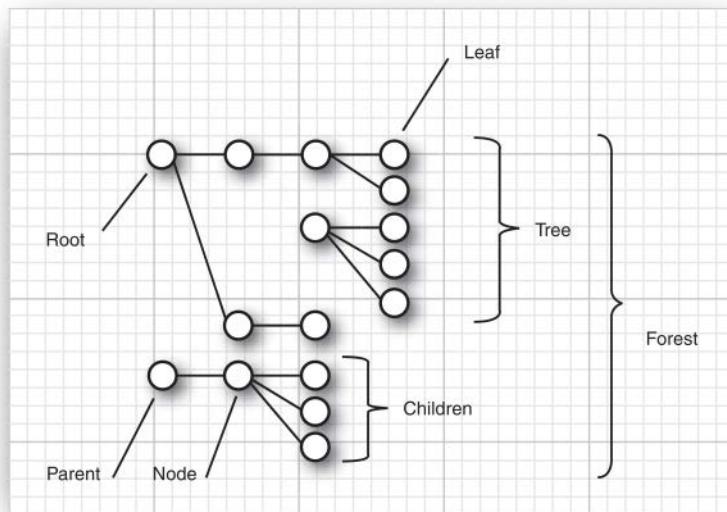


Figure 10.16 Tree terminology

10.3.1 Simple Trees

In our first example program, we will simply display a tree with a few nodes (see Figure 10.18 on p. 642). As with many other Swing components, you need to provide a model of the data, and the component displays it for you. To construct a `JTree`, supply the tree model in the constructor:

```
TreeModel model = . . .;  
JTree tree = new JTree(model);
```

NOTE: There are also constructors that construct trees out of a collection of elements:

```
JTree(Object[] nodes)  
JTree(Vector<?> nodes)  
JTree(Hashtable<?, ?> nodes) // the values become the nodes
```

These constructors are not very useful. They merely build a forest of trees, each with a single node. The third constructor seems particularly useless because the nodes appear in the seemingly random order determined by the hash codes of the keys.

How do you obtain a tree model? You can construct your own model by creating a class that implements the `TreeModel` interface. You will see later in this chapter how to do that. For now, we will stick with the `DefaultTreeModel` that the Swing library supplies.

To construct a default tree model, you must supply a root node.

```
TreeNode root = . . .;
DefaultTreeModel model = new DefaultTreeModel(root);
```

`TreeNode` is another interface. Populate the default tree model with objects of any class that implements the interface. For now, we will use the concrete node class that Swing supplies—namely, `DefaultMutableTreeNode`. This class implements the `MutableTreeNode` interface, a subinterface of `TreeNode` (see Figure 10.17).

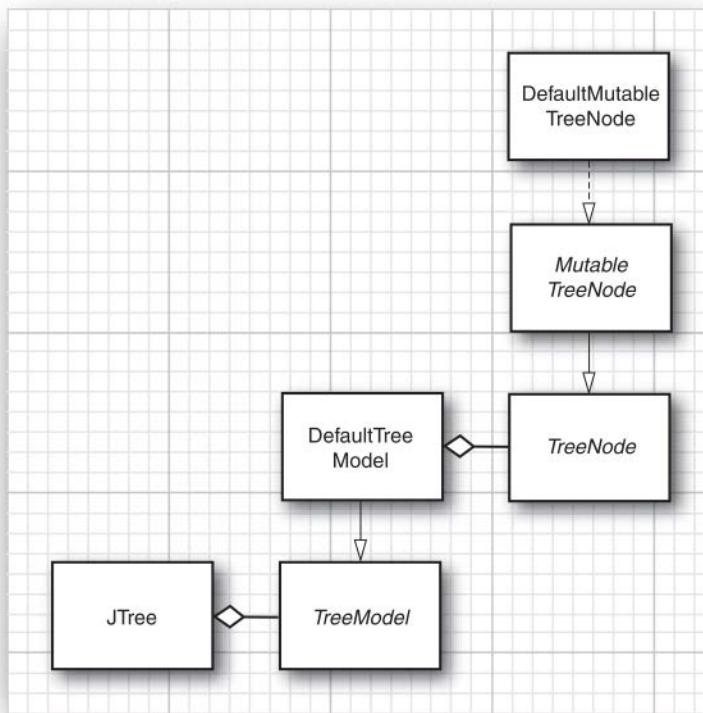


Figure 10.17 Tree classes

A default mutable tree node holds an object—the *user object*. The tree renders the user objects for all nodes. Unless you specify a renderer, the tree displays the string that is the result of the `toString` method.

In our first example, we use strings as user objects. In practice, you would usually populate a tree with more expressive user objects. For example, when displaying a directory tree, it makes sense to use `File` objects for the nodes.

You can specify the user object in the constructor, or you can set it later with the `setUserObject` method.

```
DefaultMutableTreeNode node = new DefaultMutableTreeNode("Texas");
...
node.setUserObject("California");
```

Next, you need to establish the parent/child relationships between the nodes. Start with the root node and use the `add` method to add the children:

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
root.add(country);
DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
country.add(state);
```

Figure 10.18 illustrates how the tree will look.



Figure 10.18 A simple tree

Link up all nodes in this fashion. Then, construct a `DefaultTreeModel` with the root node. Finally, construct a `JTree` with the tree model.

```
DefaultTreeModel treeModel = new DefaultTreeModel(root);
JTree tree = new JTree(treeModel);
```

Or, as a shortcut, you can simply pass the root node to the `JTree` constructor. Then the tree automatically constructs a default tree model:

```
JTree tree = new JTree(root);
```

Listing 10.12 contains the complete code.

Listing 10.12 `tree/SimpleTreeFrame.java`

```
1 package tree;
2
3 import javax.swing.*;
4 import javax.swing.tree.*;
5
6 /**
7  * This frame contains a simple tree that displays a manually constructed tree model.
8 */
9 public class SimpleTreeFrame extends JFrame
10 {
11     private static final int DEFAULT_WIDTH = 300;
12     private static final int DEFAULT_HEIGHT = 200;
13
14     public SimpleTreeFrame()
15     {
16         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
17
18         // set up tree model data
19
20         DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
21         DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
22         root.add(country);
23         DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
24         country.add(state);
25         DefaultMutableTreeNode city = new DefaultMutableTreeNode("San Jose");
26         state.add(city);
27         city = new DefaultMutableTreeNode("Cupertino");
28         state.add(city);
29         state = new DefaultMutableTreeNode("Michigan");
30         country.add(state);
```

(Continues)

Listing 10.12 (Continued)

```
31     city = new DefaultMutableTreeNode("Ann Arbor");
32     state.add(city);
33     country = new DefaultMutableTreeNode("Germany");
34     root.add(country);
35     state = new DefaultMutableTreeNode("Schleswig-Holstein");
36     country.add(state);
37     city = new DefaultMutableTreeNode("Kiel");
38     state.add(city);
39
40     // construct tree and put it in a scroll pane
41
42     JTree tree = new JTree(root);
43     add(new JScrollPane(tree));
44 }
45 }
```

When you run the program, the tree first looks as in Figure 10.19. Only the root node and its children are visible. Click on the circle icons (the *handles*) to open up the subtrees. The line sticking out from the handle icon points to the right when the subtree is collapsed and down when the subtree is expanded (see Figure 10.20). We don't know what the designers of the Metal look-and-feel had in mind, but we think of the icon as a door handle. You push down on the handle to open the subtree.



Figure 10.19 The initial tree display

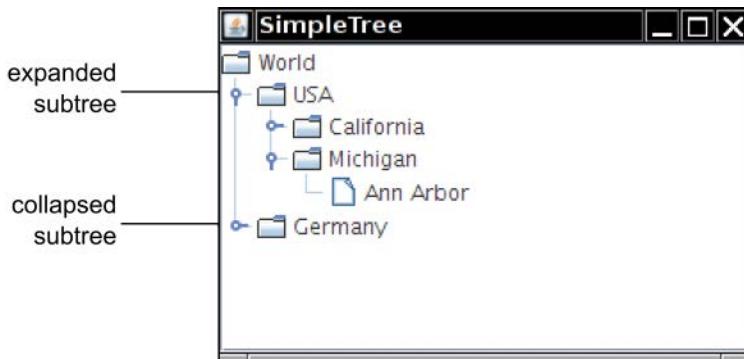


Figure 10.20 Collapsed and expanded subtrees

NOTE: Of course, the display of the tree depends on the selected look-and-feel. We just described the Metal look-and-feel. In the Windows look-and-feel, the handles have the more familiar look—a “-” or “+” in a box (see Figure 10.21).



Figure 10.21 A tree with the Windows look-and-feel

You can use the following magic incantation to turn off the lines joining parents and children (see Figure 10.22):

```
tree.putClientProperty("JTree.lineStyle", "None");
```



Figure 10.22 A tree with no connecting lines

Conversely, to make sure that the lines are shown, use

```
tree.putClientProperty("JTree.lineStyle", "Angled");
```

Another line style, "Horizontal", is shown in Figure 10.23. The tree is displayed with horizontal lines separating only the children of the root. We aren't quite sure what it is good for.



Figure 10.23 A tree with the horizontal line style

By default, there is no handle for collapsing the root of the tree. If you like, you can add one with the call

```
tree.setShowsRootHandles(true);
```

Figure 10.24 shows the result. Now you can collapse the entire tree into the root node.



Figure 10.24 A tree with a root handle

Conversely, you can hide the root altogether. You will thus display a *forest*—a set of trees, each with its own root. You still must join all trees in the forest to a common root; then, hide the root with the instruction

```
tree.setRootVisible(false);
```

Look at Figure 10.25. There appear to be two roots, labeled “USA” and “Germany.” The actual root that joins the two is made invisible.



Figure 10.25 A forest

Let's turn from the root to the leaves of the tree. Note that the leaves have an icon different from the other nodes (see Figure 10.26).

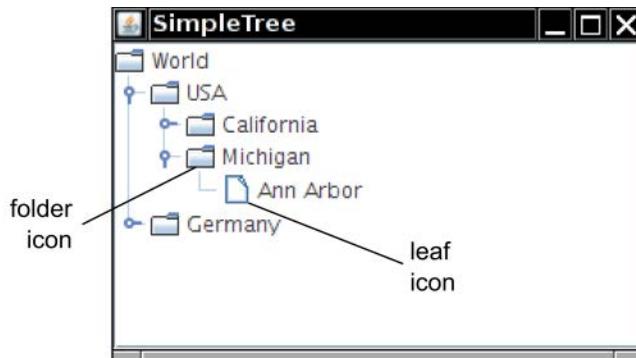


Figure 10.26 Leaf and folder icons

When the tree is displayed, each node is drawn with an icon. There are actually three kinds of icons: a leaf icon, an opened nonleaf icon, and a closed nonleaf icon. For simplicity, we refer to the last two as folder icons.

The node renderer needs to know which icon to use for each node. By default, the decision process works like this: If the `isLeaf` method of a node returns `true`, then the leaf icon is used; otherwise, a folder icon is used.

The `isLeaf` method of the `DefaultMutableTreeNode` class returns `true` if the node has no children. Thus, nodes with children get folder icons, and nodes without children get leaf icons.

Sometimes, that behavior is not appropriate. Suppose we added a node “Montana” to our sample tree, but we’re at a loss as to what cities to add. We would not want the state node to get a leaf icon because conceptually only the cities are leaves.

The `JTree` class has no idea which nodes should be leaves. It asks the tree model. If a childless node isn’t automatically a conceptual leaf, you can ask the tree model to use a different criterion for leafiness—namely, to query the “allows children” node property.

For those nodes that should not have children, call

```
node.setAllowsChildren(false);
```

Then, tell the tree model to ask the value of the “allows children” property to determine whether a node should be displayed with a leaf icon. Use the `setAsksAllowsChildren` method of the `DefaultTreeModel` class to set this behavior:

```
model.setAsksAllowsChildren(true);
```

With this decision criterion, nodes that allow children get folder icons, and nodes that don’t allow children get leaf icons.

Alternatively, if you construct the tree from the root node, supply the setting for the “asks allows children” property in the constructor.

```
JTree tree = new JTree(root, true); // nodes that don't allow children get leaf icons
```

javax.swing.JTree 1.2

- `JTree(TreeModel model)`
constructs a tree from a tree model.
- `JTree(TreeNode root)`
- `JTree(TreeNode root, boolean asksAllowsChildren)`
constructs a tree with a default tree model that displays the root and its children.
Parameters: `root` The root node
 `asksAllowsChildren` true to use the “allows children” node property
 for determining whether a node is a leaf
- `void setShowsRootHandles(boolean b)`
if `b` is true, the root node has a handle for collapsing or expanding its children.
- `void setRootVisible(boolean b)`
if `b` is true, then the root node is displayed. Otherwise, it is hidden.

javax.swing.tree.TreeNode 1.2

- `boolean isLeaf()`
returns true if this node is conceptually a leaf.
- `boolean getAllowsChildren()`
returns true if this node can have child nodes.

javax.swing.tree.MutableTreeNode 1.2

- `void setUserObject(Object userObject)`
sets the “user object” that the tree node uses for rendering.

javax.swing.tree.TreeModel 1.2

- `boolean isLeaf(Object node)`
returns true if node should be displayed as a leaf node.

javax.swing.tree.DefaultTreeModel 1.2

- void setAsksAllowsChildren(boolean b)

if b is true, nodes are displayed as leaves when their `getAllowsChildren` method returns false. Otherwise, they are displayed as leaves when their `isLeaf` method returns true.

javax.swing.tree.DefaultMutableTreeNode 1.2

- `DefaultMutableTreeNode(Object userObject)`

constructs a mutable tree node with the given user object.

- void add(MutableTreeNode child)

adds a node as the last child of this node.

- void setAllowsChildren(boolean b)

if b is true, children can be added to this node.

javax.swing.JComponent 1.2

- void putClientProperty(Object key, Object value)

adds a key/value pair to a small table that each component manages. This is an “escape hatch” mechanism that some Swing components use for storing properties specific to a look-and-feel.

10.3.2 Editing Trees and Tree Paths

In the next example program, you will see how to edit a tree. Figure 10.27 shows the user interface. If you click the Add Sibling or Add Child button, the program adds a new node (with title New) to the tree. If you click the Delete button, the program deletes the currently selected node.

To implement this behavior, you need to find out which tree node is currently selected. The `JTree` class has a surprising way of identifying nodes in a tree. It does not deal with tree nodes but with *paths of objects*, called *tree paths*. A tree path starts at the root and consists of a sequence of child nodes (see Figure 10.28).

You might wonder why the `JTree` class needs the whole path. Couldn’t it just get a `TreeNode` and keep calling the `getParent` method? In fact, the `JTree` class knows nothing about the `TreeNode` interface. That interface is never used by the `TreeModel` interface; it is only used by the `DefaultTreeModel` implementation. You can have other tree models in which the nodes do not implement the `TreeNode` interface at all. If



Figure 10.27 Editing a tree

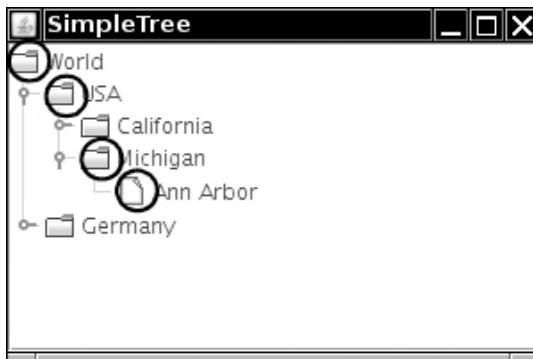


Figure 10.28 A tree path

you use a tree model that manages other types of objects, those objects might not have `getParent` and `getChild` methods. They would of course need to have some other connection to each other. It is the job of the tree model to link nodes together. The `JTree` class itself has no clue about the nature of their linkage. For that reason, the `JTree` class always needs to work with complete paths.

The `TreePath` class manages a sequence of `Object` (not `TreeNode!`) references. A number of `JTree` methods return `TreePath` objects. When you have a tree path, you usually just need to know the terminal node, which you can get with the `getLastPathComponent` method. For example, to find out the currently selected node in a tree, use the `getSelectionPath` method of the `JTree` class. You will get a `TreePath` object back, from which you can retrieve the actual node.

```
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode
    = (DefaultMutableTreeNode) selectionPath.getLastPathComponent();
```

Actually, since this particular query is so common, there is a convenience method that gives the selected node immediately:

```
DefaultMutableTreeNode selectedNode  
= (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
```

This method is not called `getSelectedNode` because the tree does not know that it contains nodes—its tree model deals only with paths of objects.

NOTE: Tree paths are one of the two ways in which the `JTree` class describes nodes. Quite a few `JTree` methods take or return an integer index—the *row position*. A row position is simply the row number (starting with 0) of the node in the tree display. Only visible nodes have row numbers, and the row number of a node changes if other nodes before it are expanded, collapsed, or modified. For that reason, you should avoid row positions. All `JTree` methods that use rows have equivalents that use tree paths instead.

Once you have the selected node, you can edit it. However, do not simply add children to a tree node:

```
selectedNode.add(newNode); // No!
```

If you change the structure of the nodes, you change the model but the associated view is not notified. You could send out a notification yourself, but if you use the `insertNodeInto` method of the `DefaultTreeModel` class, the model class takes care of that. For example, the following call appends a new node as the last child of the selected node and notifies the tree view:

```
model.insertNodeInto(newNode, selectedNode, selectedNode.getChildCount());
```

The analogous call `removeNodeFromParent` removes a node and notifies the view:

```
model.removeNodeFromParent(selectedNode);
```

If you keep the node structure in place but change the user object, you should call the following method:

```
model.nodeChanged(changedNode);
```

The automatic notification is a major advantage of using the `DefaultTreeModel`. If you supply your own tree model, you have to implement automatic notification by hand. (See *Core Swing* by Kim Topley for details.)



CAUTION: The `DefaultTreeModel` class has a `reload` method that reloads the entire model. However, don't call `reload` simply to update the tree after making a few changes. When the tree is regenerated, all nodes beyond the root's children are collapsed again. It will be quite disconcerting to your users if they have to keep expanding the tree after every change.

When the view is notified of a change in the node structure, it updates the display but does not automatically expand a node to show newly added children. In particular, if a user in our sample program adds a new child node to a node for which children are currently collapsed, the new node is silently added to the collapsed subtree. This gives the user no feedback that the command was actually carried out. In such a case, you should make a special effort to expand all parent nodes so that the newly added node becomes visible. Use the `makeVisible` method of the `JTree` class for this purpose. The `makeVisible` method expects a tree path leading to the node that should become visible.

Thus, you need to construct a tree path from the root to the newly inserted node. To get a tree path, first call the `getPathToRoot` method of the `DefaultTreeModel` class. It returns a `TreeNode[]` array of all nodes from a node to the root node. Pass that array to a `TreePath` constructor.

For example, here is how you make the new node visible:

```
TreeNode[] nodes = model.getPathToRoot(newNode);
TreePath path = new TreePath(nodes);
tree.makeVisible(path);
```

NOTE: It is curious that the `DefaultTreeModel` class feigns almost complete ignorance of the `TreePath` class, even though its job is to communicate with a `JTree`. The `JTree` class uses tree paths a lot, and it never uses arrays of node objects.

But now suppose your tree is contained inside a scroll pane. After the tree node expansion, the new node might still not be visible because it falls outside the viewport. To overcome that problem, call

```
tree.scrollPathToVisible(path);
```

instead of calling `makeVisible`. This call expands all nodes along the path and tells the ambient scroll pane to scroll the node at the end of the path into view (see Figure 10.29).



Figure 10.29 The scroll pane scrolls to display a new node.

By default, tree nodes cannot be edited. However, if you call

```
tree.setEditable(true);
```

the user can edit a node simply by double-clicking, editing the string, and pressing the Enter key. Double-clicking invokes the *default cell editor*, which is implemented by the DefaultCellEditor class (see Figure 10.30). It is possible to install other cell editors, using the same process that you have seen in our discussion of table cell editors.



Figure 10.30 The default cell editor

Listing 10.13 shows the complete source code of the tree editing program. Run the program, add a few nodes, and edit them by double-clicking them. Observe how collapsed nodes expand to show added children and how the scroll pane keeps added nodes in the viewport.

Listing 10.13 treeEdit/TreeEditFrame.java

```
1 package treeEdit;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6 import javax.swing.tree.*;
7
8 /**
9  * A frame with a tree and buttons to edit the tree.
10 */
11 public class TreeEditFrame extends JFrame
12 {
13     private static final int DEFAULT_WIDTH = 400;
14     private static final int DEFAULT_HEIGHT = 200;
15
16     private DefaultTreeModel model;
17     private JTree tree;
18
19     public TreeEditFrame()
20     {
21         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
22
23         // construct tree
24
25         TreeNode root = makeSampleTree();
26         model = new DefaultTreeModel(root);
27         tree = new JTree(model);
28         tree.setEditable(true);
29
30         // add scroll pane with tree
31
32         JScrollPane scrollPane = new JScrollPane(tree);
33         add(scrollPane, BorderLayout.CENTER);
34
35         makeButtons();
36     }
37
38     public TreeNode makeSampleTree()
39     {
40         DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
41         DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
42         root.add(country);
43         DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
44         country.add(state);
45         DefaultMutableTreeNode city = new DefaultMutableTreeNode("San Jose");
46         state.add(city);

```

(Continues)

Listing 10.13 (Continued)

```
47     city = new DefaultMutableTreeNode("San Diego");
48     state.add(city);
49     state = new DefaultMutableTreeNode("Michigan");
50     country.add(state);
51     city = new DefaultMutableTreeNode("Ann Arbor");
52     state.add(city);
53     country = new DefaultMutableTreeNode("Germany");
54     root.add(country);
55     state = new DefaultMutableTreeNode("Schleswig-Holstein");
56     country.add(state);
57     city = new DefaultMutableTreeNode("Kiel");
58     state.add(city);
59     return root;
60 }
61 /**
62 * Makes the buttons to add a sibling, add a child, and delete a node.
63 */
64 public void makeButtons()
65 {
66     JPanel panel = new JPanel();
67     JButton addSiblingButton = new JButton("Add Sibling");
68     addSiblingButton.addActionListener(event ->
69     {
70         DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree
71             .getLastSelectedPathComponent();
72
73         if (selectedNode == null) return;
74
75         DefaultMutableTreeNode parent = (DefaultMutableTreeNode) selectedNode.getParent();
76
77         if (parent == null) return;
78
79         DefaultMutableTreeNode newNode = new DefaultMutableTreeNode("New");
80
81         int selectedIndex = parent.getIndex(selectedNode);
82         model.insertNodeInto(newNode, parent, selectedIndex + 1);
83
84         // now display new node
85
86         TreeNode[] nodes = model.getPathToRoot(newNode);
87         TreePath path = new TreePath(nodes);
88         tree.scrollPathToVisible(path);
89     });
90     panel.add(addSiblingButton);
91
92     JButton addChildButton = new JButton("Add Child");
93 }
```

```
94     addChildButton.addActionListener(event ->
95     {
96         DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree
97             .getLastSelectedPathComponent();
98
99         if (selectedNode == null) return;
100
101        DefaultMutableTreeNode newNode = new DefaultMutableTreeNode("New");
102        model.insertNodeInto(newNode, selectedNode, selectedNode.getChildCount());
103
104        // now display new node
105
106        TreeNode[] nodes = model.getPathToRoot(newNode);
107        TreePath path = new TreePath(nodes);
108        tree.scrollPathToVisible(path);
109    });
110    panel.add(addChildButton);
111
112    JButton deleteButton = new JButton("Delete");
113    deleteButton.addActionListener(event ->
114    {
115        DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree
116            .getLastSelectedPathComponent();
117
118        if (selectedNode != null && selectedNode.getParent() != null) model
119            .removeNodeFromParent(selectedNode);
120    });
121    panel.add(deleteButton);
122    add(panel, BorderLayout.SOUTH);
123 }
124 }
```

javax.swing.JTree 1.2

- `TreePath getSelectionPath()`

gets the path to the currently selected node, or the path to the first selected node if multiple nodes are selected. Returns `null` if no node is selected.

- `Object getLastSelectedPathComponent()`

gets the node object that represents the currently selected node, or the first node if multiple nodes are selected. Returns `null` if no node is selected.

- `void makeVisible(TreePath path)`

expands all nodes along the path.

- `void scrollPathToVisible(TreePath path)`

expands all nodes along the path and, if the tree is contained in a scroll pane, scrolls to ensure that the last node on the path is visible.

javax.swing.tree.TreePath 1.2

- `Object getLastPathComponent()`

gets the last object on this path—that is, the node object that the path represents.

javax.swing.tree.TreeNode 1.2

- `TreeNode getParent()`

returns the parent node of this node.

- `TreeNode getChildAt(int index)`

looks up the child node at the given index. The index must be between 0 and `getChildCount() - 1`.

- `int getChildCount()`

returns the number of children of this node.

- `Enumeration children()`

returns an enumeration object that iterates through all children of this node.

javax.swing.tree.DefaultTreeModel 1.2

- `void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)`

inserts `newChild` as a new child node of `parent` at the given index and notifies the tree model listeners.

- `void removeNodeFromParent(MutableTreeNode node)`

removes `node` from this model and notifies the tree model listeners.

- `void nodeChanged(TreeNode node)`

notifies the tree model listeners that `node` has changed.

- `void nodesChanged(TreeNode parent, int[] changedChildIndexes)`

notifies the tree model listeners that all child nodes of `parent` with the given indexes have changed.

- `void reload()`

reloads all nodes into the model. This is a drastic operation that you should use only if the nodes have changed completely because of some outside influence.

10.3.3 Node Enumeration

Sometimes you need to find a node in a tree by starting at the root and visiting all children until you have found a match. The `DefaultMutableTreeNode` class has several convenience methods for iterating through nodes.

The `breadthFirstEnumeration` and `depthFirstEnumeration` methods return enumeration objects whose `nextElement` method visits all children of the current node, using either a breadth-first or depth-first traversal. Figure 10.31 shows the traversals for a sample tree—the node labels indicate the order in which the nodes are traversed.

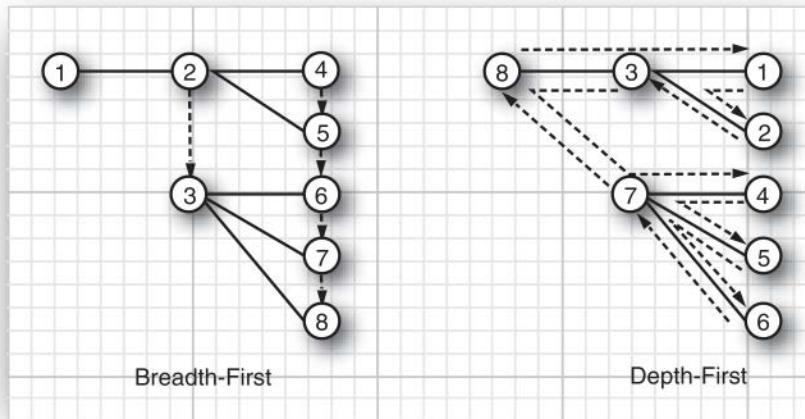


Figure 10.31 Tree traversal orders

Breadth-first enumeration is the easiest to visualize. The tree is traversed in layers. The root is visited first, followed by all of its children, then the grandchildren, and so on.

To visualize depth-first enumeration, imagine a rat trapped in a tree-shaped maze. It rushes along the first path until it comes to a leaf. Then, it backtracks and turns around to the next path, and so on.

Computer scientists also call this *postorder traversal* because the search process visits the children before visiting the parents. The `postOrderEnumeration` method is a synonym for `depthFirstEnumeration`. For completeness, there is also a `preOrderEnumeration`, a depth-first search that enumerates parents before the children.

Here is the typical usage pattern:

```
Enumeration breadthFirst = node.breadthFirstEnumeration();
while (breadthFirst.hasMoreElements())
    do something with breadthFirst.nextElement();
```

Finally, a related method, `pathFromAncestorEnumeration`, finds a path from an ancestor to a given node and enumerates the nodes along that path. That's no big deal—it just keeps calling `getParent` until the ancestor is found and then presents the path in reverse order.

In our next example program, we put node enumeration to work. The program displays inheritance trees of classes. Type the name of a class into the text field on the bottom of the frame. The class and all of its superclasses are added to the tree (see Figure 10.32).

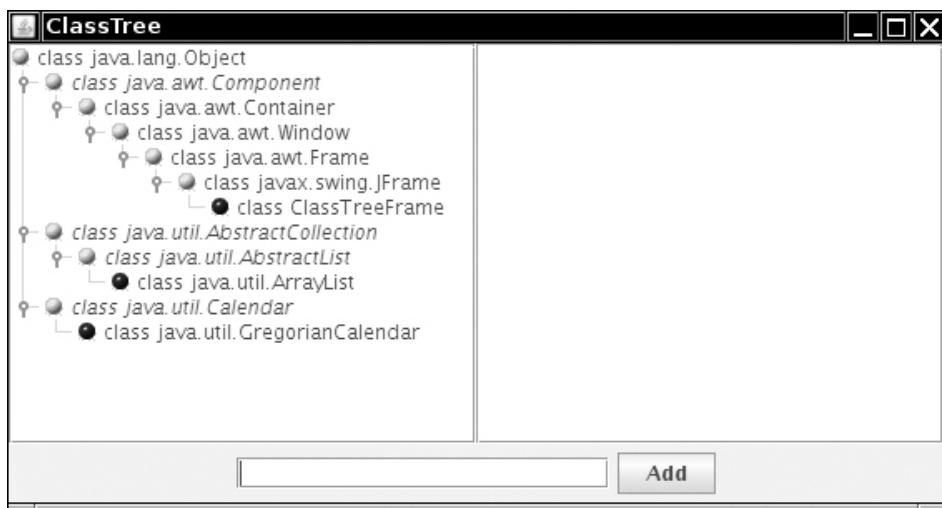


Figure 10.32 An inheritance tree

In this example, we take advantage of the fact that the user objects of the tree nodes can be objects of any type. Since our nodes describe classes, we store `Class` objects in the nodes.

We don't want to add the same class object twice, so we need to check whether a class already exists in the tree. The following method finds the node with a given user object if it exists in the tree.

```
public DefaultMutableTreeNode findUserObject(Object obj)
{
    Enumeration e = root.breadthFirstEnumeration();
    while (e.hasMoreElements())
    {
```

```
DefaultMutableTreeNode node = (DefaultMutableTreeNode) e.nextElement();
if (node.getUserObject().equals(obj))
    return node;
}
return null;
}
```

10.3.4 Rendering Nodes

In your applications, you will often need to change the way a tree component draws the nodes. The most common change is, of course, to choose different icons for nodes and leaves. Other changes might involve changing the font of the node labels or drawing images at the nodes. All these changes are possible by installing a new *tree cell renderer* into the tree. By default, the `JTree` class uses `DefaultTreeCellRenderer` objects to draw each node. The `DefaultTreeCellRenderer` class extends the `JLabel` class. The label contains the node icon and the node label.

NOTE: The cell renderer does not draw the “handles” for expanding and collapsing subtrees. The handles are part of the look-and-feel, and it is recommended that you do not change them.

You can customize the display in three ways.

- You can change the icons, font, and background color used by a `DefaultTreeCellRenderer`. These settings are used for all nodes in the tree.
- You can install a renderer that extends the `DefaultTreeCellRenderer` class and vary the icons, fonts, and background color for each node.
- You can install a renderer that implements the `TreeCellRenderer` interface to draw a custom image for each node.

Let us look at these possibilities one by one. The easiest customization is to construct a `DefaultTreeCellRenderer` object, change the icons, and install it into the tree:

```
DefaultTreeCellRenderer renderer = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif")); // used for leaf nodes
renderer.setClosedIcon(new ImageIcon("red-ball.gif")); // used for collapsed nodes
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif")); // used for expanded nodes
tree.setCellRenderer(renderer);
```

You can see the effect in Figure 10.32. We just use the “ball” icons as placeholders—presumably your user interface designer would supply you with appropriate icons to use for your applications.

We don’t recommend that you change the font or background color for an entire tree—that is really the job of the look-and-feel.

However, it can be useful to change the font of some nodes in a tree to highlight them. If you look carefully at Figure 10.32, you will notice that the *abstract* classes are set in italics.

To change the appearance of individual nodes, install a tree cell renderer. Tree cell renderers are very similar to the list cell renderers we discussed earlier in this chapter. The `TreeCellRenderer` interface has a single method:

```
Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected,
    boolean expanded, boolean leaf, int row, boolean hasFocus)
```

The `getTreeCellRendererComponent` method of the `DefaultTreeCellRenderer` class returns this—in other words, a label. (The `DefaultTreeCellRenderer` class extends the `JLabel` class.) To customize the component, extend the `DefaultTreeCellRenderer` class. Override the `getTreeCellRendererComponent` method as follows: Call the superclass method so it can prepare the label data, customize the label properties, and finally return this.

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer
{
    public Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected,
        boolean expanded, boolean leaf, int row, boolean hasFocus)
    {
        Component comp = super.getTreeCellRendererComponent(tree, value, selected,
            expanded, leaf, row, hasFocus);
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
        look at node.getUserObject();
        Font font = appropriate font;
        comp.setFont(font);
        return comp;
    }
};
```



CAUTION: The `value` parameter of the `getTreeCellRendererComponent` method is the *node* object, *not* the user object! Recall that the user object is a feature of the `DefaultMutableTreeNode`, and that a `JTree` can contain nodes of an arbitrary type. If your tree uses `DefaultMutableTreeNode` nodes, you must retrieve the user object in a second step, as we did in the preceding code sample.



CAUTION: The `DefaultTreeCellRenderer` uses the *same* label object for all nodes, only changing the label text for each node. If you change the font for a particular node, you must set it back to its default value when the method is called again. Otherwise, all subsequent nodes will be drawn in the changed font! Look at the code in Listing 10.14 to see how to restore the font to the default.

We do not show an example of a tree cell renderer that draws arbitrary graphics. If you need this capability, you can adapt the list cell renderer in Listing 10.4; the technique is entirely analogous.

The `ClassNameTreeCellRenderer` in Listing 10.14 sets the class name in either the normal or italic font, depending on the `ABSTRACT` modifier of the `Class` object. We don't want to set a particular font because we don't want to change whatever font the look-and-feel normally uses for labels. For that reason, we use the font from the label and *derive* an italic font from it. Recall that only a single shared `JLabel` object is returned by all calls. We need to hang on to the original font and restore it in the next call to the `getTreeCellRendererComponent` method.

Also, note how we change the node icons in the `ClassTreeFrame` constructor.

`javax.swing.tree.DefaultMutableTreeNode 1.2`

- `Enumeration breadthFirstEnumeration()`
- `Enumeration depthFirstEnumeration()`
- `Enumeration preOrderEnumeration()`
- `Enumeration postOrderEnumeration()`

returns enumeration objects for visiting all nodes of the tree model in a particular order. In breadth-first traversal, children that are closer to the root are visited before those that are farther away. In depth-first traversal, all children of a node are completely enumerated before its siblings are visited. The `postOrderEnumeration` method is a synonym for `depthFirstEnumeration`. The preorder traversal is identical to the postorder traversal except that parents are enumerated before their children.

`javax.swing.tree.TreeCellRenderer 1.2`

- `Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)`

returns a component whose `paint` method is invoked to render a tree cell.

<i>Parameters:</i>	<code>tree</code>	The tree containing the node to be rendered
	<code>value</code>	The node to be rendered
	<code>selected</code>	true if the node is currently selected
	<code>expanded</code>	true if the children of the node are visible
	<code>leaf</code>	true if the node needs to be displayed as a leaf
	<code>row</code>	The display row containing the node
	<code>hasFocus</code>	true if the node currently has input focus

javax.swing.tree.DefaultTreeCellRenderer 1.2

- void setLeafIcon(Icon icon)
- void setOpenIcon(Icon icon)
- void setClosedIcon(Icon icon)

sets the icon to show for a leaf node, an expanded node, and a collapsed node.

10.3.5 Listening to Tree Events

Most commonly, a tree component is paired with some other component. When the user selects tree nodes, some information shows up in another window. See Figure 10.33 for an example. When the user selects a class, the instance and static variables of that class are displayed in the text area to the right.

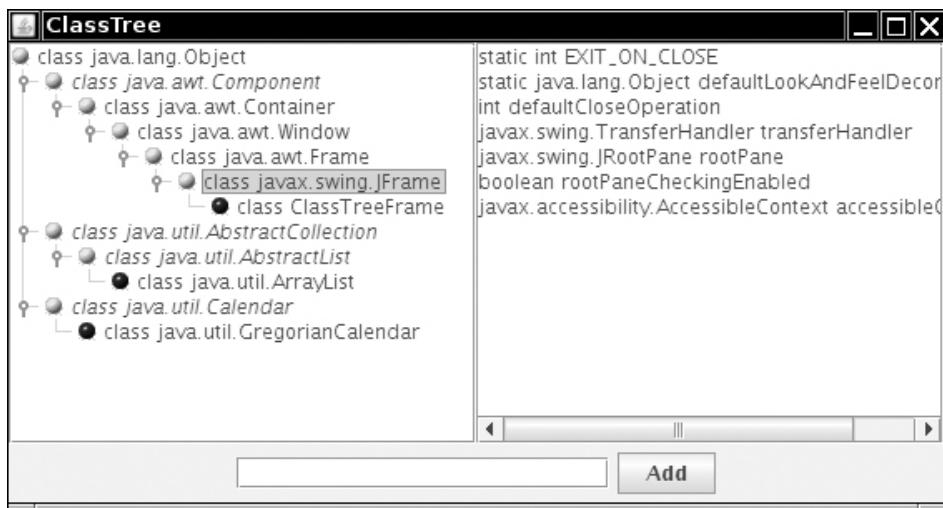


Figure 10.33 A class browser

To obtain this behavior, you need to install a *tree selection listener*. The listener must implement the `TreeSelectionListener` interface—an interface with a single method:

```
void valueChanged(TreeSelectionEvent event)
```

That method is called whenever the user selects or deselects tree nodes.

Add the listener to the tree in the normal way:

```
tree.addTreeSelectionListener(listener);
```

You can specify whether the user is allowed to select a single node, a contiguous range of nodes, or an arbitrary, potentially discontiguous, set of nodes. The `JTree` class uses a `TreeSelectionModel` to manage node selection. You need to retrieve the model to set the selection state to one of `SINGLE_TREE_SELECTION`, `CONTIGUOUS_TREE_SELECTION`, or `DISCONTIGUOUS_TREE_SELECTION`. (Discontiguous selection mode is the default.) For example, in our class browser, we want to allow selection of only a single class:

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

Apart from setting the selection mode, you need not worry about the tree selection model.

NOTE: How the user selects multiple items depends on the look-and-feel. In the Metal look-and-feel, hold down the Ctrl key while clicking an item to add it to the selection, or to remove it if it is currently selected. Hold down the Shift key while clicking an item to select a *range* of items, extending from the previously selected item to the new item.

To find out the current selection, query the tree with the `getSelectionPaths` method:

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

If you restricted the user to single-item selection, you can use the convenience method `getSelectionPath` which returns the first selected path or `null` if no path was selected.



CAUTION: The `TreeSelectionEvent` class has a `getPaths` method that returns an array of `TreePath` objects, but that array describes *selection changes*, not the current selection.

Listing 10.14 shows the frame class for the class tree program. The program displays inheritance hierarchies and customizes the display to show abstract classes in italics. (See Listing 10.15 for the cell renderer.) You can type the name of any class into the text field at the bottom of the frame. Press the Enter key or click the Add button to add the class and its superclasses to the tree. You must enter the full package name, such as `java.util.ArrayList`.

This program is a bit tricky because it uses reflection to construct the class tree. This work is done inside the `addClass` method. (The details are not that important. We use the class tree in this example because inheritance yields a nice supply of trees without laborious coding. When you display trees in your applications, you will have your own source of hierarchical data.) The method uses the

breadth-first search algorithm to find whether the current class is already in the tree by calling the `findUserObject` method that we implemented in the preceding section. If the class is not already in the tree, we add the superclasses to the tree, then make the new class node a child and make that node visible.

When you select a tree node, the text area to the right is filled with the fields of the selected class. In the frame constructor, we restrict the user to single-item selection and add a tree selection listener. When the `valueChanged` method is called, we ignore its event parameter and simply ask the tree for the current selection path. As always, we need to get the last node of the path and look up its user object. We then call the `getFieldDescription` method which uses reflection to assemble a string with all fields of the selected class.

Listing 10.14 `treeRender/ClassTreeFrame.java`

```
1 package treeRender;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.lang.reflect.*;
6 import java.util.*;
7
8 import javax.swing.*;
9 import javax.swing.tree.*;
10
11 /**
12 * This frame displays the class tree, a text field, and an "Add" button to add more classes
13 * into the tree.
14 */
15 public class ClassTreeFrame extends JFrame
16 {
17     private static final int DEFAULT_WIDTH = 400;
18     private static final int DEFAULT_HEIGHT = 300;
19
20     private DefaultMutableTreeNode root;
21     private DefaultTreeModel model;
22     private JTree tree;
23     private JTextField textField;
24     private JTextArea textArea;
25
26     public ClassTreeFrame()
27     {
28         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
29
30         // the root of the class tree is Object
31         root = new DefaultMutableTreeNode(java.lang.Object.class);
32         model = new DefaultTreeModel(root);
33         tree = new JTree(model);
```

```
34
35 // add this class to populate the tree with some data
36 addClass(getClass());
37
38 // set up node icons
39 ClassNameTreeCellRenderer renderer = new ClassNameTreeCellRenderer();
40 renderer.setClosedIcon(new ImageIcon(getClass().getResource("red-ball.gif")));
41 renderer.setOpenIcon(new ImageIcon(getClass().getResource("yellow-ball.gif")));
42 renderer.setLeafIcon(new ImageIcon(getClass().getResource("blue-ball.gif")));
43 tree.setCellRenderer(renderer);
44
45 // set up selection mode
46 tree.addTreeSelectionListener(event ->
47 {
48     // the user selected a different node--update description
49     TreePath path = tree.getSelectionPath();
50     if (path == null) return;
51     DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) path
52         .getLastPathComponent();
53     Class<?> c = (Class<?>) selectedNode.getUserObject();
54     String description = getFieldDescription(c);
55     textArea.setText(description);
56 });
57 int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
58 tree.getSelectionModel().setSelectionMode(mode);
59
60 // this text area holds the class description
61 textArea = new JTextArea();
62
63 // add tree and text area
64 JPanel panel = new JPanel();
65 panel.setLayout(new GridLayout(1, 2));
66 panel.add(new JScrollPane(tree));
67 panel.add(new JScrollPane(textArea));
68
69 add(panel, BorderLayout.CENTER);
70
71 addTextField();
72 }
73
74 /**
75 * Add the text field and "Add" button to add a new class.
76 */
77 public void addTextField()
78 {
79     JPanel panel = new JPanel();
80
81     ActionListener addListener = event ->
82     {
```

(Continues)

Listing 10.14 (Continued)

```
83     // add the class whose name is in the text field
84     try
85     {
86         String text = textField.getText();
87         addClass(Class.forName(text)); // clear text field to indicate success
88         textField.setText("");
89     }
90     catch (ClassNotFoundException e)
91     {
92         JOptionPane.showMessageDialog(null, "Class not found");
93     }
94 };
95
96 // new class names are typed into this text field
97 textField = new JTextField(20);
98 textField.addActionListener(addListener);
99 panel.add(textField);
100
101 JButton addButton = new JButton("Add");
102 addButton.addActionListener(addListener);
103 panel.add(addButton);
104
105 add(panel, BorderLayout.SOUTH);
106 }
107
108 /**
109 * Finds an object in the tree.
110 * @param obj the object to find
111 * @return the node containing the object or null if the object is not present in the tree
112 */
113 @SuppressWarnings("unchecked")
114 public DefaultMutableTreeNode findUserObject(Object obj)
115 {
116     // find the node containing a user object
117     Enumeration<TreeNode> e = (Enumeration<TreeNode>) root.breadthFirstEnumeration();
118     while (e.hasMoreElements())
119     {
120         DefaultMutableTreeNode node = (DefaultMutableTreeNode) e.nextElement();
121         if (node.getUserObject().equals(obj)) return node;
122     }
123     return null;
124 }
125
126 /**
127 * Adds a new class and any parent classes that aren't yet part of the tree
128 * @param c the class to add
129 * @return the newly added node
```

```
130     */
131     public DefaultMutableTreeNode addClass(Class<?> c)
132     {
133         // add a new class to the tree
134
135         // skip non-class types
136         if (c.isInterface() || c.isPrimitive()) return null;
137
138         // if the class is already in the tree, return its node
139         DefaultMutableTreeNode node = findUserObject(c);
140         if (node != null) return node;
141
142         // class isn't present--first add class parent recursively
143
144         Class<?> s = c.getSuperclass();
145
146         DefaultMutableTreeNode parent;
147         if (s == null) parent = root;
148         else parent = addClass(s);
149
150         // add the class as a child to the parent
151         DefaultMutableTreeNode newNode = new DefaultMutableTreeNode(c);
152         model.insertNodeInto(newNode, parent, parent.getChildCount());
153
154         // make node visible
155         TreePath path = new TreePath(model.getPathToRoot(newNode));
156         tree.makeVisible(path);
157
158         return newNode;
159     }
160
161 /**
162 * Returns a description of the fields of a class.
163 * @param the class to be described
164 * @return a string containing all field types and names
165 */
166 public static String getFieldDescription(Class<?> c)
167 {
168     // use reflection to find types and names of fields
169     StringBuilder r = new StringBuilder();
170     Field[] fields = c.getDeclaredFields();
171     for (int i = 0; i < fields.length; i++)
172     {
173         Field f = fields[i];
174         if ((f.getModifiers() & Modifier.STATIC) != 0) r.append("static ");
175         r.append(f.getType().getName());
176         r.append(" ");
177         r.append(f.getName());
178         r.append("\n");
179     }
180 }
```

(Continues)

Listing 10.14 (Continued)

```
179     }
180     return r.toString();
181   }
182 }
```

Listing 10.15 treeRender/ClassNameTreeCellRenderer.java

```
1 package treeRender;
2
3 import java.awt.*;
4 import java.lang.reflect.*;
5 import javax.swing.*;
6 import javax.swing.tree.*;
7
8 /**
9  * This class renders a class name either in plain or italic. Abstract classes are italic.
10 */
11 public class ClassNameTreeCellRenderer extends DefaultTreeCellRenderer
12 {
13     private Font plainFont = null;
14     private Font italicFont = null;
15
16     public Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected,
17             boolean expanded, boolean leaf, int row, boolean hasFocus)
18     {
19         super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row, hasFocus);
20         // get the user object
21         DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
22         Class<?> c = (Class<?>) node.getUserObject();
23
24         // the first time, derive italic font from plain font
25         if (plainFont == null)
26         {
27             plainFont = getFont();
28             // the tree cell renderer is sometimes called with a label that has a null font
29             if (plainFont != null) italicFont = plainFont.deriveFont(Font.ITALIC);
30         }
31
32         // set font to italic if the class is abstract, plain otherwise
33         if (((c.getModifiers() & Modifier.ABSTRACT) == 0) setFont(plainFont);
34         else setFont(italicFont);
35         return this;
36     }
37 }
```

javax.swing.JTree 1.2

- `TreePath getSelectionPath()`
- `TreePath[] getSelectionPaths()`

returns the first selected path, or an array of paths to all selected nodes. If no paths are selected, both methods return `null`.

javax.swing.event.TreeSelectionListener 1.2

- `void valueChanged(TreeSelectionEvent event)`
is called whenever nodes are selected or deselected.

javax.swing.event.TreeSelectionEvent 1.2

- `TreePath getPath()`
- `TreePath[] getPaths()`

gets the first path or all paths that have *changed* in this selection event. If you want to know the current selection, not the selection change, call `JTree.getSelectionPaths` instead.

10.3.6 Custom Tree Models

In the final example, we implement a program that inspects the contents of an object, just like a debugger does (see Figure 10.34).

Before going further, compile and run the example program. Each node corresponds to an instance field. If the field is an object, expand it to see *its* instance fields. The program inspects the contents of the frame window. If you poke around a few of the instance fields, you should be able to find some familiar classes. You'll also gain some respect for how complex the Swing user interface components are under the hood.

What's remarkable about the program is that the tree does not use the `DefaultTreeModel`. If you already have data that are hierarchically organized, you might not want to build a duplicate tree and worry about keeping both trees synchronized. That is the situation in our case—the inspected objects are already linked to each other through the object references, so there is no need to replicate the linking structure.

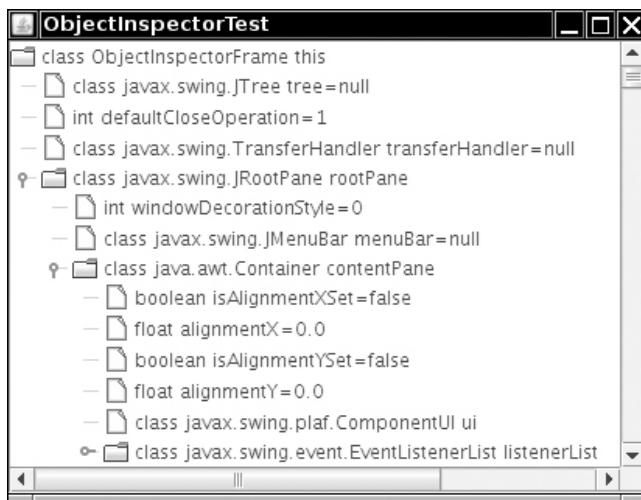


Figure 10.34 An object inspection tree

The `TreeModel` interface has only a handful of methods. The first group of methods enables the `JTree` to find the tree nodes by first getting the root, then the children. The `JTree` class calls these methods only when the user actually expands a node.

```
Object getRoot()  
int getChildCount(Object parent)  
Object getChild(Object parent, int index)
```

This example shows why the `TreeModel` interface, like the `JTree` class itself, does not need an explicit notion of nodes. The root and its children can be any objects. The `TreeModel` is responsible for telling the `JTree` how they are connected.

The next method of the `TreeModel` interface is the reverse of `getChild`:

```
int getIndexofChild(Object parent, Object child)
```

Actually, this method can be implemented in terms of the first three—see the code in Listing 10.16.

The tree model tells the `JTree` which nodes should be displayed as leaves:

```
boolean isLeaf(Object node)
```

If your code changes the tree model, the tree needs to be notified so that it can redraw itself. The tree adds itself as a `TreeModelListener` to the model. Thus, the model must support the usual listener management methods:

```
void addTreeModelListener(TreeModelListener l)  
void removeTreeModelListener(TreeModelListener l)
```

You can see the implementations for these methods in Listing 10.17.

When the model modifies the tree contents, it calls one of the four methods of the `TreeModelListener` interface:

```
void treeNodesChanged(TreeModelEvent e)
void treeNodesInserted(TreeModelEvent e)
void treeNodesRemoved(TreeModelEvent e)
void treeStructureChanged(TreeModelEvent e)
```

The `TreeModelEvent` object describes the location of the change. The details of assembling a tree model event that describes an insertion or removal event are quite technical. You only need to worry about firing these events if your tree can actually have nodes added and removed. In Listing 10.16, we show how to fire one event by replacing the root with a new object.



TIP: To simplify the code for event firing, use the `javax.swing.EventListenerList` convenience class that collects listeners. The last three methods of Listing 10.17 show how to use the class.

Finally, if the user edits a tree node, your model is called with the change:

```
void valueForPathChanged(TreePath path, Object newValue)
```

If you don't allow editing, this method is never called.

If you don't need to support editing, constructing a tree model is easily done. Implement the three methods

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

These methods describe the structure of the tree. Supply routine implementations of the other five methods, as in Listing 10.16. You are then ready to display your tree.

Now let's turn to the implementation of the example program. Our tree will contain objects of type `Variable`.

NOTE: Had we used the `DefaultTreeModel`, our nodes would have been objects of type `DefaultMutableTreeNode` with *user objects* of type `Variable`.

For example, suppose you inspect the variable

```
Employee joe;
```

That variable has a *type* `Employee.class`, a *name* "joe", and a *value*—the value of the object reference `joe`. In Listing 10.18, we define a class `Variable` that describes a variable in a program:

```
Variable v = new Variable(Employee.class, "joe", joe);
```

If the type of the variable is a primitive type, you must use an object wrapper for the value.

```
new Variable(double.class, "salary", new Double(salary));
```

If the type of the variable is a class, the variable has *fields*. Using reflection, we enumerate all fields and collect them in an `ArrayList`. Since the `getFields` method of the `Class` class does not return the fields of the superclass, we need to call `getFields` on all superclasses as well. You can find the code in the `Variable` constructor. The `getFields` method of our `Variable` class returns the array of fields. Finally, the `toString` method of the `Variable` class formats the node label. The label always contains the variable type and name. If the variable is not a class, the label also contains the value.

NOTE: If the type is an array, we do not display the elements of the array. This would not be difficult to do; we leave it as the proverbial “exercise for the reader.”

Let’s move on to the tree model. The first two methods are simple.

```
public Object getRoot()
{
    return root;
}

public int getChildCount(Object parent)
{
    return ((Variable) parent).getFields().size();
}
```

The `getChild` method returns a new `Variable` object that describes the field with the given index. The `getType` and `getName` methods of the `Field` class yield the field type and name. By using reflection, you can read the field value as `f.get(parentValue)`. That method can throw an `IllegalAccessException`. However, we made all fields accessible in the `Variable` constructor, so this won’t happen in practice.

Here is the complete code of the `getChild` method:

```
public Object getChild(Object parent, int index)
{
    ArrayList fields = ((Variable) parent).getFields();
    Field f = (Field) fields.get(index);
```

```
Object parentValue = ((Variable) parent).getValue();
try
{
    return new Variable(f.getType(), f.getName(), f.get(parentValue));
}
catch (IllegalAccessException e)
{
    return null;
}
```

These three methods reveal the structure of the object tree to the JTree component. The remaining methods are routine—see the source code in Listing 10.17.

There is one remarkable fact about this tree model: It actually describes an *infinite* tree. You can verify this by following one of the WeakReference objects. Click on the variable named referent. It leads you right back to the original object. You get an identical subtree, and you can open its WeakReference object again, ad infinitum. Of course, you cannot *store* an infinite set of nodes; the tree model simply generates the nodes on demand as the user expands the parents. Listing 10.16 shows the frame class of the sample program.

Listing 10.16 treeModel/ObjectInspectorFrame.java

```
1 package treeModel;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * This frame holds the object tree.
8  */
9 public class ObjectInspectorFrame extends JFrame
10 {
11     private JTree tree;
12     private static final int DEFAULT_WIDTH = 400;
13     private static final int DEFAULT_HEIGHT = 300;
14
15     public ObjectInspectorFrame()
16     {
17         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19         // we inspect this frame object
20
21         Variable v = new Variable(getClass(), "this", this);
22         ObjectTreeModel model = new ObjectTreeModel();
23         model.setRoot(v);
```

(Continues)

Listing 10.16 *(Continued)*

```
24
25     // construct and show tree
26
27     tree = new JTree(model);
28     add(new JScrollPane(tree), BorderLayout.CENTER);
29 }
30 }
```

Listing 10.17 `treeModel/ObjectTreeModel.java`

```
1 package treeModel;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5 import javax.swing.event.*;
6 import javax.swing.tree.*;
7
8 /**
9  * This tree model describes the tree structure of a Java object. Children are the objects that
10 * are stored in instance variables.
11 */
12 public class ObjectTreeModel implements TreeModel
13 {
14     private Variable root;
15     private EventListenerList listenerList = new EventListenerList();
16
17     /**
18      * Constructs an empty tree.
19      */
20     public ObjectTreeModel()
21     {
22         root = null;
23     }
24
25     /**
26      * Sets the root to a given variable.
27      * @param v the variable that is being described by this tree
28      */
29     public void setRoot(Variable v)
30     {
31         Variable oldRoot = v;
32         root = v;
33         fireTreeStructureChanged(oldRoot);
34     }
35
36     public Object getRoot()
```

```
37     {
38         return root;
39     }
40
41     public int getChildCount(Object parent)
42     {
43         return ((Variable) parent).getFields().size();
44     }
45
46     public Object getChild(Object parent, int index)
47     {
48         ArrayList<Field> fields = ((Variable) parent).getFields();
49         Field f = (Field) fields.get(index);
50         Object parentValue = ((Variable) parent).getValue();
51         try
52         {
53             return new Variable(f.getType(), f.getName(), f.get(parentValue));
54         }
55         catch (IllegalAccessException e)
56         {
57             return null;
58         }
59     }
60
61     public int getIndexofChild(Object parent, Object child)
62     {
63         int n = getChildCount(parent);
64         for (int i = 0; i < n; i++)
65             if (getChild(parent, i).equals(child)) return i;
66         return -1;
67     }
68
69     public boolean isLeaf(Object node)
70     {
71         return getChildCount(node) == 0;
72     }
73
74     public void valueForPathChanged(TreePath path, Object newValue)
75     {
76     }
77
78     public void addTreeModelListener(TreeModelListener l)
79     {
80         listenerList.add(TreeModelListener.class, l);
81     }
82
83     public void removeTreeModelListener(TreeModelListener l)
84     {
```

(Continues)

Listing 10.17 (Continued)

```
85     listenerList.remove(TreeModelListener.class, l);
86 }
87
88 protected void fireTreeStructureChanged(Object oldRoot)
89 {
90     TreeModelEvent event = new TreeModelEvent(this, new Object[] { oldRoot });
91     for (TreeModelListener l : listenerList.getListeners(TreeModelListener.class))
92         l.treeStructureChanged(event);
93 }
94 }
```

Listing 10.18 treeModel/Variable.java

```
1 package treeModel;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5
6 /**
7 * A variable with a type, name, and value.
8 */
9 public class Variable
10 {
11     private Class<?> type;
12     private String name;
13     private Object value;
14     private ArrayList<Field> fields;
15
16     /**
17      * Construct a variable.
18      * @param aType the type
19      * @param aName the name
20      * @param aValue the value
21      */
22     public Variable(Class<?> aType, String aName, Object aValue)
23     {
24         type = aType;
25         name = aName;
26         value = aValue;
27         fields = new ArrayList<>();
28
29         // find all fields if we have a class type except we don't expand strings and null values
30
31         if (!type.isPrimitive() && !type.isArray() && !type.equals(String.class) && value != null)
32         {
33             // get fields from the class and all superclasses
```

```
34         for (Class<?> c = value.getClass(); c != null; c = c.getSuperclass())
35     {
36         Field[] fs = c.getDeclaredFields();
37         AccessibleObject.setAccessible(fs, true);
38
39         // get all nonstatic fields
40         for (Field f : fs)
41             if ((f.getModifiers() & Modifier.STATIC) == 0) fields.add(f);
42     }
43 }
44
45 /**
46 * Gets the value of this variable.
47 * @return the value
48 */
49 public Object getValue()
50 {
51     return value;
52 }
53
54 /**
55 * Gets all nonstatic fields of this variable.
56 * @return an array list of variables describing the fields
57 */
58 public ArrayList<Field> getFields()
59 {
60     return fields;
61 }
62
63
64 public String toString()
65 {
66     String r = type + " " + name;
67     if (type.isPrimitive()) r += "=" + value;
68     else if (type.equals(String.class)) r += "=" + value;
69     else if (value == null) r += "=null";
70     return r;
71 }
72 }
```

javax.swing.tree.TreeModel 1.2

- `Object getRoot()`
returns the root node.
- `int getChildCount(Object parent)`
gets the number of children of the parent node.

(Continues)

javax.swing.tree.TreeModel 1.2 (Continued)

- `Object getChild(Object parent, int index)`
gets the child node of the parent node at the given index.
- `int getIndexOfChild(Object parent, Object child)`
gets the index of the child node in the parent node, or -1 if child is not a child of parent in this tree model.
- `boolean isLeaf(Object node)`
returns true if node is conceptually a leaf of the tree.
- `void addTreeModelListener(TreeModelListener l)`
- `void removeTreeModelListener(TreeModelListener l)`
adds or removes listeners that are notified when the information in the tree model changes.
- `void valueForPathChanged(TreePath path, Object newValue)`
is called when a cell editor has modified the value of a node.

Parameters: `path` The path to the node that has been edited
 `newValue` The replacement value returned by the editor

javax.swing.event.TreeModelListener 1.2

- `void treeNodesChanged(TreeModelEvent e)`
- `void treeNodesInserted(TreeModelEvent e)`
- `void treeNodesRemoved(TreeModelEvent e)`
- `void treeStructureChanged(TreeModelEvent e)`

is called by the tree model when the tree has been modified.

javax.swing.event.TreeModelEvent 1.2

- `TreeModelEvent(Object eventSource, TreePath node)`
constructs a tree model event.

Parameters: `eventSource` The tree model generating this event
 `node` The path to the node that is being changed

10.4 Text Components

Figure 10.35 shows all text components that are included in the Swing library. You already saw the three most commonly used components—`JTextField`, `JPasswordField`, and `JTextArea`—in Volume I, Chapter 9. In the following sections, we will introduce the remaining text components. We will also discuss the `JSpinner` component that contains a formatted text field together with tiny “up” and “down” buttons to change its contents.

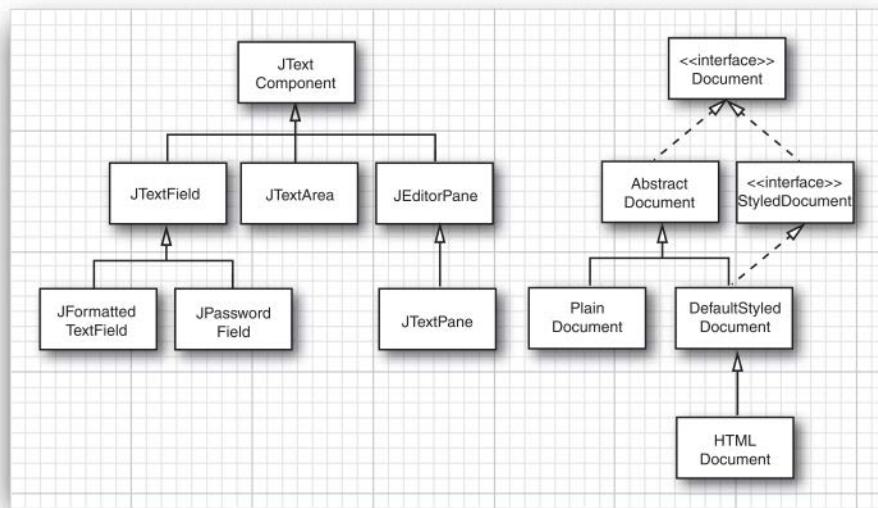


Figure 10.35 The hierarchy of text components and documents

All text components render and edit the data stored in a model object of a class implementing the `Document` interface. The `JTextField` and `JTextArea` components use a `PlainDocument` that simply stores a sequence of lines of plain text without any formatting.

A `JEditorPane` can show and edit styled text (with fonts, colors, etc.) in a variety of formats, most notably HTML; see Section 10.4.4, “Displaying HTML with the `JEditorPane`,” on p. 712. The `StyledDocument` interface describes the additional requirements of styles, fonts, and colors. The `HTMLDocument` class implements this interface.

The subclass `JTextPane` of `JEditorPane` also holds styled text as well as embedded Swing components. We do not cover the very complex `JTextPane` in this book but instead

refer you to the detailed description in *Core Swing* by Kim Topley. For a typical use of the `JTextPane` class, have a look at the `StylePad` demo that is included in the JDK.

10.4.1 Change Tracking in Text Components

Most of the intricacies of the `Document` interface are of interest only if you implement your own text editor. There is, however, one common use of the interface: tracking changes.

Sometimes, you may want to update a part of your user interface whenever a user edits text, without waiting for the user to click a button. Here is a simple example. We show three text fields for the red, blue, and green component of a color. Whenever the content of the text fields changes, the color should be updated. Figure 10.36 shows the running application of Listing 10.19.



Figure 10.36 Tracking changes in a text field

First of all, note that it is not a good idea to monitor keystrokes. Some keystrokes (such as the arrow keys) don't change the text. More importantly, the text can be updated by mouse gestures (such as "middle mouse button pasting" in X11). Instead, you should ask the *document* (and not the text component) to notify you whenever the data have changed by installing a *document listener*:

```
textField.getDocument().addDocumentListener(listener);
```

When the text has changed, one of the following `DocumentListener` methods is called:

```
void insertUpdate(DocumentEvent event)
void removeUpdate(DocumentEvent event)
void changedUpdate(DocumentEvent event)
```

The first two methods are called when characters have been inserted or removed. The third method is not called at all for text fields. For more complex document types, it would be called when some other change, such as a change in formatting, has occurred. Unfortunately, there is no single callback to tell you that the text has changed—usually you don't much care how it has changed. There is no adapter class, either. Thus, your document listener must implement all three methods. Here is what we do in our sample program:

```
DocumentListener listener = new DocumentListener()
{
    public void insertUpdate(DocumentEvent event) { setColor(); }
    public void removeUpdate(DocumentEvent event) { setColor(); }
    public void changedUpdate(DocumentEvent event) {}
}
```

The `setColor` method uses the `getText` method to obtain the current user input strings from the text fields and sets the color.

Our program has one limitation. Users can type malformed input, such as "twenty", into the text field, or leave a field blank. For now, we catch the `NumberFormatException` that the `parseInt` method throws, and we simply don't update the color when the text field entry is not a number. In the next section, you will see how you can prevent the user from entering invalid input in the first place.

NOTE: Instead of listening to document events, you can add an action event listener to a text field. The action listener is notified whenever the user presses the Enter key. We don't recommend this approach, because users don't always remember to press Enter when they are done entering data. If you use an action listener, you should also install a focus listener so that you can track when the user leaves the text field.

Listing 10.19 `textChange/ColorFrame.java`

```
1 package textChange;
2
3 import java.awt.*;
4 import javax.swing.*;
5 import javax.swing.event.*;
6
7 /**
8  * A frame with three text fields to set the background color.
9  */
10 public class ColorFrame extends JFrame
11 {
12     private JPanel panel;
13     private JTextField redField;
14     private JTextField greenField;
15     private JTextField blueField;
16
17     public ColorFrame()
18     {
```

(Continues)

Listing 10.19 (Continued)

```
19     DocumentListener listener = new DocumentListener()
20     {
21         public void insertUpdate(DocumentEvent event) { setColor(); }
22         public void removeUpdate(DocumentEvent event) { setColor(); }
23         public void changedUpdate(DocumentEvent event) {}
24     };
25
26     panel = new JPanel();
27
28     panel.add(new JLabel("Red:"));
29     redField = new JTextField("255", 3);
30     panel.add(redField);
31     redField.getDocument().addDocumentListener(listener);
32
33     panel.add(new JLabel("Green:"));
34     greenField = new JTextField("255", 3);
35     panel.add(greenField);
36     greenField.getDocument().addDocumentListener(listener);
37
38     panel.add(new JLabel("Blue:"));
39     blueField = new JTextField("255", 3);
40     panel.add(blueField);
41     blueField.getDocument().addDocumentListener(listener);
42
43     add(panel);
44     pack();
45 }
46
47 /**
48 * Set the background color to the values stored in the text fields.
49 */
50 public void setColor()
{
51     try
52     {
53         int red = Integer.parseInt(redField.getText().trim());
54         int green = Integer.parseInt(greenField.getText().trim());
55         int blue = Integer.parseInt(blueField.getText().trim());
56         panel.setBackground(new Color(red, green, blue));
57     }
58     catch (NumberFormatException e)
59     {
60         // don't set the color if the input can't be parsed
61     }
62 }
63 }
64 }
```

javax.swing.JComponent 1.2

- Dimension getPreferredSize()
- void setPreferredSize(Dimension d)

gets or sets the preferred size of this component.

javax.swing.text.Document 1.2

- int getLength()
returns the number of characters currently in the document.
- String getText(int offset, int length)
returns the text contained within the given portion of the document.
Parameters: offset The start of the text
 length The length of the desired string
- void addDocumentListener(DocumentListener listener)
registers the listener to be notified when the document changes.

javax.swing.event.DocumentEvent 1.2

- Document getDocument()
gets the document that is the source of the event.

javax.swing.event.DocumentListener 1.2

- void changedUpdate(DocumentEvent event)
is called whenever an attribute or set of attributes changes.
- void insertUpdate(DocumentEvent event)
is called whenever an insertion into the document occurs.
- void removeUpdate(DocumentEvent event)
is called whenever a portion of the document has been removed.

10.4.2 Formatted Input Fields

In the previous example program, we wanted the program user to type numbers, not arbitrary strings. That is, the user is allowed to enter only digits 0 through 9

and a hyphen (-). The hyphen, if present at all, must be the *first* symbol of the input string.

On the surface, this input validation task sounds simple. We can install a key listener to the text field and consume all key events that aren't digits or a hyphen. Unfortunately, this simple approach, although commonly recommended for input validation, does not work well in practice. First, not every combination of the valid input characters is a valid number. For example, --3 and 3-3 aren't valid, even though they are made up from valid input characters. But more importantly, there are other ways of changing the text that don't involve typing character keys. Depending on the look-and-feel, certain key combinations can be used to cut, copy, and paste text. For example, in the Metal look-and-feel, the Ctrl+V key combination pastes the content of the paste buffer into the text field. That is, we also need to monitor that the user doesn't paste in an invalid character. Filtering keystrokes to prevent invalid content begins to look like a real chore. This is certainly not something that an application programmer should have to worry about.

Perhaps surprisingly, before Java SE 1.4, there were no components for entering numeric values. Starting with the first edition of Core Java, we supplied an implementation for an `IntTextField`—a text field for entering a properly formatted integer. In later editions, we changed the implementation to extract whatever limited advantage we could from the various half-baked validation schemes added in each version of Java. Finally, in Java SE 1.4, the Swing designers faced the issue head-on and supplied a versatile `JFormattedTextField` class that can be used not just for numeric input but also for dates or even more esoteric formatted values such as IP addresses.

10.4.2.1 Integer Input

Let's get started with an easy case: a text field for integer input.

```
JFormattedTextField intField = new JFormattedTextField(NumberFormat.getIntegerInstance());
```

The `NumberFormat.getIntegerInstance` returns a formatter object that formats integers using the current locale. In the U.S. locale, commas are used as decimal separators, allowing users to enter values such as 1,729. Chapter 7 explains in detail how you can select other locales.

As with any text field, you can set the number of columns:

```
intField.setColumns(6);
```

You can set a default value with the `setValue` method. That method takes an `Object` parameter, so you'll need to wrap the default `int` value in an `Integer` object:

```
intField.setValue(new Integer(100));
```

Typically, users will supply inputs in multiple text fields and then click a button to read all values. When the button is clicked, you can get the user-supplied value with the `getValue` method. That method returns an `Object` result, and you need to cast it into the appropriate type. The `JFormattedTextField` returns an object of type `Long` if the user edited the value. However, if the user made no changes, the original `Integer` object is returned. Therefore, you should cast the return value to the common superclass `Number`:

```
Number value = (Number) intField.getValue();
int v = value.intValue();
```

The formatted text field is not very interesting until you consider what happens when a user provides illegal input. That is the topic of the next section.

10.4.2.2 Behavior on Loss of Focus

Consider what happens when a user supplies input to a text field. The user types input and eventually decides to leave the field, perhaps by clicking on another component with the mouse. Then the text field *loses focus*. The I-beam cursor is no longer visible in the text field, and keystrokes are directed toward a different component.

When the formatted text field loses focus, the formatter looks at the text string that the user produced. If the formatter knows how to convert the text string to an object, the text is valid. Otherwise it is invalid. You can use the `isValid` method to check whether the current content of the text field is valid.

The default behavior on loss of focus is called “commit or revert.” If the text string is valid, it is *committed*. The formatter converts it to an object. That object becomes the current value of the field (that is, the return value of the `getValue` method that you saw in the preceding section). The value is then converted back to a string, which becomes the text string visible in the field. For example, the integer formatter recognizes the input `1729` as valid, sets the current value to `new Long(1729)`, and converts it back into a string with a decimal comma: `1,729`.

Conversely, if the text string is invalid, the current value is not changed and the text field *reverts* to the string that represents the old value. For example, if the user enters a bad value, such as `x1`, the old value is restored when the text field loses focus.

NOTE: The integer formatter regards a text string as valid if it starts with an integer. For example, `1729x` is a valid string. It is converted to the number `1729`, which is then formatted as the string `1,729`.

You can set other behaviors with the `setFocusLostBehavior` method. The “commit” behavior is subtly different from the default. If the text string is invalid, then both the text string and the field value stay unchanged—they are now out of sync. The “persist” behavior is even more conservative. Even if the text string is valid, neither the text field nor the current value are changed. You would need to call `commitEdit`, `setValue`, or `setText` to bring them back in sync. Finally, there is a “revert” behavior that doesn’t seem to be useful at all: Whenever focus is lost, the user input is disregarded, and the text string reverts to the old value.

NOTE: Generally, the “commit or revert” default behavior is reasonable. There is just one potential problem. Suppose a dialog box contains a text field for an integer value. A user enters a string “1729”, with a leading space, and clicks the OK button. The leading space makes the number invalid, and the field value reverts to the old value. The action listener of the OK button retrieves the field value and closes the dialog box. The user never knows that the new value has been rejected. In this situation, it is appropriate to select the “commit” behavior and have the OK button listener check that all field edits are valid before closing the dialog box.

10.4.2.3 Filters

The basic functionality of formatted text fields is straightforward and sufficient for most uses. However, you can add a couple of refinements. Perhaps you want to prevent the user from entering nondigits altogether. You can achieve that behavior with a *document filter*. Recall that in the model-view-controller architecture, the controller translates input events into commands that modify the underlying document of the text field—that is, the text string stored in a `PlainDocument` object. For example, whenever the controller processes a command that causes text to be inserted into the document, it calls the “insert string” command. The string to be inserted can be either a single character or the content of the paste buffer. A document filter can intercept this command and modify the string or cancel the insertion altogether. Here is the code for the `insertString` method of a filter that analyzes the string to be inserted and inserts only the characters that are digits or a minus sign. (The code handles supplementary Unicode characters, as explained in Volume I, Chapter 3. See Chapter 2 for the `StringBuilder` class.)

```
public void insertString(FilterBypass fb, int offset, String string, AttributeSet attr)
    throws BadLocationException
{
    StringBuilder builder = new StringBuilder(string);
    for (int i = builder.length() - 1; i >= 0; i--)
    {
        int cp = builder.codePointAt(i);
```

```
if (!Character.isDigit(cp) && cp != '-')
{
    builder.deleteCharAt(i);
    if (Character.isSupplementaryCodePoint(cp))
    {
        i--;
        builder.deleteCharAt(i);
    }
}
super.insertString(fb, offset, builder.toString(), attr);
}
```

You should also override the `replace` method of the `DocumentFilter` class—it is called when text is selected and then replaced. The implementation of the `replace` method is straightforward—see Listing 10.21 on p. 698.

Now you need to install the document filter. Unfortunately, there is no straightforward method to do that. You need to override the `getDocumentFilter` method of a formatter class and pass an object of that formatter class to the `JFormattedTextField`. The integer text field uses an `InternationalFormatter` that is initialized with `NumberFormat.getIntegerInstance()`. Here is how you install a formatter to yield the desired filter:

```
JFormattedTextField intField = new JFormattedTextField(new
    InternationalFormatter(NumberFormat.getIntegerInstance())
{
    private DocumentFilter filter = new IntFilter();
    protected DocumentFilter getDocumentFilter()
    {
        return filter;
    }
});
```

NOTE: The Java SE documentation states that the `DocumentFilter` class was invented to avoid subclassing. Until Java SE 1.3, filtering in a text field was achieved by extending the `PlainDocument` class and overriding the `insertString` and `replace` methods. Now the `PlainDocument` class has a pluggable filter instead. That is a splendid improvement. It would have been even more splendid if the filter had also been made pluggable in the formatter class. Alas, it was not, and we must subclass the formatter.

Try out the `FormatTest` example program at the end of this section. The third text field has a filter installed. You can insert only digits or the minus (-) character. Note that you can still enter invalid strings such as 1-2-3. In general, it is impossible to avoid all invalid strings through filtering. For example, the string “-” is invalid,

but a filter can't reject it because it is a prefix of a legal string "-1". Even though filters can't give perfect protection, it makes sense to use them to reject inputs that are obviously invalid.



TIP: Another use for filtering is to turn all characters of a string to upper case.

Such a filter is easy to write. In the `insertString` and `replace` methods of the filter, convert the string to be inserted to upper case and then invoke the superclass method.

10.4.2.4 Verifiers

There is another potentially useful mechanism to alert users to invalid inputs. You can attach a *verifier* to any `JComponent`. If the component loses focus, the verifier is queried. If the verifier reports the content of the component to be invalid, the component immediately regains focus. The user is thus forced to fix the content before supplying other inputs.

A verifier must extend the abstract `InputVerifier` class and define a `verify` method. It is particularly easy to define a verifier that checks formatted text fields. The `isEditValid` method of the `JFormattedTextField` class calls the formatter and returns `true` if the formatter can turn the text string into an object. Here is the verifier, attached to a `JFormattedTextField`:

```
intField.setInputVerifier(new InputVerifier()
{
    public boolean verify(JComponent component)
    {
        JFormattedTextField field = (JFormattedTextField) component;
        return field.isEditValid();
    }
});
```

The fourth text field in the example program has this verifier attached. Try entering an invalid number (such as `x1729`) and press the Tab key or click with the mouse on another text field. Note that the field immediately regains focus. However, if you click the OK button, the action listener calls `getValue`, which reports the last good value.

A verifier is not entirely foolproof. If you click on a button, the button notifies its action listeners before an invalid component regains focus. The action listeners can then get an invalid result from the component that failed verification. There is a reason for this behavior: Users might want to click a Cancel button without first having to fix an invalid input.

10.4.2.5 Other Standard Formatters

Besides the integer formatter, the `JFormattedTextField` supports several other formatters. The `NumberFormat` class has static methods

```
getNumberInstance  
getCurrencyInstance  
getPercentInstance
```

that yield formatters of floating-point numbers, currency values, and percentages. For example, you can obtain a text field for the input of currency values by calling

```
JFormattedTextField currencyField = new JFormattedTextField(NumberFormat.getCurrencyInstance());
```

To edit dates and times, call one of the static methods of the `DateFormat` class:

```
getDateInstance  
getTimeInstance  
getDateTimeInstance
```

For example,

```
JFormattedTextField dateField = new JFormattedTextField(DateFormat.getDateInstance());
```

This field edits a date in the default or “medium” format such as

Aug 5, 2007

You can instead choose a “short” format such as

8/5/07

by calling

```
DateFormat.getDateInstance(DateFormat.SHORT)
```

NOTE: By default, the date format is “lenient.” That is, an invalid date such as February 31, 2002, is rolled over to the next valid date, March 3, 2002. That behavior might be surprising to your users. In that case, call `setLenient(false)` on the `DateFormat` object.

The `DefaultFormatter` can format objects of any class that has a constructor with a `String` parameter and a matching `toString` method. For example, the `URL` class has a `URL(String)` constructor that can be used to construct a URL from a string, such as

```
URL url = new URL("http://horstmann.com");
```

Therefore, you can use the `DefaultFormatter` to format `URL` objects. The formatter calls `toString` on the field value to initialize the field text. When the field loses focus, the formatter constructs a new object of the same class as the current value, using

the constructor with a `String` parameter. If that constructor throws an exception, the edit is not valid. You can try that out in the example program by entering a URL that does not start with a prefix such as `http:`.

NOTE: By default, the `DefaultFormatter` is in *overwrite mode*. That is different from the other formatters and not very useful. Call `setOverwriteMode(false)` to turn off overwrite mode.

Finally, the `MaskFormatter` is useful for fixed-size patterns that contain some constant and some variable characters. For example, Social Security numbers (such as 078-05-1120) can be formatted with a

```
new MaskFormatter("###-##-####")
```

The `#` symbol denotes a single digit. Table 10.3 shows the symbols that you can use in a mask formatter.

Table 10.3 `MaskFormatter` Symbols

Symbol	Explanation
#	A digit
?	A letter
U	A letter, converted to upper case
L	A letter, converted to lower case
A	A letter or digit
H	A hexadecimal digit [0-9A-Fa-f]
*	Any character
'	Escape character to include a symbol in the pattern

You can restrict the characters that can be typed into the field by calling one of the methods of the `MaskFormatter` class:

```
setValidCharacters  
setInvalidCharacters
```

For example, to read in a letter grade (such as A+ or F), you could use

```
MaskFormatter formatter = new MaskFormatter("U*");  
formatter.setValidCharacters("ABCDF+- ");
```

However, there is no way of specifying that the second character cannot be a letter.

Note that the string formatted by the mask formatter has exactly the same length as the pattern. If the user erases characters during editing, they are replaced with the *placeholder character*. The default placeholder character is space, but you can change it with the `setPlaceholderCharacter` method, for example,

```
formatter.setPlaceholderCharacter('0');
```

By default, a mask formatter is in overtype mode, which is quite intuitive—try it out in the example program. Also, note that the caret position jumps over the fixed characters in the mask.

The mask formatter is very effective for rigid patterns such as Social Security numbers or American telephone numbers. However, note that no variation at all is permitted in the mask pattern. For example, you cannot use a mask formatter for international telephone numbers that have a variable number of digits.

10.4.2.6 Custom Formatters

If none of the standard formatters is appropriate, it is fairly easy to define your own formatter. Consider 4-byte IP addresses such as

```
130.65.86.66
```

You can't use a `MaskFormatter` because each byte might be represented by one, two, or three digits. Also, we want to check in the formatter that each byte's value is at most 255.

To define your own formatter, extend the `DefaultFormatter` class and override the methods

```
String valueToString(Object value)  
Object stringToValue(String text)
```

The first method turns the field value into the string that is displayed in the text field. The second method parses the text that the user typed and turns it back into an object. If either method detects an error, it should throw a `ParseException`.

In our example program, we store an IP address in a `byte[]` array of length 4. The `valueToString` method forms a string that separates the bytes with periods. Note that byte values are signed quantities between -128 and 127. (For example, in an IP address 130.65.86.66, the first octet is actually the byte with value -126.) To turn negative byte values into unsigned integer values, add 256.

```
public String valueToString(Object value) throws ParseException  
{  
    if (!(value instanceof byte[]))  
        throw new ParseException("Not a byte[]", 0);  
    byte[] a = (byte[]) value;  
    if (a.length != 4)
```

```
        throw new ParseException("Length != 4", 0);
StringBuilder builder = new StringBuilder();
for (int i = 0; i < 4; i++)
{
    int b = a[i];
    if (b < 0) b += 256;
    builder.append(String.valueOf(b));
    if (i < 3) builder.append('.');
}
return builder.toString();
}
```

Conversely, the `stringValue` method parses the string and produces a `byte[]` object if the string is valid. If not, it throws a `ParseException`.

```
public Object stringValue(String text) throws ParseException
{
    StringTokenizer tokenizer = new StringTokenizer(text, ".");
    byte[] a = new byte[4];

    for (int i = 0; i < 4; i++)
    {
        int b = 0;
        try
        {
            b = Integer.parseInt(tokenizer.nextToken());
        }
        catch (NumberFormatException e)
        {
            throw new ParseException("Not an integer", 0);
        }
        if (b < 0 || b >= 256)
            throw new ParseException("Byte out of range", 0);
        a[i] = (byte) b;
    }
    return a;
}
```

Try out the IP address field in the sample program. If you enter an invalid address, the field reverts to the last valid address. The complete formatter is shown in Listing 10.22.

The program in Listing 10.20 shows various formatted text fields in action (see Figure 10.37). Click the OK button to retrieve the current values from the fields.

NOTE: The “Swing Connection” online newsletter has a short article describing a formatter that matches any regular expression. See www.oracle.com/technetwork/java/refft-138955.html.



Figure 10.37 The FormatTest program

Listing 10.20 textFormat/FormatTestFrame.java

```
1 package textFormat;
2
3 import java.awt.*;
4 import java.net.*;
5 import java.text.*;
6 import java.util.*;
7
8 import javax.swing.*;
9 import javax.swing.text.*;
10
11 /**
12 * A frame with a collection of formatted text fields and a button that displays the field values.
13 */
14 public class FormatTestFrame extends JFrame
15 {
16     private DocumentFilter filter = new IntFilter();
17     private JButton okButton;
18     private JPanel mainPanel;
19
20     public FormatTestFrame()
21     {
22         JPanel buttonPanel = new JPanel();
23         okButton = new JButton("Ok");
24         buttonPanel.add(okButton);
25         add(buttonPanel, BorderLayout.SOUTH);
26
27         mainPanel = new JPanel();
28         mainPanel.setLayout(new GridLayout(0, 3));
```

(Continues)

Listing 10.20 (Continued)

```
29     add(mainPanel, BorderLayout.CENTER);
30
31     JFormattedTextField intField = new JFormattedTextField(NumberFormat.getIntegerInstance());
32     intField.setValue(new Integer(100));
33     addRow("Number:", intField);
34
35     JFormattedTextField intField2 = new JFormattedTextField(NumberFormat.getIntegerInstance());
36     intField2.setValue(new Integer(100));
37     intField2.setFocusLostBehavior(JFormattedTextField.COMMIT);
38     addRow("Number (Commit behavior):", intField2);
39
40     JFormattedTextField intField3 = new JFormattedTextField(new InternationalFormatter(
41         NumberFormat.getIntegerInstance())
42     {
43         protected DocumentFilter getDocumentFilter()
44     {
45         return filter;
46     }
47 });
48     intField3.setValue(new Integer(100));
49     addRow("Filtered Number", intField3);
50
51     JFormattedTextField intField4 = new JFormattedTextField(NumberFormat.getIntegerInstance());
52     intField4.setValue(new Integer(100));
53     intField4.setInputVerifier(new InputVerifier()
54     {
55         public boolean verify(JComponent component)
56     {
57         JFormattedTextField field = (JFormattedTextField) component;
58         return field.isEditValid();
59     }
60 });
61     addRow("Verified Number:", intField4);
62
63     JFormattedTextField currencyField = new JFormattedTextField(NumberFormat
64         .getCurrencyInstance());
65     currencyField.setValue(new Double(10));
66     addRow("Currency:", currencyField);
67
68     JFormattedTextField dateField = new JFormattedTextField(DateFormat.getDateInstance());
69     dateField.setValue(new Date());
70     addRow("Date (default):", dateField);
71
72     DateFormat format = DateFormat.getDateInstance(DateFormat.SHORT);
```

```
73     format.setLenient(false);
74     JFormattedTextField dateField2 = new JFormattedTextField(format);
75     dateField2.setValue(new Date());
76     addRow("Date (short, not lenient):", dateField2);
77
78     try
79     {
80         DefaultFormatter formatter = new DefaultFormatter();
81         formatter.setOverwriteMode(false);
82         JFormattedTextField urlField = new JFormattedTextField(formatter);
83         urlField.setValue(new URL("http://java.sun.com"));
84         addRow("URL:", urlField);
85     }
86     catch (MalformedURLException ex)
87     {
88         ex.printStackTrace();
89     }
90
91     try
92     {
93         MaskFormatter formatter = new MaskFormatter("###-##-####");
94         formatter.setPlaceholderCharacter('0');
95         JFormattedTextField ssnField = new JFormattedTextField(formatter);
96         ssnField.setValue("078-05-1120");
97         addRow("SSN Mask:", ssnField);
98     }
99     catch (ParseException ex)
100    {
101        ex.printStackTrace();
102    }
103
104    JFormattedTextField ipField = new JFormattedTextField(new IPAddressFormatter());
105    ipField.setValue(new byte[] { (byte) 130, 65, 86, 66 });
106    addRow("IP Address:", ipField);
107    pack();
108 }
109
110 /**
111 * Adds a row to the main panel.
112 * @param labelText the label of the field
113 * @param field the sample field
114 */
115 public void addRow(String labelText, final JFormattedTextField field)
116 {
117     mainPanel.add(new JLabel(labelText));
118     mainPanel.add(field);
119     final JLabel valueLabel = new JLabel();
120     mainPanel.add(valueLabel);
```

(Continues)

Listing 10.20 *(Continued)*

```
121     okButton.addActionListener(event ->
122     {
123         Object value = field.getValue();
124         Class<?> cl = value.getClass();
125         String text = null;
126         if (cl.isArray())
127         {
128             if (cl.getComponentType().isPrimitive())
129             {
130                 try
131                 {
132                     text = Arrays.class.getMethod("toString", cl).invoke(null, value)
133                         .toString();
134                 }
135                 catch (ReflectiveOperationException ex)
136                 {
137                     // ignore reflection exceptions
138                 }
139             }
140             else text = Arrays.toString((Object[]) value);
141         }
142         else text = value.toString();
143         valueLabel.setText(text);
144     });
145 }
146 }
```

Listing 10.21 *textFormat/IntFilter.java*

```
1 package textFormat;
2
3 import javax.swing.text.*;
4
5 /**
6  * A filter that restricts input to digits and a '-' sign.
7  */
8 public class IntFilter extends DocumentFilter
9 {
10     public void insertString(FilterBypass fb, int offset, String string, AttributeSet attr)
11         throws BadLocationException
12     {
13         StringBuilder builder = new StringBuilder(string);
14         for (int i = builder.length() - 1; i >= 0; i--)
15         {
16             int cp = builder.codePointAt(i);
```

```
17     if (!Character.isDigit(cp) && cp != '-')
18     {
19         builder.deleteCharAt(i);
20         if (Character.isSupplementaryCodePoint(cp))
21         {
22             i--;
23             builder.deleteCharAt(i);
24         }
25     }
26 }
27 super.insertString(fb, offset, builder.toString(), attr);
28 }
29
30 public void replace(FilterBypass fb, int offset, int length, String string, AttributeSet attr)
31     throws BadLocationException
32 {
33     if (string != null)
34     {
35         StringBuilder builder = new StringBuilder(string);
36         for (int i = builder.length() - 1; i >= 0; i--)
37         {
38             int cp = builder.codePointAt(i);
39             if (!Character.isDigit(cp) && cp != '-')
40             {
41                 builder.deleteCharAt(i);
42                 if (Character.isSupplementaryCodePoint(cp))
43                 {
44                     i--;
45                     builder.deleteCharAt(i);
46                 }
47             }
48         }
49         string = builder.toString();
50     }
51     super.replace(fb, offset, length, string, attr);
52 }
53 }
```

Listing 10.22 `textFormat/IPAddressFormatter.java`

```
1 package textFormat;
2
3 import java.text.*;
4 import java.util.*;
5 import javax.swing.text.*;
```

(Continues)

Listing 10.22 (Continued)

```
6  /**
7  * A formatter for 4-byte IP addresses of the form a.b.c.d
8  */
9
10 public class IPAddressFormatter extends DefaultFormatter
11 {
12     public String valueToString(Object value) throws ParseException
13     {
14         if (!(value instanceof byte[])) throw new ParseException("Not a byte[]", 0);
15         byte[] a = (byte[]) value;
16         if (a.length != 4) throw new ParseException("Length != 4", 0);
17         StringBuilder builder = new StringBuilder();
18         for (int i = 0; i < 4; i++)
19         {
20             int b = a[i];
21             if (b < 0) b += 256;
22             builder.append(String.valueOf(b));
23             if (i < 3) builder.append('.');
24         }
25         return builder.toString();
26     }
27
28     public Object stringToValue(String text) throws ParseException
29     {
30         StringTokenizer tokenizer = new StringTokenizer(text, ".");
31         byte[] a = new byte[4];
32         for (int i = 0; i < 4; i++)
33         {
34             int b = 0;
35             if (!tokenizer.hasMoreTokens()) throw new ParseException("Too few bytes", 0);
36             try
37             {
38                 b = Integer.parseInt(tokenizer.nextToken());
39             }
40             catch (NumberFormatException e)
41             {
42                 throw new ParseException("Not an integer", 0);
43             }
44             if (b < 0 || b >= 256) throw new ParseException("Byte out of range", 0);
45             a[i] = (byte) b;
46         }
47         if (tokenizer.hasMoreTokens()) throw new ParseException("Too many bytes", 0);
48         return a;
49     }
50 }
```

javax.swing.JFormattedTextField 1.4

- `JFormattedTextField(Format fmt)`
constructs a text field that uses the specified format.
- `JFormattedTextField(JFormattedTextField.AbstractFormatter formatter)`
constructs a text field that uses the specified formatter. Note that `DefaultFormatter` and `InternationalFormatter` are subclasses of `JFormattedTextField.AbstractFormatter`.
- `Object getValue()`
returns the current valid value of the field. Note that this might not correspond to the string being edited.
- `void setValue(Object value)`
attempts to set the value of the given object. The attempt fails if the formatter cannot convert the object to a string.
- `void commitEdit()`
attempts to set the valid value of the field from the edited string. The attempt might fail if the formatter cannot convert the string.
- `boolean isEditValid()`
checks whether the edited string represents a valid value.
- `int getFocusLostBehavior()`
- `void setFocusLostBehavior(int behavior)`
gets or sets the “focus lost” behavior. Legal values for `behavior` are the constants `COMMIT_OR_REVERT`, `REVERT`, `COMMIT`, and `PERSIST` of the `JFormattedTextField` class.

javax.swing.JFormattedTextField.AbstractFormatter 1.4

- `abstract String valueToString(Object value)`
converts a value to an editable string. Throws a `ParseException` if `value` is not appropriate for this formatter.
- `abstract Object stringToValue(String s)`
converts a string to a value. Throws a `ParseException` if `s` is not in the appropriate format.
- `DocumentFilter getDocumentFilter()`
Override this method to provide a document filter that restricts inputs into the text field. A return value of `null` indicates that no filtering is needed.

javax.swing.text.DefaultFormatter 1.3

- `boolean getOverwriteMode()`
 - `void setOverwriteMode(boolean mode)`

gets or sets the overwrite mode. If `mode` is true, new characters overwrite existing characters when editing text.

javax.swing.text.DocumentFilter 1.4

- void insertString(DocumentFilter.FilterBypass bypass, int offset, String text, AttributeSet attrib)

is invoked before a string is inserted into a document. You can override the method and modify the string. You can disable insertion by not calling `super.insertString` or by calling bypass methods to modify the document without filtering.

Parameters: bypass An object that allows you to execute edit commands that bypass the filter

offset The offset at which to insert the text

text The characters to insert

attrib The formatting attribute

```
documentFilter.FilterBypass.bypass int offset int length String text Attr
```

- ```
var replace = document.createElement('div').createBypassBypass, offset, length, string, attributeset attrib)
```

is invoked before a part of a document is replaced with a new string. You can override the method and modify the string. You can disable replacement by not calling `super.replace` or by calling `bypass` methods to modify the document without filtering.

*Parameters:* bypass An object that allows you to execute edit commands that bypass the filter

**offset** The offset at which to insert the text

length The length of the

text The characters to insert

attrib The formatting attribute

```
mentFilter.FilterBypass bypass, int offset, int length)
```

- is invoked before a part of a document is removed. Get the document's

`bypass.getDocument()` if you need to analyze the effect of the removal.

**Parameters:** `bypass` An object that allows you to execute edit commands.

**offset** The offset of the part to be removed.

**length** The length of the part to be removed.

The length of the

**javax.swing.text.MaskFormatter 1.4**

- `MaskFormatter(String mask)`

constructs a mask formatter with the given mask. See Table 10.3 on p. 692 for the symbols in a mask.

- `String getValidCharacters()`

- `void setValidCharacters(String characters)`

gets or sets the valid editing characters. Only the characters in the given string are accepted for the variable parts of the mask.

- `String getInvalidCharacters()`

- `void setInvalidCharacters(String characters)`

gets or sets the invalid editing characters. None of the characters in the given string are accepted as input.

- `char getPlaceholderCharacter()`

- `void setPlaceholderCharacter(char ch)`

gets or sets the placeholder character used for the mask's variable characters that the user has not yet supplied. The default placeholder character is space.

- `String getPlaceholder()`

- `void setPlaceholder(String s)`

gets or sets the placeholder string. Its tail end is used if the user has not supplied all variable characters in the mask. If it is `null` or shorter than the mask, the placeholder character fills remaining inputs.

- `boolean getValueContainsLiteralCharacters()`

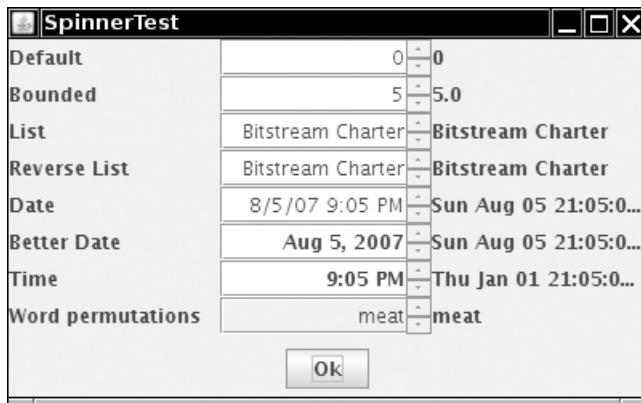
- `void setValueContainsLiteralCharacters(boolean b)`

gets or sets the “value contains literal characters” flag. If this flag is `true`, the field value contains the literal (nonvariable) parts of the mask. If it is `false`, the literal characters are removed. The default is `true`.

### 10.4.3 The JSpinner Component

A `JSpinner` is a component that contains a text field and two small buttons on the side. When the buttons are clicked, the text field value is incremented or decremented (see Figure 10.38).

The values in the spinner can be numbers, dates, values from a list, or, in the most general case, any sequence of values for which predecessors and successors can be determined. The `JSpinner` class defines standard data models for the first three cases. You can define your own data model to describe arbitrary sequences.



**Figure 10.38** Several variations of the `JSpinner` component

By default, a spinner manages an integer, and the buttons increment or decrement it by 1. You can get the current value by calling the `getValue` method. That method returns an `Object`. Cast it to an `Integer` and retrieve the wrapped value.

```
JSpinner defaultSpinner = new JSpinner();
...
int value = (Integer) defaultSpinner.getValue();
```

You can change the increment to a value other than 1, and you can supply the lower and upper bounds. Here is a spinner with the starting value of 5 and the increment of 0.5, bounded between 0 and 10:

```
JSpinner boundedSpinner = new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
```

There are two `SpinnerNumberModel` constructors, one with only `int` parameters and one with `double` parameters. If any of the parameters is a floating-point number, the second constructor is used. It sets the spinner value to a `Double` object.

Spinners aren't restricted to numeric values. You can have a spinner iterate through any collection of values. Simply pass a `SpinnerListModel` to the `JSpinner` constructor. You can construct a `SpinnerListModel` from an array or a class implementing the `List` interface (such as an `ArrayList`). In our sample program, we display a spinner control with all available font names.

```
String[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
```

However, we found that the direction of the iteration was mildly confusing because it is opposite to that of a combo box. In a combo box, higher values are *below* lower values, so you would expect the downward arrow to navigate toward higher values. But the spinner increments the array index so that the upward arrow

yields higher values. There is no provision for reversing the traversal order in the `SpinnerListModel`, but an impromptu anonymous subclass yields the desired result:

```
JSpinner reverseListSpinner = new JSpinner(
 new SpinnerListModel(fonts)
 {
 public Object getNextValue()
 {
 return super.getPreviousValue();
 }

 public Object getPreviousValue()
 {
 return super.getNextValue();
 }
 });
```

Try both versions and see which you find more intuitive.

Another good use for a spinner is for a date that the user can increment or decrement. You can get such a spinner, initialized with today's date, with the call

```
JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
```

However, if you look at Figure 10.38, you will see that the spinner text shows both date and time, such as

8/05/07 9:05 PM

The time doesn't make any sense for a date picker. It turns out to be somewhat difficult to make the spinner show just the date. Here is the magic incantation:

```
JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
String pattern = ((SimpleDateFormat) DateFormat.getDateInstance().toPattern();
betterDateSpinner.setEditor(new JSpinner.DateEditor(betterDateSpinner, pattern));
```

Using the same approach, you can also make a time picker:

```
JSpinner timeSpinner = new JSpinner(new SpinnerDateModel());
pattern = ((SimpleDateFormat) DateFormat.getTimeInstance(DateFormat.SHORT)).toPattern();
timeSpinner.setEditor(new JSpinner.DateEditor(timeSpinner, pattern));
```

You can display arbitrary sequences in a spinner by defining your own spinner model. In our sample program, we have a spinner that iterates through all permutations of the string "meat". You can get to "mate", "meta", "team", and the rest of the 24 permutations by clicking the spinner buttons.

To define your own model, extend the `AbstractSpinnerModel` class and define the following four methods:

```
Object getValue()
void setValue(Object value)
```

```
Object getNextValue()
Object getPreviousValue()
```

The `getValue` method returns the value stored by the model. The `setValue` method sets a new value. It should throw an `IllegalArgumentException` if the new value is not appropriate.



**CAUTION:** The `setValue` method must call the `fireStateChanged` method after setting the new value. Otherwise, the spinner field won't be updated.

---

The `getNextValue` and `getPreviousValue` methods return the values that should come after or before the current value, or `null` if the end of the traversal has been reached.



**CAUTION:** The `getNextValue` and `getPreviousValue` methods should *not* change the current value. When a user clicks on the upward arrow of the spinner, the `getNextValue` method is called. If the return value is not `null`, it is set by a call to `setValue`.

---

In the sample program, we use a standard algorithm to determine the next and previous permutations (see Listing 10.24). The details of the algorithm are not important.

Listing 10.23 shows how to generate the various spinner types. Click the OK button to see the spinner values.

---

#### Listing 10.23 spinner/SpinnerFrame.java

---

```
1 package spinner;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.text.*;
6 import javax.swing.*;
7
8 /**
9 * A frame with a panel that contains several spinners and a button that displays the spinner
10 * values.
11 */
12 public class SpinnerFrame extends JFrame
13 {
14 private JPanel mainPanel;
15 private JButton okButton;
```

```
16
17 public SpinnerFrame()
18 {
19 JPanel buttonPanel = new JPanel();
20 okButton = new JButton("Ok");
21 buttonPanel.add(okButton);
22 add(buttonPanel, BorderLayout.SOUTH);
23
24 mainPanel = new JPanel();
25 mainPanel.setLayout(new GridLayout(0, 3));
26 add(mainPanel, BorderLayout.CENTER);
27
28 JSpinner defaultSpinner = new JSpinner();
29 addRow("Default", defaultSpinner);
30
31 JSpinner boundedSpinner = new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
32 addRow("Bounded", boundedSpinner);
33
34 String[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment()
35 .getAvailableFontFamilyNames();
36
37 JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
38 addRow("List", listSpinner);
39
40 JSpinner reverseListSpinner = new JSpinner(new SpinnerListModel(fonts)
41 {
42 public Object getNextValue() { return super.getPreviousValue(); }
43 public Object getPreviousValue() { return super.getNextValue(); }
44 });
45 addRow("Reverse List", reverseListSpinner);
46
47 JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
48 addRow("Date", dateSpinner);
49
50 JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
51 String pattern = ((SimpleDateFormat) DateFormat.getDateInstance()).toPattern();
52 betterDateSpinner.setEditor(new JSpinner.DateEditor(betterDateSpinner, pattern));
53 addRow("Better Date", betterDateSpinner);
54
55 JSpinner timeSpinner = new JSpinner(new SpinnerDateModel());
56 pattern = ((SimpleDateFormat) DateFormat.getTimeInstance(DateFormat.SHORT)).toPattern();
57 timeSpinner.setEditor(new JSpinner.DateEditor(timeSpinner, pattern));
58 addRow("Time", timeSpinner);
59
60 JSpinner permSpinner = new JSpinner(new PermutationSpinnerModel("meat"));
61 addRow("Word permutations", permSpinner);
62 pack();
63 }
64 }
```

(Continues)

**Listing 10.23** *(Continued)*

```
65 /**
66 * Adds a row to the main panel.
67 * @param labelText the label of the spinner
68 * @param spinner the sample spinner
69 */
70 public void addRow(String labelText, final JSpinner spinner)
71 {
72 mainPanel.add(new JLabel(labelText));
73 mainPanel.add(spinner);
74 final JLabel valueLabel = new JLabel();
75 mainPanel.add(valueLabel);
76 okButton.addActionListener(event ->
77 {
78 Object value = spinner.getValue();
79 valueLabel.setText(value.toString());
80 });
81 }
82 }
```

---

**Listing 10.24** *spinner/PermutationSpinnerModel.java*

```
1 package spinner;
2
3 import javax.swing.*;
4
5 /**
6 * A model that dynamically generates word permutations.
7 */
8 public class PermutationSpinnerModel extends AbstractSpinnerModel
9 {
10 private String word;
11
12 /**
13 * Constructs the model.
14 * @param w the word to permute
15 */
16 public PermutationSpinnerModel(String w)
17 {
18 word = w;
19 }
20
21 public Object getValue()
22 {
23 return word;
24 }
25 }
```

```
26 public void setValue(Object value)
27 {
28 if (!(value instanceof String)) throw new IllegalArgumentException();
29 word = (String) value;
30 fireStateChanged();
31 }
32
33 public Object getNextValue()
34 {
35 int[] codePoints = toCodePointArray(word);
36 for (int i = codePoints.length - 1; i > 0; i--)
37 {
38 if (codePoints[i - 1] < codePoints[i])
39 {
40 int j = codePoints.length - 1;
41 while (codePoints[i - 1] > codePoints[j])
42 j--;
43 swap(codePoints, i - 1, j);
44 reverse(codePoints, i, codePoints.length - 1);
45 return new String(codePoints, 0, codePoints.length);
46 }
47 }
48 reverse(codePoints, 0, codePoints.length - 1);
49 return new String(codePoints, 0, codePoints.length);
50 }
51
52 public Object getPreviousValue()
53 {
54 int[] codePoints = toCodePointArray(word);
55 for (int i = codePoints.length - 1; i > 0; i--)
56 {
57 if (codePoints[i - 1] > codePoints[i])
58 {
59 int j = codePoints.length - 1;
60 while (codePoints[i - 1] < codePoints[j])
61 j--;
62 swap(codePoints, i - 1, j);
63 reverse(codePoints, i, codePoints.length - 1);
64 return new String(codePoints, 0, codePoints.length);
65 }
66 }
67 reverse(codePoints, 0, codePoints.length - 1);
68 return new String(codePoints, 0, codePoints.length);
69 }
70
71 private static int[] toCodePointArray(String str)
72 {
73 int[] codePoints = new int[str.codePointCount(0, str.length())];
```

(Continues)

**Listing 10.24 (Continued)**

```
74 for (int i = 0, j = 0; i < str.length(); i++, j++)
75 {
76 int cp = str.codePointAt(i);
77 if (Character.isSupplementaryCodePoint(cp)) i++;
78 codePoints[j] = cp;
79 }
80 return codePoints;
81 }
82
83 private static void swap(int[] a, int i, int j)
84 {
85 int temp = a[i];
86 a[i] = a[j];
87 a[j] = temp;
88 }
89
90 private static void reverse(int[] a, int i, int j)
91 {
92 while (i < j)
93 {
94 swap(a, i, j);
95 i++;
96 j--;
97 }
98 }
99 }
```

---

**javax.swing.JSpinner 1.4**

- **JSpinner()**  
constructs a spinner that edits an integer with starting value 0, increment 1, and no bounds.
- **JSpinner(SpinnerModel model)**  
constructs a spinner that uses the given data model.
- **Object getValue()**  
gets the current value of the spinner.
- **void setValue(Object value)**  
attempts to set the value of the spinner. Throws an `IllegalArgumentException` if the model does not accept the value.
- **void setEditor(JComponent editor)**  
sets the component used for editing the spinner value.

**javax.swing.SpinnerNumberModel 1.4**

- `SpinnerNumberModel(int initval, int minimum, int maximum, int stepSize)`
- `SpinnerNumberModel(double initval, double minimum, double maximum, double stepSize)`

These constructors yield number models that manage an `Integer` or `Double` value. Use the `MIN_VALUE` and `MAX_VALUE` constants of the `Integer` and `Double` classes for unbounded values.

|                    |                       |                                         |
|--------------------|-----------------------|-----------------------------------------|
| <i>Parameters:</i> | <code>initval</code>  | The initial value                       |
|                    | <code>minimum</code>  | The minimum valid value                 |
|                    | <code>maximum</code>  | The maximum valid value                 |
|                    | <code>stepSize</code> | The increment or decrement of each spin |

**javax.swing.SpinnerListModel 1.4**

- `SpinnerListModel(Object[] values)`
- `SpinnerListModel(List values)`

These constructors yield models that select a value from among the given values.

**javax.swing.SpinnerDateModel 1.4**

- `SpinnerDateModel()`  
constructs a date model with today's date as the initial value, no lower or upper bounds, and an increment of `Calendar.DAY_OF_MONTH`.
- `SpinnerDateModel(Date initval, Comparable minimum, Comparable maximum, int step)`

|                    |                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Parameters:</i> | <code>initval</code> | The initial value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|                    | <code>minimum</code> | The minimum valid value, or <code>null</code> if no lower bound is desired                                                                                                                                                                                                                                                                                                                                                                                                                             |
|                    | <code>maximum</code> | The maximum valid value, or <code>null</code> if no upper bound is desired                                                                                                                                                                                                                                                                                                                                                                                                                             |
|                    | <code>step</code>    | The date value to increment or decrement on each spin. One of the constants <code>ERA</code> , <code>YEAR</code> , <code>MONTH</code> , <code>WEEK_OF_YEAR</code> , <code>WEEK_OF_MONTH</code> , <code>DAY_OF_MONTH</code> , <code>DAY_OF_YEAR</code> , <code>DAY_OF_WEEK</code> , <code>DAY_OF_WEEK_IN_MONTH</code> , <code>AM_PM</code> , <code>HOUR</code> , <code>HOUR_OF_DAY</code> , <code>MINUTE</code> , <code>SECOND</code> , or <code>MILLISECOND</code> of the <code>Calendar</code> class. |

**java.text.SimpleDateFormat 1.1**

- `String toPattern() 1.2`

gets the editing pattern for this date formatter. A typical pattern is "yyyy-MM-dd". See the Java SE documentation for more details about the pattern.

**javax.swing.JSpinner.DateEditor 1.4**

- `DateEditor(JSpinner spinner, String pattern)`

constructs a date editor for a spinner.

*Parameters:*      `spinner`      The spinner to which this editor belongs  
                      `pattern`      The format pattern for the associated `SimpleDateFormat`

**javax.swing.AbstractSpinnerModel 1.4**

- `Object getValue()`

gets the current value of the model.

- `void setValue(Object value)`

attempts to set a new value for the model. Throws an `IllegalArgumentException` if the value is not acceptable. When overriding this method, you should call `fireStateChanged` after setting the new value.

- `Object getNextValue()`

- `Object getPreviousValue()`

computes (but does not set) the next or previous value in the sequence that this model defines.

#### 10.4.4 Displaying HTML with the JEditorPane

Unlike the text components discussed up to this point, the `JEditorPane` can display and edit styled text, in particular HTML and RTF. (RTF is the “Rich Text Format” used by a number of Microsoft applications for document interchange. It is a poorly documented format that doesn’t work well even between Microsoft’s own applications. We do not cover RTF capabilities in this book.)

Frankly, the `JEditorPane` is not as functional as one would like it to be. The HTML renderer can display simple files, but it chokes at many complex pages that you typically find on the Web. The HTML editor is limited and unstable.

A plausible application for the `JEditorPane` is to display program help in HTML format. By having control over the help files you provide, you can stay away from features that the `JEditorPane` does not display well.

---

**NOTE:** For more information on an industrial-strength help system, check out JavaHelp at <http://javahelp.java.net>.

---

The program in Listing 10.25 contains an editor pane that shows the contents of an HTML page. Type a URL into the text field. The URL must start with `http:` or `file:`. Then, click the Load button. The selected HTML page is displayed in the editor pane (see Figure 10.39).



Figure 10.39 The editor pane displaying an HTML page

The hyperlinks are active: If you click a link, the application loads it. The Back button returns to the previous page.

This program is in fact a very simple browser. Of course, it does not have any of the comfort features, such as page caching or bookmark lists, that you would expect from a real browser. The editor pane does not even display applets!

If you click the `Editable` checkbox, the editor pane becomes editable. You can type in text and use the Backspace key to delete text. The component also understands the `Ctrl+X`, `Ctrl+C`, and `Ctrl+V` shortcuts for cut, copy, and paste. However, you would have to do quite a bit of programming to add support for fonts and formatting.

When the component is editable, hyperlinks are not active. Also, with some web pages you can see JavaScript commands, comments, and other tags when edit mode is turned on (see Figure 10.40). The example program lets you investigate the editing feature, but we recommend that you omit it in your programs.

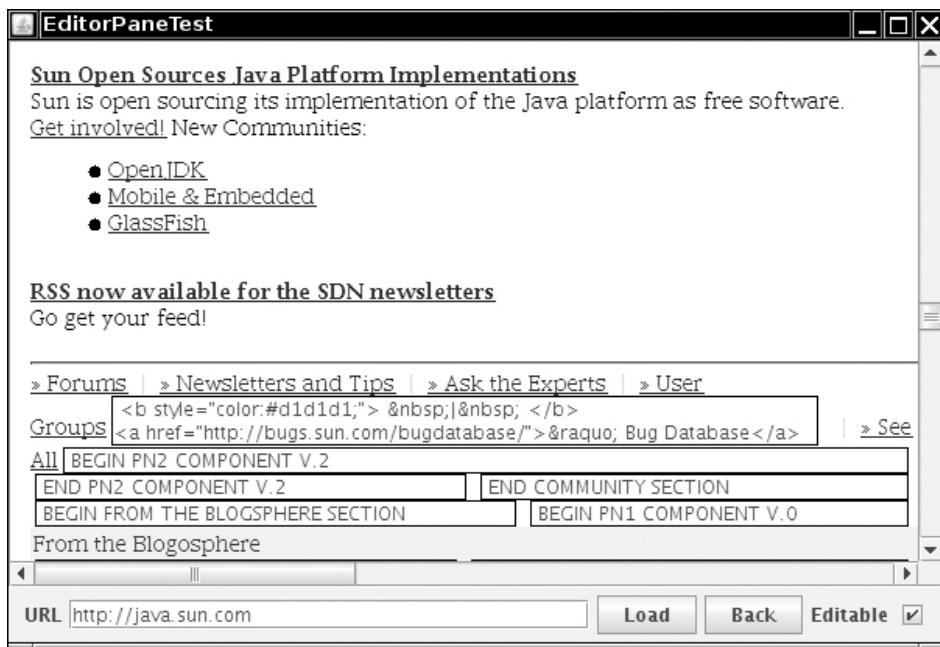


Figure 10.40 The editor pane in edit mode



**TIP:** By default, the `JEditorPane` is in edit mode. Call `editorPane.setEditable(false)` to turn it off.

---

The features of the editor pane that you saw in the example program are easy to use. The `setPage` method loads a new document. For example,

```
JEditorPane editorPane = new JEditorPane();
editorPane setPage(url);
```

The parameter is either a string or a `URL` object. The `JEditorPane` class extends the `JTextComponent` class. Therefore, you can call the `setText` method as well—it simply displays plain text.



**TIP:** The API documentation is unclear about whether `setPage` loads the new document in a separate thread (which is generally what you want—the `JEditorPane` is no speed demon). However, you can force loading in a separate thread with the following incantation:

```
AbstractDocument doc = (AbstractDocument) editorPane.getDocument();
doc.setAsynchronousLoadPriority(0);
```

To listen to hyperlink clicks, add a `HyperlinkListener`. The `HyperlinkListener` interface has a single method, `hyperlinkUpdate`, that is called when the user moves over or clicks on a link. The method has a parameter of type `HyperlinkEvent`.

You need to call the `getEventType` method to find out what kind of event occurred. There are three possible return values:

```
HyperlinkEvent.EventType.ACTIVATED
HyperlinkEvent.EventType.ENTERED
HyperlinkEvent.EventType.EXITED
```

The first value indicates that the user clicked on the hyperlink. In that case, you typically want to open the new link. You can use the second and third values to give some visual feedback, such as a tooltip, when the mouse hovers over the link.

---

**NOTE:** It is a complete mystery why there aren't three separate methods to handle activation, entry, and exit in the `HyperlinkListener` interface.

---

The `getURL` method of the `HyperlinkEvent` class returns the URL of the hyperlink. For example, here is how you can install a hyperlink listener that follows the links the user activated:

```
editorPane.addHyperlinkListener(event ->
{
 if (event.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
 {
 try
 {
 editorPane.setPage(event.getURL());
 }
 catch (IOException e)
 {
```

```
 editorPane.setText("Exception: " + e);
 }
}
});
```

The event handler simply gets the URL and updates the editor pane. The `setPage` method can throw an `IOException`. In that case, we display an error message as plain text.

The program in Listing 10.25 shows all the features that you need to put together an HTML help system. Under the hood, the `JEditorPane` is even more complex than the tree and table components. However, if you aren't writing a text editor or a renderer of a custom text format, that complexity is hidden from you.

---

**Listing 10.25** `editorPane/EditorPaneFrame.java`

---

```
1 package editorPane;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.util.*;
7 import javax.swing.*;
8 import javax.swing.event.*;
9
10 /**
11 * This frame contains an editor pane, a text field and button to enter a URL and load a document,
12 * and a Back button to return to a previously loaded document.
13 */
14 public class EditorPaneFrame extends JFrame
15 {
16 private static final int DEFAULT_WIDTH = 600;
17 private static final int DEFAULT_HEIGHT = 400;
18
19 public EditorPaneFrame()
20 {
21 setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
22
23 final Stack<String> urlStack = new Stack<>();
24 final JEditorPane editorPane = new JEditorPane();
25 final JTextField url = new JTextField(30);
26
27 // set up hyperlink listener
28
29 editorPane.setEditable(false);
30 editorPane.addHyperlinkListener(event ->
31 {
32 if (event.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
33 {
```

```
34 try
35 {
36 // remember URL for back button
37 urlStack.push(event.getURL().toString());
38 // show URL in text field
39 url.setText(event.getURL().toString());
40 editorPane.setPage(event.getURL());
41 }
42 catch (IOException e)
43 {
44 editorPane.setText("Exception: " + e);
45 }
46 }
47);
48
49 // set up checkbox for toggling edit mode
50
51 final JCheckBox editable = new JCheckBox();
52 editable.addActionListener(event ->
53 editorPane.setEditable(editable.isSelected()));
54
55 // set up load button for loading URL
56
57 ActionListener listener = event ->
58 {
59 try
60 {
61 // remember URL for back button
62 urlStack.push(url.getText());
63 editorPane.setPage(url.getText());
64 }
65 catch (IOException e)
66 {
67 editorPane.setText("Exception: " + e);
68 }
69 };
70
71 JButton loadButton = new JButton("Load");
72 loadButton.addActionListener(listener);
73 url.addActionListener(listener);
74
75 // set up back button and button action
76
77 JButton backButton = new JButton("Back");
78 backButton.addActionListener(event ->
79 {
80 if (urlStack.size() <= 1) return;
81 try
82 {
```

(Continues)

**Listing 10.25 (Continued)**

```
83 // get URL from back button
84 urlStack.pop();
85 // show URL in text field
86 urlString = urlStack.peek();
87 url.setText(urlString);
88 editorPane.setPage(urlString);
89 }
90 catch (IOException e)
91 {
92 editorPane.setText("Exception: " + e);
93 }
94 });
95
96 add(new JScrollPane(editorPane), BorderLayout.CENTER);
97
98 // put all control components in a panel
99
100 JPanel panel = new JPanel();
101 panel.add(new JLabel("URL"));
102 panel.add(url);
103 panel.add(loadButton);
104 panel.add(backButton);
105 panel.add(new JLabel("Editable"));
106 panel.add(editable);
107
108 add(panel, BorderLayout.SOUTH);
109 }
110 }
```

---

***javax.swing.JEditorPane 1.2***

- `void setPage(URL url)`  
loads the page from `url` into the editor pane.
- `void addHyperlinkListener(HyperLinkListener listener)`  
adds a hyperlink listener to this editor pane.

***javax.swing.event.HyperLinkListener 1.2***

- `void hyperlinkUpdate(HyperlinkEvent event)`  
is called whenever a hyperlink was selected.

**javax.swing.event.HyperlinkEvent 1.2**

- URL getURL()  
returns the URL of the selected hyperlink.

## 10.5 Progress Indicators

In the following sections, we discuss three classes for indicating the progress of a slow activity. A `JProgressBar` is a Swing component that indicates progress. A `ProgressMonitor` is a dialog box that contains a progress bar. A `ProgressMonitorInputStream` displays a progress monitor dialog box while the stream is read.

### 10.5.1 Progress Bars

A *progress bar* is a simple component—just a rectangle that is partially filled with color to indicate the progress of an operation. By default, progress is indicated by a string “*n%*”. You can see a progress bar in the bottom right of Figure 10.41.



Figure 10.41 A progress bar

You can construct a progress bar much as you construct a slider—by supplying the minimum and maximum value and an optional orientation:

```
progressBar = new JProgressBar(0, 1000);
progressBar = new JProgressBar(SwingConstants.VERTICAL, 0, 1000);
```

You can also set the minimum and maximum with the `setMinimum` and `setMaximum` methods.

Unlike a slider, the progress bar cannot be adjusted by the user. Your program needs to call `setValue` to update it.

If you call

```
progressBar.setStringPainted(true);
```

the progress bar computes the completion percentage and displays a string “*n%*”. If you want to show a different string, you can supply it with the `setString` method:

```
if (progressBar.getValue() > 900)
 progressBar.setString("Almost Done");
```

The program in Listing 10.26 shows a progress bar that monitors a simulated time-consuming activity.

The `SimulatedActivity` class increments a value `current` ten times per second. When it reaches a target value, the activity finishes. We use the `SwingWorker` class to implement the activity and update the progress bar in the `process` method. The `SwingWorker` invokes the method from the event dispatch thread, so it is safe to update the progress bar. (See Volume I, Chapter 14 for more information about thread safety in Swing.)

Java SE 1.4 added support for an *indeterminate* progress bar that shows an animation indicating some kind of progress, without specifying the percentage of completion. That is the kind of progress bar that you see in your browser—it indicates that the browser is waiting for the server and has no idea how long the wait might be. To display the “indeterminate wait” animation, call the `setIndeterminate` method.

Listing 10.26 shows the full program code.

---

**Listing 10.26** `progressBar/ProgressBarFrame.java`

---

```
1 package progressBar;
2
3 import java.awt.*;
4 import java.util.List;
5
6 import javax.swing.*;
7
8 /**
9 * A frame that contains a button to launch a simulated activity, a progress bar, and a text area
10 * for the activity output.
11 */
12 public class ProgressBarFrame extends JFrame
13 {
14 public static final int TEXT_ROWS = 10;
15 public static final int TEXT_COLUMNS = 40;
16
17 private JButton startButton;
18 private JProgressBar progressBar;
19 private JCheckBox checkBox;
```

```
20 private JTextArea textArea;
21 private SimulatedActivity activity;
22
23 public ProgressBarFrame()
24 {
25 // this text area holds the activity output
26 textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
27
28 // set up panel with button and progress bar
29
30 final int MAX = 1000;
31 JPanel panel = new JPanel();
32 startButton = new JButton("Start");
33 progressBar = new JProgressBar(0, MAX);
34 progressBar.setStringPainted(true);
35 panel.add(startButton);
36 panel.add(progressBar);
37
38 checkBox = new JCheckBox("indeterminate");
39 checkBox.addActionListener(event ->
40 {
41 progressBar.setIndeterminate(checkBox.isSelected());
42 progressBar.setStringPainted(!progressBar.isIndeterminate());
43 });
44 panel.add(checkBox);
45 add(new JScrollPane(textArea), BorderLayout.CENTER);
46 add(panel, BorderLayout.SOUTH);
47
48 // set up the button action
49
50 startButton.addActionListener(event ->
51 {
52 startButton.setEnabled(false);
53 activity = new SimulatedActivity(MAX);
54 activity.execute();
55 });
56 pack();
57 }
58
59 class SimulatedActivity extends SwingWorker<Void, Integer>
60 {
61 private int current;
62 private int target;
63
64 /**
65 * Constructs the simulated activity that increments a counter from 0 to a
66 * given target.
67 * @param t the target value of the counter
68 */
69 }
```

(Continues)

**Listing 10.26 (Continued)**

```
69 public SimulatedActivity(int t)
70 {
71 current = 0;
72 target = t;
73 }
74
75 protected Void doInBackground() throws Exception
76 {
77 try
78 {
79 while (current < target)
80 {
81 Thread.sleep(100);
82 current++;
83 publish(current);
84 }
85 }
86 catch (InterruptedException e)
87 {
88 }
89 return null;
90 }
91
92 protected void process(List<Integer> chunks)
93 {
94 for (Integer chunk : chunks)
95 {
96 textArea.append(chunk + "\n");
97 progressBar.setValue(chunk);
98 }
99 }
100
101 protected void done()
102 {
103 startButton.setEnabled(true);
104 }
105}
106}
```

---

## 10.5.2 Progress Monitors

A progress bar is a simple component that can be placed inside a window. In contrast, a `ProgressMonitor` is a complete dialog box that contains a progress bar (see Figure 10.42). The dialog box contains a Cancel button. If you click it, the monitor dialog box is closed. In addition, your program can query whether the user has

canceled the dialog box and terminate the monitored action. (Note that the class name does not start with a “J”.)

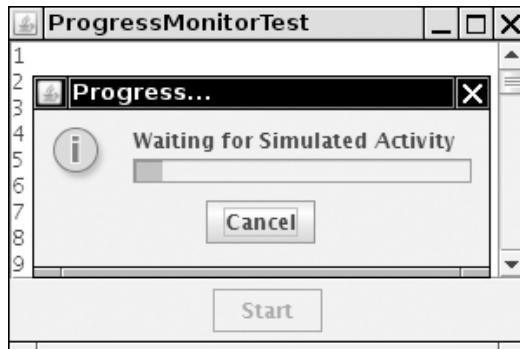


Figure 10.42 A progress monitor dialog box

Construct a progress monitor by supplying the following:

- The parent component over which the dialog box should pop up
- An object (which should be a string, icon, or component) that is displayed in the dialog box
- An optional note to display below the object
- The minimum and maximum values

However, the progress monitor cannot measure progress or cancel an activity by itself. You still need to periodically set the progress value by calling the `setProgress` method. (This is the equivalent of the `setValue` method of the `JProgressBar` class.) When the monitored activity has concluded, call the `close` method to dismiss the dialog box. You can reuse the same dialog box by calling `start` again.

The biggest problem with using a progress monitor dialog box is handling the cancellation requests. You cannot attach an event handler to the Cancel button. Instead, you need to periodically call the `isCanceled` method to see if the user has clicked the Cancel button.

If your worker thread can block indefinitely (for example, when reading input from a network connection), it cannot monitor the Cancel button. In our sample program, we will show you how to use a timer for that purpose. We will also make the timer responsible for updating the progress measurement.

If you run the program in Listing 10.27, you can observe an interesting feature of the progress monitor dialog box. The dialog box doesn’t come up immediately.

Instead, it waits for a short interval to see if the activity has already been completed or is likely to complete in less time than it would take for the dialog box to appear.

Use the `setMillisToDecideToPopup` method to set the number of milliseconds to wait between the construction of the dialog object and the decision whether to show the pop-up at all. The default value is 500 milliseconds. The `setMillisToPopup` is your estimation of the time the dialog box needs to pop up. The Swing designers set this value to a default of 2 seconds. Clearly they were mindful of the fact that Swing dialogs don't always come up as snappily as we all would like. You should probably not touch this value.

---

**Listing 10.27** progressMonitor/ProgressMonitorFrame.java

---

```
1 package progressMonitor;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8 * A frame that contains a button to launch a simulated activity and a text area for the activity
9 * output.
10 */
11 class ProgressMonitorFrame extends JFrame
12 {
13 public static final int TEXT_ROWS = 10;
14 public static final int TEXT_COLUMNS = 40;
15
16 private Timer cancelMonitor;
17 private JButton startButton;
18 private ProgressMonitor progressDialog;
19 private JTextArea textArea;
20 private SimulatedActivity activity;
21
22 public ProgressMonitorFrame()
23 {
24 // this text area holds the activity output
25 textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
26
27 // set up a button panel
28 JPanel panel = new JPanel();
29 startButton = new JButton("Start");
30 panel.add(startButton);
31
32 add(new JScrollPane(textArea), BorderLayout.CENTER);
33 add(panel, BorderLayout.SOUTH);
34
35 // set up the button action
```

```
36 startButton.addActionListener(event ->
37 {
38 startButton.setEnabled(false);
39 final int MAX = 1000;
40
41 // start activity
42 activity = new SimulatedActivity(MAX);
43 activity.execute();
44
45 // launch progress dialog
46 progressDialog = new ProgressMonitor(ProgressMonitorFrame.this,
47 "Waiting for Simulated Activity", null, 0, MAX);
48 cancelMonitor.start();
49 });
50
51
52 // set up the timer action
53
54 cancelMonitor = new Timer(500, event ->
55 {
56 if (progressDialog.isCanceled())
57 {
58 activity.cancel(true);
59 startButton.setEnabled(true);
60 }
61 else if (activity.isDone())
62 {
63 progressDialog.close();
64 startButton.setEnabled(true);
65 }
66 else
67 {
68 progressDialog.setProgress(activity.getProgress());
69 }
70 });
71 pack();
72 }
73
74 class SimulatedActivity extends SwingWorker<Void, Integer>
75 {
76 private int current;
77 private int target;
78
79 /**
80 * Constructs the simulated activity that increments a counter from 0 to a
81 * given target.
82 * @param t the target value of the counter
83 */
```

(Continues)

**Listing 10.27 (Continued)**

```
84 public SimulatedActivity(int t)
85 {
86 current = 0;
87 target = t;
88 }
89
90 protected Void doInBackground() throws Exception
91 {
92 try
93 {
94 while (current < target)
95 {
96 Thread.sleep(100);
97 current++;
98 textArea.append(current + "\n");
99 setProgress(current);
100 }
101 }
102 catch (InterruptedException e)
103 {
104 }
105 return null;
106 }
107}
108}
```

---

### 10.5.3 Monitoring the Progress of Input Streams

The Swing package contains a useful stream filter, `ProgressMonitorInputStream`, that automatically pops up a dialog box that monitors how much of the stream has been read.

This filter is extremely easy to use. Insert a `ProgressMonitorInputStream` into your usual sequence of filtered streams. (See Chapter 2 for more information on streams.)

For example, suppose you read text from a file. You start out with a `FileInputStream`:

```
FileInputStream in = new FileInputStream(f);
```

Normally, you would convert `in` to an `InputStreamReader`:

```
InputStreamReader reader = new InputStreamReader(in);
```

However, to monitor the stream, first turn the file input stream into a stream with a progress monitor:

```
ProgressMonitorInputStream progressIn = new ProgressMonitorInputStream(parent, caption, in);
```

Supply the parent component, a caption, and, of course, the stream to monitor. The `read` method of the progress monitor stream simply passes along the bytes and updates the progress dialog box.

You can now go on building your filter sequence:

```
InputStreamReader reader = new InputStreamReader(progressIn);
```

That's all there is to it. When the file is being read, the progress monitor automatically pops up (see Figure 10.43). This is a very nice application of stream filtering.

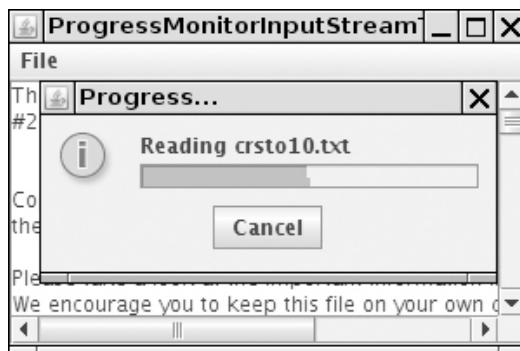


Figure 10.43 A progress monitor for an input stream



**CAUTION:** The progress monitor stream uses the `available` method of the `InputStream` class to determine the total number of bytes in the stream. However, the `available` method only reports the number of bytes in the stream that are available *without blocking*. Progress monitors work well for files and HTTP URLs because their length is known in advance, but they don't work with all streams.

The program in Listing 10.28 counts the lines in a file. If you read in a large file (such as "The Count of Monte Cristo" in the gutenberg directory of the companion code), the progress dialog box pops up.

If the user clicks the Cancel button, the input stream closes. The code that processes the input already knows how to deal with the end of input, so no change to the programming logic is required to handle cancellation.

Note that the program doesn't use a very efficient way of filling up the text area. It would be faster to first read the file into a `StringBuilder` and then set the text of the text area to the string builder contents. However, in this example program,

we actually like this slow approach—it gives you more time to admire the progress dialog box.

To avoid flicker, we do not display the text area while it is filling up.

---

**Listing 10.28** `progressMonitorInputStream/TextFrame.java`

---

```
1 package progressMonitorInputStream;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.util.*;
6
7 import javax.swing.*;
8
9 /**
10 * A frame with a menu to load a text file and a text area to display its contents. The text
11 * area is constructed when the file is loaded and set as the content pane of the frame when
12 * the loading is complete. That avoids flicker during loading.
13 */
14 public class TextFrame extends JFrame
15 {
16 public static final int TEXT_ROWS = 10;
17 public static final int TEXT_COLUMNS = 40;
18
19 private JMenuItem openItem;
20 private JMenuItem exitItem;
21 private JTextArea textArea;
22 private JFileChooser chooser;
23
24 public TextFrame()
25 {
26 textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
27 add(new JScrollPane(textArea));
28
29 chooser = new JFileChooser();
30 chooser.setCurrentDirectory(new File("."));
31
32 JMenuBar menuBar = new JMenuBar();
33 setJMenuBar(menuBar);
34 JMenu fileMenu = new JMenu("File");
35 menuBar.add(fileMenu);
36 openItem = new JMenuItem("Open");
37 openItem.addActionListener(event ->
38 {
39 try
40 {
41 openFile();
42 }
43 }
44);
45 }
46
47 void openFile()
48 {
49 File file = chooser.showOpenDialog(this);
50 if (file != null)
51 {
52 String content = new String(Files.readAllBytes(file.toPath()));
53 textArea.setText(content);
54 }
55 }
56 }
```

```
43 catch (IOException exception)
44 {
45 exception.printStackTrace();
46 }
47);
48
49 fileMenu.add(openItem);
50 exitItem = new JMenuItem("Exit");
51 exitItem.addActionListener(event -> System.exit(0));
52 fileMenu.add(exitItem);
53 pack();
54 }
55
56 /**
57 * Prompts the user to select a file, loads the file into a text area, and sets it as the
58 * content pane of the frame.
59 */
60 public void openFile() throws IOException
61 {
62 int r = chooser.showOpenDialog(this);
63 if (r != JFileChooser.APPROVE_OPTION) return;
64 final File f = chooser.getSelectedFile();
65
66 // set up stream and reader filter sequence
67
68 InputStream fileIn = Files.newInputStream(f.toPath());
69 final ProgressMonitorInputStream progressIn = new ProgressMonitorInputStream(
70 this, "Reading " + f.getName(), fileIn);
71
72 textArea.setText("");
73
74 SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>()
75 {
76 protected Void doInBackground() throws Exception
77 {
78 try (Scanner in = new Scanner(progressIn, "UTF-8"))
79 {
80 while (in.hasNextLine())
81 {
82 String line = in.nextLine();
83 textArea.append(line);
84 textArea.append("\n");
85 }
86 }
87 return null;
88 }
89 };
90 worker.execute();
91 }
92 }
```

**javax.swing.JProgressBar 1.2**

- `JProgressBar()`
- `JProgressBar(int direction)`
- `JProgressBar(int min, int max)`
- `JProgressBar(int direction, int min, int max)`

constructs a slider with the given direction, minimum, and maximum.

*Parameters:*      `direction`      One of `SwingConstants.HORIZONTAL` or `SwingConstants.VERTICAL`.  
The default is horizontal.  
  
                  `min, max`      The minimum and maximum for the progress bar  
values. Defaults are 0 and 100.

- `int getMinimum()`
- `int getMaximum()`
- `void setMinimum(int value)`
- `void setMaximum(int value)`

gets or sets the minimum and maximum values.

- `int getValue()`
- `void setValue(int value)`

gets or sets the current value.

- `String getString()`
- `void setString(String s)`

gets or sets the string to be displayed in the progress bar. If the string is `null`, a default string “*n%*” is displayed.

- `boolean isStringPainted()`
- `void setStringPainted(boolean b)`

gets or sets the “string painted” property. If this property is `true`, a string is painted on top of the progress bar. The default is `false`.

- `boolean isIndeterminate() 1.4`
- `void setIndeterminate(boolean b) 1.4`

gets or sets the “indeterminate” property. If this property is `true`, the progress bar becomes a block that moves backward and forward, indicating a wait of unknown duration. The default is `false`.

**javax.swing.ProgressMonitor 1.2**

- `ProgressMonitor(Component parent, Object message, String note, int min, int max)`

constructs a progress monitor dialog box.

|                    |                       |                                                                                                                                                                                    |
|--------------------|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Parameters:</i> | <code>parent</code>   | The parent component over which this dialog box pops up                                                                                                                            |
|                    | <code>message</code>  | The message object to display in the dialog box                                                                                                                                    |
|                    | <code>note</code>     | The optional string to display under the message. If this value is <code>null</code> , no space is set aside for the note, and a later call to <code>setNote</code> has no effect. |
|                    | <code>min, max</code> | The minimum and maximum values of the progress bar                                                                                                                                 |

- `void setNote(String note)`  
changes the note text.
- `void setProgress(int value)`  
sets the progress bar value to the given value.
- `void close()`  
closes this dialog box.
- `boolean isCanceled()`  
returns true if the user canceled this dialog box.

**javax.swing.ProgressMonitorInputStream 1.2**

- `ProgressMonitorInputStream(Component parent, Object message, InputStream in)`

constructs an input stream filter with an associated progress monitor dialog box.

|                    |                      |                                                         |
|--------------------|----------------------|---------------------------------------------------------|
| <i>Parameters:</i> | <code>parent</code>  | The parent component over which this dialog box pops up |
|                    | <code>message</code> | The message object to display in the dialog box         |
|                    | <code>in</code>      | The input stream that is being monitored                |

## 10.6 Component Organizers and Decorators

We conclude the discussion of advanced Swing features with a presentation of components that help organize other components. These include the *split pane*, a mechanism for splitting an area into multiple parts with boundaries that can be adjusted; the *tabbed pane* which uses tab dividers to allow a user to flip through

multiple panels; and the *desktop pane* that can be used to implement applications displaying multiple *internal frames*. We will close with a discussion of *layers*—decorators that can be superimposed over other components.

### 10.6.1 Split Panes

A split pane splits a component into two parts, with an adjustable boundary in between. Figure 10.44 shows a frame with two split panes. The components in the outer split pane are arranged vertically, with a text area on the bottom and another split pane on the top. That split pane's components are arranged horizontally, with a list on the left and a label containing an image on the right.

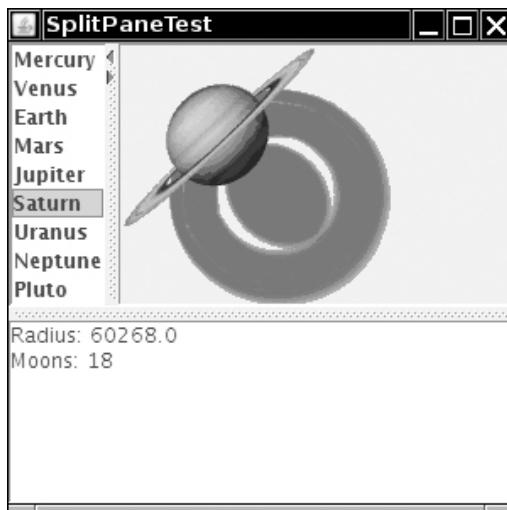


Figure 10.44 A frame with two nested split panes

Construct a split pane by specifying the orientation—one of `JSplitPane.HORIZONTAL_SPLIT` or `JSplitPane.VERTICAL_SPLIT`, followed by the two components. For example,

```
JSplitPane innerPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, planetList, planetImage);
```

That's all you have to do. If you like, you can add “one-touch expand” icons to the splitter bar. You can see those icons in the top pane in Figure 10.44. In the Metal look-and-feel, they are small triangles. If you click one of them, the splitter moves all the way in the direction to which the triangle is pointing, expanding one of the panes completely.

To add this capability, call

```
innerPane.setOneTouchExpandable(true);
```

The “continuous layout” feature continuously repaints the contents of both components as the user adjusts the splitter. That looks classier, but it can be slow. Turn on that feature with the call

```
innerPane.setContinuousLayout(true);
```

In the example program, we left the bottom splitter at the default (no continuous layout). When you drag it, you only move a black outline. When you release the mouse, the components are repainted.

The straightforward program in Listing 10.29 populates a list box with planets. When the user makes a selection, the planet image is displayed to the right and a description is placed in the text area on the bottom. Run the program, adjust the splitters, and try out the one-touch expansion and continuous layout features.

---

**Listing 10.29** splitPane/SplitPaneFrame.java

---

```
1 package splitPane;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8 * This frame consists of two nested split panes to demonstrate planet images and data.
9 */
10 class SplitPaneFrame extends JFrame
11 {
12 private static final int DEFAULT_WIDTH = 300;
13 private static final int DEFAULT_HEIGHT = 300;
14
15 private Planet[] planets = { new Planet("Mercury", 2440, 0), new Planet("Venus", 6052, 0),
16 new Planet("Earth", 6378, 1), new Planet("Mars", 3397, 2),
17 new Planet("Jupiter", 71492, 16), new Planet("Saturn", 60268, 18),
18 new Planet("Uranus", 25559, 17), new Planet("Neptune", 24766, 8),
19 new Planet("Pluto", 1137, 1), };
20
21 public SplitPaneFrame()
22 {
23 setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
24
25 // set up components for planet names, images, descriptions
26
27 final JList<Planet> planetList = new JList<>(planets);
28 final JLabel planetImage = new JLabel();
29 final JTextArea planetDescription = new JTextArea();
```

(Continues)

**Listing 10.29 (Continued)**

```

30
31 planetList.addListSelectionListener(event ->
32 {
33 Planet value = (Planet) planetList.getSelectedValue();
34
35 // update image and description
36
37 planetImage.setIcon(value.getImage());
38 planetDescription.setText(value.getDescription());
39 });
40
41 // set up split panes
42
43 JSplitPane innerPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, planetList, planetImage);
44
45 innerPane.setContinuousLayout(true);
46 innerPane.setOneTouchExpandable(true);
47
48 JSplitPane outerPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT, innerPane,
49 planetDescription);
50
51 add(outerPane, BorderLayout.CENTER);
52 }
53 }
```

**javax.swing.JSplitPane 1.2**

- `JSplitPane()`
  - `JSplitPane(int direction)`
  - `JSplitPane(int direction, boolean continuousLayout)`
  - `JSplitPane(int direction, Component first, Component second)`
  - `JSplitPane(int direction, boolean continuousLayout, Component first, Component second)`
- constructs a new split pane.

*Parameters:*      `direction`

One of `HORIZONTAL_SPLIT` or `VERTICAL_SPLIT`

`continuousLayout`

true if the components are continuously updated  
when the splitter is moved

`first, second`

The components to add

- `boolean isOneTouchExpandable()`
- `void setOneTouchExpandable(boolean b)`

gets or sets the “one-touch expandable” property. When this property is set, the splitter has two icons to completely expand one or the other component.

(Continues)

**javax.swing.JSplitPane 1.2 (Continued)**

- boolean isContinuousLayout()
- void setContinuousLayout(boolean b)

gets or sets the “continuous layout” property. When this property is set, the components are continuously updated when the splitter is moved.

- void setLeftComponent(Component c)
- void setTopComponent(Component c)

These operations have the same effect, setting c as the first component in the split pane.

- void setRightComponent(Component c)
- void setBottomComponent(Component c)

These operations have the same effect, setting c as the second component in the split pane.

## 10.6.2 Tabbed Panes

Tabbed panes are a familiar user interface device to break up a complex dialog box into subsets of related options. You can also use tabs to let a user flip through a set of documents or images (see Figure 10.45). That is what we do in our sample program.

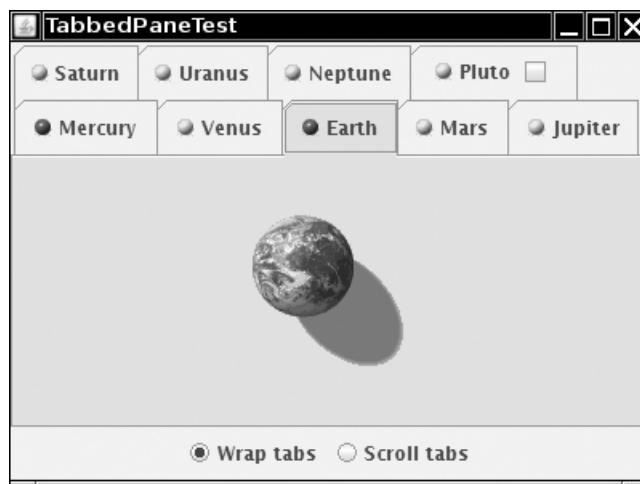


Figure 10.45 A tabbed pane

To create a tabbed pane, first construct a `JTabbedPane` object, then add tabs to it.

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.addTab(title, icon, component);
```

The last parameter of the `addTab` method has type `Component`. To add multiple components into the same tab, first pack them up in a container, such as a `JPanel`.

The icon is optional; for example, the `addTab` method does not require an icon:

```
tabbedPane.addTab(title, component);
```

You can also add a tab in the middle of the tab collection with the `insertTab` method:

```
tabbedPane.insertTab(title, icon, component, tooltip, index);
```

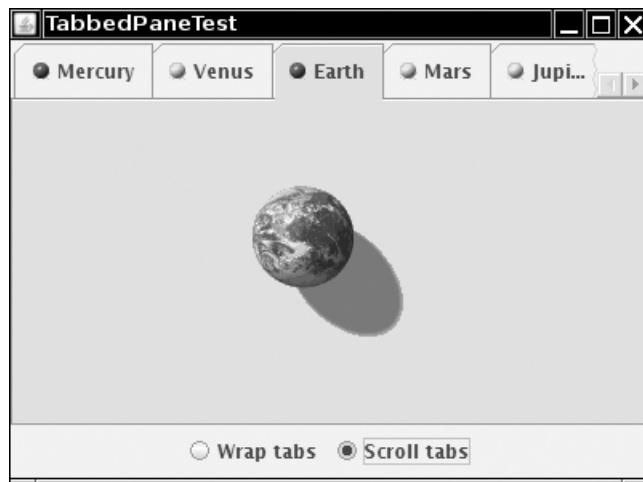
To remove a tab from the tab collection, use

```
tabPane.removeTabAt(index);
```

When you add a new tab to the tab collection, it is not automatically displayed. You must select it with the `setSelectedIndex` method. For example, here is how you show a tab that you just added to the end:

```
tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);
```

If you have a lot of tabs, they can take up quite a bit of space. Starting with Java SE 1.4, you can display the tabs in scrolling mode, in which only one row of tabs is displayed, together with a set of arrow buttons that allow the user to scroll through the tab set (see Figure 10.46).



**Figure 10.46** A tabbed pane with scrolling tabs

Set the tab layout to wrapped or scrolling mode by calling

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
```

or

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
```

The tab labels can have mnemonics, just like menu items. For example,

```
int marsIndex = tabbedPane.indexOfTab("Mars");
tabbedPane.setMnemonicAt(marsIndex, KeyEvent.VK_M);
```

Now the M is underlined, and users can select the tab by pressing Alt+M.

You can add arbitrary components into the tab titles. First, add the tab, then call

```
tabbedPane.setTabComponentAt(index, component);
```

In our sample program, we add a “close box” to the Pluto tab (because, after all, astronomers do not consider Pluto a planet). This is achieved by setting the tab component to a panel containing two components: a label with the icon and tab text, and a checkbox with an action listener that removes the tab.

The example program shows a useful technique with tabbed panes. Sometimes, you may want to update a component just before it is displayed. Here, we load the planet image only when the user actually clicks a tab.

To be notified whenever the user clicks on a tab, install a `ChangeListener` with the tabbed pane. Note that you must install the listener with the tabbed pane itself, not with any of the components.

```
tabbedPane.addChangeListener(listener);
```

When the user selects a tab, the `stateChanged` method of the change listener is called. You can retrieve the tabbed pane as the source of the event. Call the `getSelectedIndex` method to find out which pane is about to be displayed.

```
public void stateChanged(ChangeEvent event)
{
 int n = tabbedPane.getSelectedIndex();
 loadTab(n);
}
```

In Listing 10.30, we first set all tab components to `null`. When a new tab is selected, we test whether its component is still `null`. If so, we replace it with the image. (This happens instantaneously when you click on the tab. You will not see an empty pane.) Just for fun, we also change the icon from a yellow ball to a red ball to indicate which panes have been visited.

**Listing 10.30** tabbedPane/TabbedPaneFrame.java

```
1 package tabbedPane;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8 * This frame shows a tabbed pane and radio buttons to switch between wrapped and scrolling tab
9 * layout.
10 */
11 public class TabbedPaneFrame extends JFrame
12 {
13 private static final int DEFAULT_WIDTH = 400;
14 private static final int DEFAULT_HEIGHT = 300;
15
16 private JTabbedPane tabbedPane;
17
18 public TabbedPaneFrame()
19 {
20 setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
21
22 tabbedPane = new JTabbedPane();
23 // we set the components to null and delay their loading until the tab is shown
24 // for the first time
25
26 ImageIcon icon = new ImageIcon(getClass().getResource("yellow-ball.gif"));
27
28 tabbedPane.addTab("Mercury", icon, null);
29 tabbedPane.addTab("Venus", icon, null);
30 tabbedPane.addTab("Earth", icon, null);
31 tabbedPane.addTab("Mars", icon, null);
32 tabbedPane.addTab("Jupiter", icon, null);
33 tabbedPane.addTab("Saturn", icon, null);
34 tabbedPane.addTab("Uranus", icon, null);
35 tabbedPane.addTab("Neptune", icon, null);
36 tabbedPane.addTab("Pluto", null, null);
37
38 final int plutoIndex = tabbedPane.indexOfTab("Pluto");
39 JPanel plutoPanel = new JPanel();
40 plutoPanel.add(new JLabel("Pluto", icon, SwingConstants.LEADING));
41 JToggleButton plutoCheckBox = new JCheckBox();
42 plutoCheckBox.addActionListener(event -> tabbedPane.remove(plutoIndex));
43 plutoPanel.add(plutoCheckBox);
44 tabbedPane.setTabComponentAt(plutoIndex, plutoPanel);
45
46 add(tabbedPane, "Center");
```

```
47 tabbedPane.addChangeListener(event ->
48 {
49 // check if this tab still has a null component
50 if (tabbedPane.getSelectedComponent() == null)
51 {
52 // set the component to the image icon
53
54 int n = tabbedPane.getSelectedIndex();
55 loadTab(n);
56 }
57 });
58
59 loadTab(0);
60
61
62 JPanel buttonPanel = new JPanel();
63 ButtonGroup buttonGroup = new ButtonGroup();
64 JRadioButton wrapButton = new JRadioButton("Wrap tabs");
65 wrapButton.addActionListener(event ->
66 tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT));
67 buttonPanel.add(wrapButton);
68 buttonGroup.add(wrapButton);
69 wrapButton.setSelected(true);
70 JRadioButton scrollButton = new JRadioButton("Scroll tabs");
71 scrollButton.addActionListener(event ->
72 tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT));
73 buttonPanel.add(scrollButton);
74 buttonGroup.add(scrollButton);
75 add(buttonPanel, BorderLayout.SOUTH);
76 }
77
78 /**
79 * Loads the tab with the given index.
80 * @param n the index of the tab to load
81 */
82 private void loadTab(int n)
83 {
84 String title = tabbedPane.getTitleAt(n);
85 ImageIcon planetIcon = new ImageIcon(getClass().getResource(title + ".gif"));
86 tabbedPane.setComponentAt(n, new JLabel(planetIcon));
87
88 // indicate that this tab has been visited--just for fun
89
90 tabbedPane.setIconAt(n, new ImageIcon(getClass().getResource("red-ball.gif")));
91 }
92 }
93 }
```

**javax.swing.JTabbedPane 1.2**

- `JTabbedPane()`
- `JTabbedPane(int placement)`

constructs a tabbed pane.

*Parameters:*      `placement`                  One of `SwingConstants.TOP`, `SwingConstants.LEFT`,  
`SwingConstants.RIGHT`, or `SwingConstants.BOTTOM`

- `void addTab(String title, Component c)`
- `void addTab(String title, Icon icon, Component c)`
- `void addTab(String title, Icon icon, Component c, String tooltip)`  
adds a tab to the end of the tabbed pane.
- `void insertTab(String title, Icon icon, Component c, String tooltip, int index)`  
inserts a tab to the tabbed pane at the given index.
- `void removeTabAt(int index)`  
removes the tab at the given index.
- `void setSelectedIndex(int index)`  
selects the tab at the given index.
- `int getSelectedIndex()`  
returns the index of the selected tab.
- `Component getSelectedComponent()`  
returns the component of the selected tab.
- `String getTitleAt(int index)`
- `void setTitleAt(int index, String title)`
- `Icon getIconAt(int index)`
- `void setIconAt(int index, Icon icon)`
- `Component getComponentAt(int index)`
- `void setComponentAt(int index, Component c)`  
gets or sets the title, icon, or component at the given index.
- `int indexOfTab(String title)`
- `int indexOfTab(Icon icon)`
- `int indexOfComponent(Component c)`  
returns the index of the tab with the given title, icon, or component.
- `int getTabCount()`  
returns the total number of tabs in this tabbed pane.

(Continues)

**javax.swing.JTabbedPane 1.2 (Continued)**

- `int getTabLayoutPolicy()`
- `void setTabLayoutPolicy(int policy) 1.4`

gets or sets the tab layout policy. `policy` is one of `JTabbedPane.WRAP_TAB_LAYOUT` or `JTabbedPane.SCROLL_TAB_LAYOUT`.

- `int getMnemonicAt(int index) 1.4`
- `void setMnemonicAt(int index, int mnemonic)`

gets or sets the mnemonic character at a given tab index. The character is specified as a `VK_X` constant from the `KeyEvent` class. -1 means that there is no mnemonic.

- `Component getTabComponentAt(int index) 6`
- `void setTabComponentAt(int index, Component c) 6`

gets or sets the component that renders the title of the tab with the given index. If this component is `null`, the tab icon and title are rendered. Otherwise, only the given component is rendered in the tab.

- `int indexOfTabComponent(Component c) 6`  
returns the index of the tab with the given title component.
- `void addChangeListener(ChangeListener listener)`  
adds a change listener that is notified when the user selects a different tab.

### 10.6.3 Desktop Panes and Internal Frames

Many applications present information in multiple windows that are all contained inside a large frame. If you minimize the application frame, all of its windows are hidden at the same time. In the Windows environment, this user interface is sometimes called the *multiple document interface* (MDI). Figure 10.47 shows a typical application using this interface.

For a time, this user interface style was popular, but it has become less prevalent in recent years. Nowadays, many applications simply display a separate top-level frame for each document. Which is better? MDI reduces window clutter, but having separate top-level windows means that you can use the buttons and hotkeys of the host windowing system to flip through your windows.

#### 10.6.3.1 Displaying Internal Frames

In the world of Java, where you can't rely on a rich host windowing system, it makes a lot of sense to have your application manage its frames.

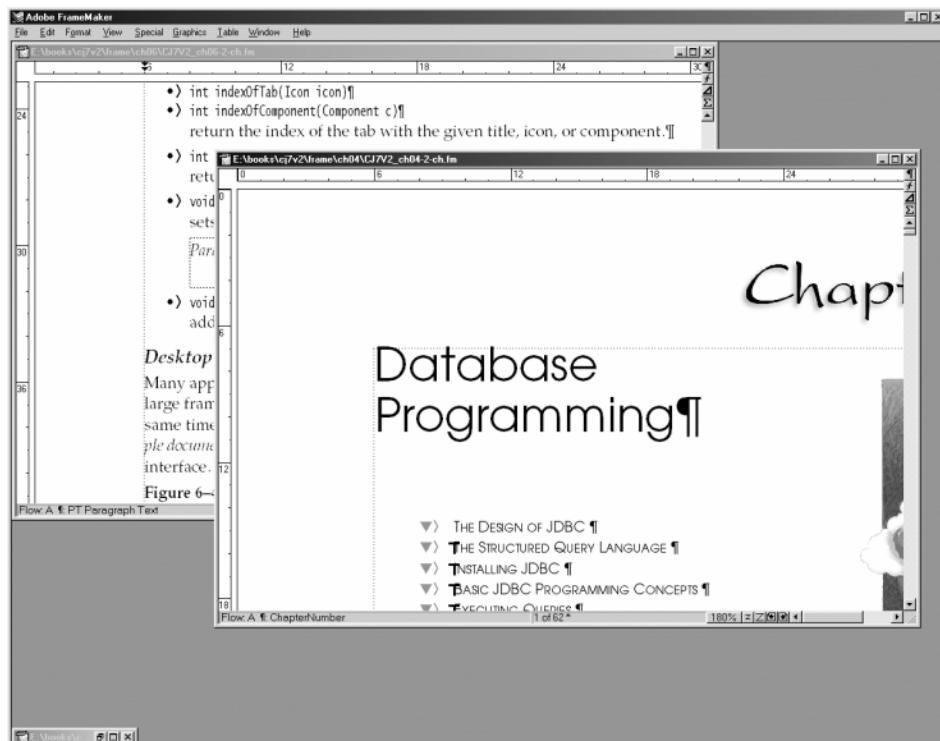


Figure 10.47 A multiple document interface application

Figure 10.48 shows a Java application with three internal frames. Two of them have decorations on the borders to maximize and iconify them. The third is in its iconified state.

In the Metal look-and-feel, the internal frames have distinctive “grabber” areas that you can use to move the frames around. You can resize the windows by dragging the resize corners.

To achieve this capability, follow these steps:

1. Use a regular `JFrame` window for the application.
2. Add the `JDesktopPane` to the `JFrame`.

```
desktop = new JDesktopPane();
add(desktop, BorderLayout.CENTER);
```

3. Construct `JInternalFrame` windows. You can specify whether you want the icons for resizing or closing the frame. Normally, you want all icons.

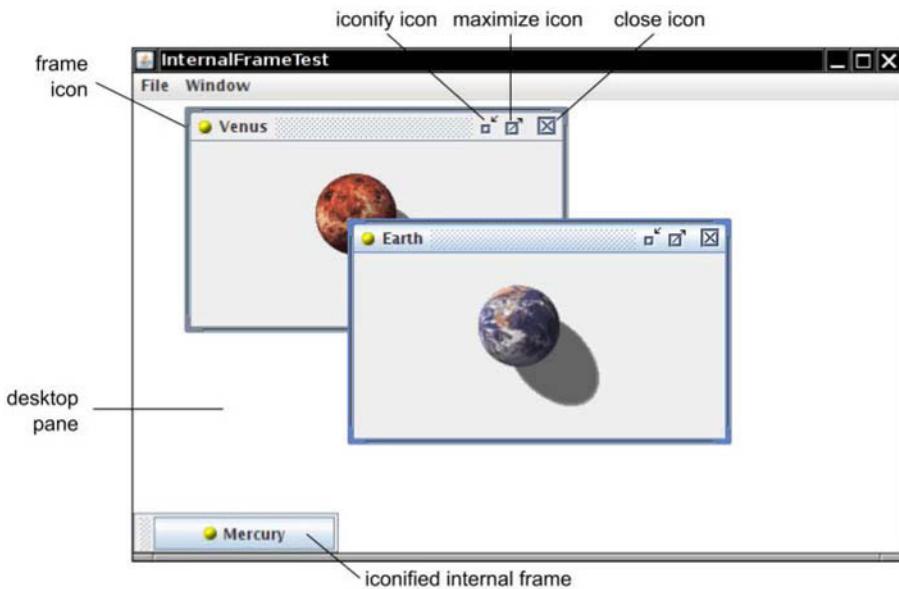


Figure 10.48 A Java application with three internal frames

```
JInternalFrame iframe = new JInternalFrame(title,
 true, // resizable
 true, // closable
 true, // maximizable
 true); // iconifiable
```

4. Add components to the frame.

```
iframe.add(c, BorderLayout.CENTER);
```

5. Set a frame icon. The icon is shown in the top left corner of the frame.

```
iframe setFrameIcon(icon);
```



**NOTE:** In the current version of the Metal look-and-feel, the frame icon is not displayed in iconified frames.

6. Set the size of the internal frame. As with regular frames, internal frames initially have a size of 0 by 0 pixels. You don't want internal frames to be displayed on top of each other, so use a variable position for the next frame. Use the reshape method to set both the position and size of the frame.

```
iframe.reshape(nextFrameX, nextFrameY, width, height);
```

7. As with `JFrame` instances, you need to make the frame visible.

```
iframe.setVisible(true);
```

---

**NOTE:** In earlier versions of Swing, internal frames were automatically visible and this call was not necessary.

---

8. Add the frame to the `JDesktopPane`.

```
desktop.add(iframe);
```

9. You will probably want to make the new frame the *selected frame*. Of the internal frames on the desktop, only the selected frame receives keyboard focus. In the Metal look-and-feel, the selected frame has a blue title bar, whereas the other frames have gray title bars. Use the `setSelected` method to select a frame. However, the “selected” property can be *vetoed*—the currently selected frame can refuse to give up focus. In that case, the `setSelected` method throws a `PropertyVetoException` that you need to handle.

```
try
{
 iframe.setSelected(true);
}
catch (PropertyVetoException ex)
{
 // attempt was vetoed
}
```

10. You will probably want to move the position of the next internal frame down so that it won’t overlay the existing frame. A good distance between frames is the height of the title bar, which you can obtain as

```
int frameDistance = iframe.getHeight() - iframe.getContentPane().getHeight();
```

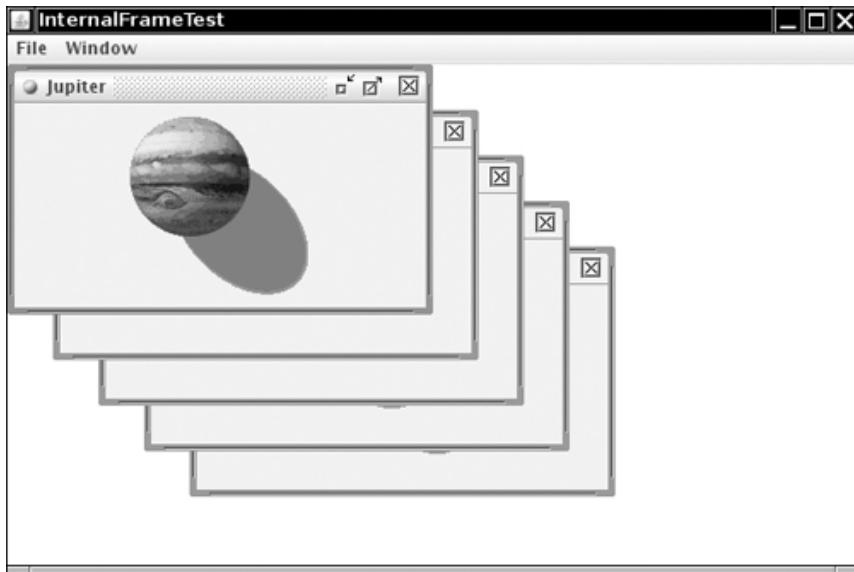
11. Use that distance to determine the next internal frame’s position.

```
nextFrameX += frameDistance;
nextFrameY += frameDistance;
if (nextFrameX + width > desktop.getWidth())
 nextFrameX = 0;
if (nextFrameY + height > desktop.getHeight())
 nextFrameY = 0;
```

### 10.6.3.2 Cascading and Tiling

In Windows, there are standard commands for *cascading* and *tiling* windows (see Figures 10.49 and 10.50). The Java `JDesktopPane` and `JInternalFrame` classes have no

built-in support for these operations. In Listing 10.31, we show how you can implement these operations yourself.



**Figure 10.49** Cascaded internal frames

To cascade all windows, reshape windows to the same size and stagger their positions. The `getAllFrames` method of the `JDesktopPane` class returns an array of all internal frames.

```
JInternalFrame[] frames = desktop.getAllFrames();
```

However, you need to pay attention to the frame state. An internal frame can be in one of three states:

- Icon
- Resizable
- Maximum

Use the `isIcon` method to find out which internal frames are currently icons and should be skipped. However, if a frame is in the maximum state, you first need to set it to be resizable by calling `setMaximum(false)`. This is another property that can be vetoed, so you must catch the `PropertyVetoException`.

The following loop cascades all internal frames on the desktop:

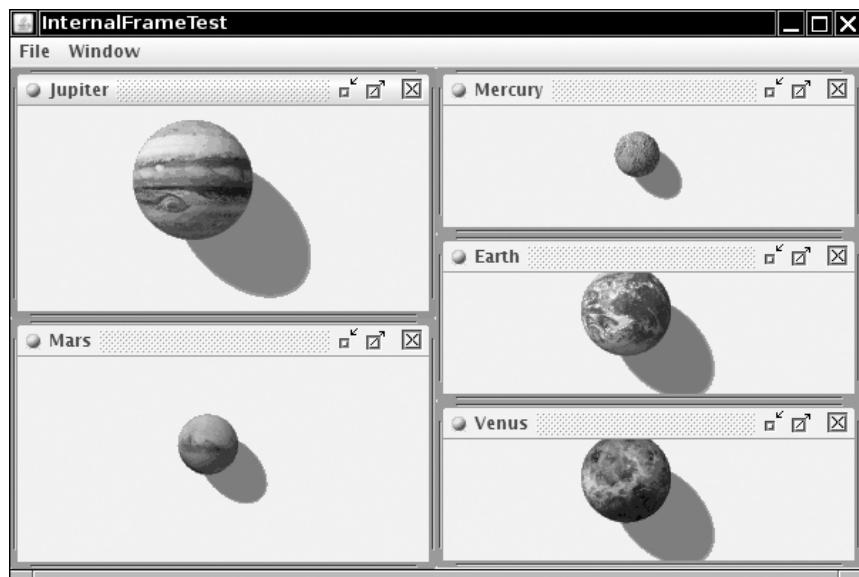


Figure 10.50 Tiled internal frames

```
for (JInternalFrame frame : desktop.getAllFrames())
{
 if (!frame.isIcon())
 {
 try
 {
 // try to make maximized frames resizable; this might be vetoed
 frame.setMaximum(false);
 frame.reshape(x, y, width, height);
 x += frameDistance;
 y += frameDistance;
 // wrap around at the desktop edge

 if (x + width > desktop.getWidth()) x = 0;
 if (y + height > desktop.getHeight()) y = 0;
 }
 catch (PropertyVetoException ex)
 {}
 }
}
```

Tiling frames is trickier, particularly if the number of frames is not a perfect square. First, count the number of frames that are not icons. Compute the number of rows in the first column as

```
int rows = (int) Math.sqrt(frameCount);
```

Then the number of columns is

```
int cols = frameCount / rows;
```

The last

```
int extra = frameCount % rows;
```

columns have `rows + 1` rows.

Here is the loop for tiling all frames on the desktop:

```
int width = desktop.getWidth() / cols;
int height = desktop.getHeight() / rows;
int r = 0;
int c = 0;
for (JInternalFrame frame : desktop.getAllFrames())
{
 if (!frame.isIcon())
 {
 try
 {
 frame.setMaximum(false);
 frame.reshape(c * width, r * height, width, height);
 r++;
 if (r == rows)
 {
 r = 0;
 c++;
 if (c == cols - extra)
 {
 // start adding an extra row
 rows++;
 height = desktop.getHeight() / rows;
 }
 }
 }
 catch (PropertyVetoException ex)
 {}
 }
}
```

The example program shows another common frame operation: moving the selection from the current frame to the next frame that isn't an icon. Traverse all frames and call `isSelected` until you find the currently selected frame. Then, look for the next frame in the sequence that isn't an icon, and try to select it by calling

```
frames[next].setSelected(true);
```

As before, that method can throw a `PropertyVetoException`, in which case you have to keep looking. If you come back to the original frame, then no other frame was selectable, and you give up. Here is the complete loop:

```
JInternalFrame[] frames = desktop.getAllFrames();
for (int i = 0; i < frames.length; i++)
{
 if (frames[i].isSelected())
 {
 // find next frame that isn't an icon and can be selected
 int next = (i + 1) % frames.length;

 while (next != i)
 {
 if (!frames[next].isIcon())
 {
 try
 {
 // all other frames are icons or veto selection
 frames[next].setSelected(true);
 frames[next].toFront();
 frames[i].toBack();
 return;
 }
 catch (PropertyVetoException ex)
 {}
 }
 next = (next + 1) % frames.length;
 }
 }
}
```

### 10.6.3.3 Vetoing Property Settings

Now that you have seen all these veto exceptions, you might wonder how your frames can issue a veto. The `JInternalFrame` class uses a rarely used mechanism from the JavaBeans specification for monitoring the setting of properties. We won't discuss this mechanism in full detail, but we will show you how your frames can veto requests for property changes.

Frames don't usually want to use a veto to protest iconization or loss of focus, but it is very common for frames to check whether it is OK to *close* them. You can close a frame with the `setClosed` method of the `JInternalFrame` class. Since the method is *vetoable*, it calls all registered *vetoable change listeners* before proceeding to make the change. That gives each of the listeners the opportunity to throw a `PropertyVetoException` and thereby terminate the call to `setClosed` before it changed any settings.

In our example program, we put up a dialog box to ask the user whether it is OK to close the window (see Figure 10.51). If the user doesn't agree, the window stays open.

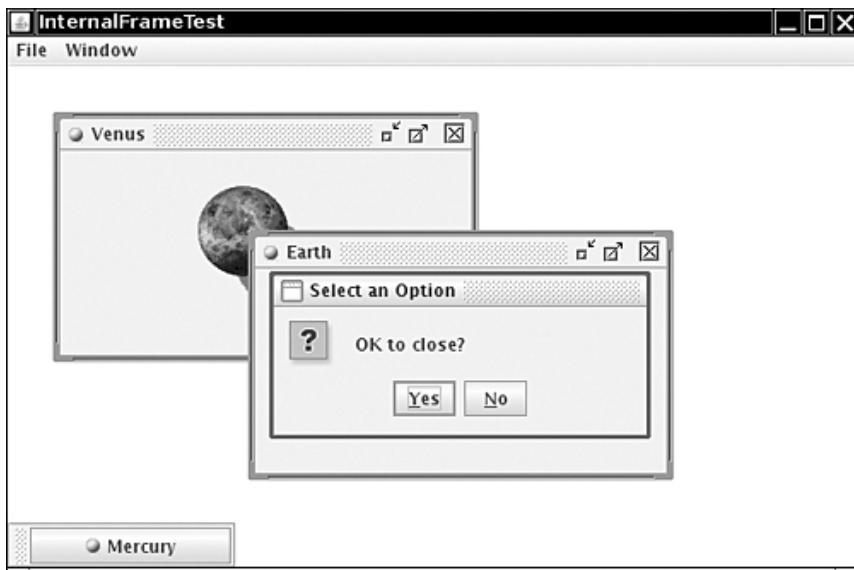


Figure 10.51 The user can veto the close property.

Here is how you achieve such a notification.

1. Add a listener object to each frame. The object must belong to some class that implements the `VetoableChangeListener` interface. It is best to add the listener right after constructing the frame. In our example, we use the frame class that constructs the internal frames. Another option would be to use an anonymous inner class.

```
iframe.addVetoableChangeListener(listener);
```

2. Implement the `vetoableChange` method, the only method required by the `VetoableChangeListener` interface. The method receives a `PropertyChangeEvent` object. Use the `getName` method to find the name of the property that is about to be changed (such as "closed" if the method call to `veto` is `setClosed(true)`). You obtain the property name by removing the "set" prefix from the method name and changing the next letter to lower case.

3. Use the `getNewValue` method to get the proposed new value.

```
String name = event.getPropertyName();
Object value = event.getNewValue();
if (name.equals("closed") && value.equals(true))
{
 ask user for confirmation
}
```

4. Simply throw a `PropertyVetoException` to block the property change. Return normally if you don't want to veto the change.

```
class DesktopFrame extends JFrame
 implements VetoableChangeListener
{
 .
 .
 public void vetoableChange(PropertyChangeEvent event)
 throws PropertyVetoException
 {
 .
 .
 if (not ok)
 throw new PropertyVetoException(reason, event);
 // return normally if ok
 }
}
```

#### 10.6.3.4 Dialogs in Internal Frames

If you use internal frames, you should not use the `JDialog` class for dialog boxes. Those dialog boxes have two disadvantages:

- They are heavyweight because they create a new frame in the windowing system.
- The windowing system does not know how to position them relative to the internal frame that spawned them.

Instead, for simple dialog boxes, use the `showInternalXxxDialog` methods of the `JOptionPane` class. They work exactly like the `showXxxDialog` methods, except they position a lightweight window over an internal frame.

As for more complex dialog boxes, construct them with a `JInternalFrame`. Unfortunately, you then have no built-in support for modal dialog boxes.

In our sample program, we use an internal dialog box to ask the user whether it is OK to close a frame.

```
int result = JOptionPane.showInternalConfirmDialog(
 iframe, "OK to close?", "Select an Option", JOptionPane.YES_NO_OPTION);
```

---

**NOTE:** If you simply want to be *notified* when a frame is closed, you should not use the veto mechanism. Instead, install an `InternalFrameListener`. An internal frame listener works just like a `WindowListener`. When the internal frame is closing, the `internalFrameClosing` method is called instead of the familiar `windowClosing` method. The other six internal frame notifications (opened/closed, iconified/deiconified, activated/deactivated) also correspond to the window listener methods.

---

#### 10.6.3.5 Outline Dragging

One criticism that developers have leveled against internal frames is that performance has not been great. By far the slowest operation is dragging a frame with complex content across the desktop. The desktop manager keeps asking the frame to repaint itself as it is being dragged, which is quite slow.

Actually, if you use Windows or X Windows with a poorly written video driver, you'll experience the same problem. Window dragging appears to be fast on most systems because the video hardware supports the dragging operation by mapping the image inside the frame to a different screen location during the dragging process.

To improve performance without greatly degrading the user experience, you can turn "outline dragging" on. When the user drags the frame, only the outline of the frame is continuously updated. The inside is repainted only when the frame is dropped to its final resting place.

To turn on outline dragging, call

```
desktop.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```

This setting is the equivalent of "continuous layout" in the `JSplitPane` class.

---

**NOTE:** In early versions of Swing, you had to use the magic incantation

```
desktop.putClientProperty("JDesktopPane.dragMode", "outline");
```

to turn on outline dragging.

---

In the sample program, you can use the Window → Drag Outline checkbox menu selection to toggle outline dragging on or off.

---

**NOTE:** The internal frames on the desktop are managed by a `DesktopManager` class. You don't need to know about this class for normal programming. It is possible to implement a different desktop behavior by installing a new desktop manager, but we don't cover that.

---

Listing 10.31 populates a desktop with internal frames that show HTML pages. The File → Open menu option pops up a file dialog box for reading a local HTML file into a new internal frame. If you click on any link, the linked document is displayed in another internal frame. Try out the Window → Cascade and Window → Tile commands.

---

**Listing 10.31** internalFrame/DesktopFrame.java

---

```
1 package internalFrame;
2
3 import java.awt.*;
4 import java.beans.*;
5
6 import javax.swing.*;
7
8 /**
9 * This desktop frame contains editor panes that show HTML documents.
10 */
11 public class DesktopFrame extends JFrame
12 {
13 private static final int DEFAULT_WIDTH = 600;
14 private static final int DEFAULT_HEIGHT = 400;
15 private static final String[] planets = { "Mercury", "Venus", "Earth", "Mars", "Jupiter",
16 "Saturn", "Uranus", "Neptune", "Pluto", };
17
18 private JDesktopPane desktop;
19 private int nextFrameX;
20 private int nextFrameY;
21 private int frameDistance;
22 private int counter;
23
24 public DesktopFrame()
25 {
26 setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
27
28 desktop = new JDesktopPane();
29 add(desktop, BorderLayout.CENTER);
30
31 // set up menus
32
33 JMenuBar menuBar = new JMenuBar();
34 setJMenuBar(menuBar);
35 JMenu fileMenu = new JMenu("File");
36 menuBar.add(fileMenu);
37 JMenuItem openItem = new JMenuItem("New");
```

```
38 openItem.addActionListener(event ->
39 {
40 createInternalFrame(new JLabel(
41 new ImageIcon(getClass().getResource(planets[counter] + ".gif"))),
42 planets[counter]);
43 counter = (counter + 1) % planets.length;
44 });
45 fileMenu.add(openItem);
46 JMenuItem exitItem = new JMenuItem("Exit");
47 exitItem.addActionListener(event -> System.exit(0));
48 fileMenu.add(exitItem);
49 JMenu windowMenu = new JMenu("Window");
50 menuBar.add(windowMenu);
51 JMenuItem nextItem = new JMenuItem("Next");
52 nextItem.addActionListener(event -> selectNextWindow());
53 windowMenu.add(nextItem);
54 JMenuItem cascadeItem = new JMenuItem("Cascade");
55 cascadeItem.addActionListener(event -> cascadeWindows());
56 windowMenu.add(cascadeItem);
57 JMenuItem tileItem = new JMenuItem("Tile");
58 tileItem.addActionListener(event -> tileWindows());
59 windowMenu.add(tileItem);
60 final JCheckBoxMenuItem dragOutlineItem = new JCheckBoxMenuItem("Drag Outline");
61 dragOutlineItem.addActionListener(event ->
62 desktop.setDragMode(dragOutlineItem.isSelected() ? JDesktopPane.OUTLINE_DRAG_MODE
63 : JDesktopPane.LIVE_DRAG_MODE));
64 windowMenu.add(dragOutlineItem);
65 }
66 /**
67 * Creates an internal frame on the desktop.
68 * @param c the component to display in the internal frame
69 * @param t the title of the internal frame
70 */
71 public void createInternalFrame(Component c, String t)
72 {
73 final JInternalFrame iframe = new JInternalFrame(t, true, // resizable
74 true, // closable
75 true, // maximizable
76 true); // iconifiable
77
78 iframe.add(c, BorderLayout.CENTER);
79 desktop.add(iframe);
80
81 iframe setFrameIcon(new ImageIcon(getClass().getResource("document.gif")));
82
83 // add listener to confirm frame closing
```

(Continues)

**Listing 10.31 (Continued)**

```
85 iframe.addVetoableChangeListener(event ->
86 {
87 String name = event.getPropertyName();
88 Object value = event.getNewValue();
89
90 // we only want to check attempts to close a frame
91 if (name.equals("closed") && value.equals(true))
92 {
93 // ask user if it is ok to close
94 int result = JOptionPane.showInternalConfirmDialog(iframe, "OK to close?", "Select an Option", JOptionPane.YES_NO_OPTION);
95
96 // if the user doesn't agree, veto the close
97 if (result != JOptionPane.YES_OPTION)
98 throw new PropertyVetoException("User canceled close", event);
99 }
100 });
101
102
103 // position frame
104 int width = desktop.getWidth() / 2;
105 int height = desktop.getHeight() / 2;
106 iframe.reshape(nextFrameX, nextFrameY, width, height);
107
108 iframe.show();
109
110 // select the frame--might be vetoed
111 try
112 {
113 iframe.setSelected(true);
114 }
115 catch (PropertyVetoException ex)
116 {
117 }
118
119 frameDistance = iframe.getHeight() - iframe.getContentPane().getHeight();
120
121 // compute placement for next frame
122
123 nextFrameX += frameDistance;
124 nextFrameY += frameDistance;
125 if (nextFrameX + width > desktop.getWidth()) nextFrameX = 0;
126 if (nextFrameY + height > desktop.getHeight()) nextFrameY = 0;
127 }
128
129 /**
130 * Cascades the noniconified internal frames of the desktop.
131 */
```

```
132 public void cascadeWindows()
133 {
134 int x = 0;
135 int y = 0;
136 int width = desktop.getWidth() / 2;
137 int height = desktop.getHeight() / 2;
138
139 for (JInternalFrame frame : desktop.getAllFrames())
140 {
141 if (!frame.isIcon())
142 {
143 try
144 {
145 // try to make maximized frames resizable; this might be vetoed
146 frame.setMaximum(false);
147 frame.reshape(x, y, width, height);
148
149 x += frameDistance;
150 y += frameDistance;
151 // wrap around at the desktop edge
152 if (x + width > desktop.getWidth()) x = 0;
153 if (y + height > desktop.getHeight()) y = 0;
154 }
155 catch (PropertyVetoException ex)
156 {
157 }
158 }
159 }
160 }
161
162 /**
163 * Tiles the noniconified internal frames of the desktop.
164 */
165 public void tileWindows()
166 {
167 // count frames that aren't iconized
168 int frameCount = 0;
169 for (JInternalFrame frame : desktop.getAllFrames())
170 if (!frame.isIcon()) frameCount++;
171 if (frameCount == 0) return;
172
173 int rows = (int) Math.sqrt(frameCount);
174 int cols = frameCount / rows;
175 int extra = frameCount % rows;
176 // number of columns with an extra row
177
178 int width = desktop.getWidth() / cols;
179 int height = desktop.getHeight() / rows;
180 int r = 0;
```

(Continues)

**Listing 10.31 (Continued)**

```
181 int c = 0;
182 for (JInternalFrame frame : desktop.getAllFrames())
183 {
184 if (!frame.isIcon())
185 {
186 try
187 {
188 frame.setMaximum(false);
189 frame.reshape(c * width, r * height, width, height);
190 r++;
191 if (r == rows)
192 {
193 r = 0;
194 C++;
195 if (c == cols - extra)
196 {
197 // start adding an extra row
198 rows++;
199 height = desktop.getHeight() / rows;
200 }
201 }
202 }
203 catch (PropertyVetoException ex)
204 {
205 }
206 }
207 }
208 }
209 /**
210 * Brings the next noniconified internal frame to the front.
211 */
212 public void selectNextWindow()
213 {
214 JInternalFrame[] frames = desktop.getAllFrames();
215 for (int i = 0; i < frames.length; i++)
216 {
217 if (frames[i].isSelected())
218 {
219 // find next frame that isn't an icon and can be selected
220 int next = (i + 1) % frames.length;
221 while (next != i)
222 {
223 if (!frames[next].isIcon())
224 {
225 try
226 {
227 // all other frames are icons or veto selection
228 }
229 }
230 }
231 }
232 }
233 }
```

```
229 frames[next].setSelected(true);
230 frames[next].toFront();
231 frames[i].toBack();
232 return;
233 }
234 catch (PropertyVetoException ex)
235 {
236 }
237 }
238 next = (next + 1) % frames.length;
239 }
240 }
241 }
242 }
243 }
```

**javax.swing.JDesktopPane 1.2**

- `JInternalFrame[] getAllFrames()`  
gets all internal frames in this desktop pane.
- `void setDragMode(int mode)`  
sets the drag mode to live or outline drag mode.

*Parameters:* mode One of `JDesktopPane.LIVE_DRAG_MODE` or  
`JDesktopPane.OUTLINE_DRAG_MODE`

**javax.swing.JInternalFrame 1.2**

- `JInternalFrame()`
- `JInternalFrame(String title)`
- `JInternalFrame(String title, boolean resizable)`
- `JInternalFrame(String title, boolean resizable, boolean closable)`
- `JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)`
- `JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)`

constructs a new internal frame.

*Parameters:* title The string to display in the title bar  
resizable true if the frame can be resized  
closable true if the frame can be closed  
maximizable true if the frame can be maximized  
iconifiable true if the frame can be iconified

(Continues)

**javax.swing.JInternalFrame 1.2 (Continued)**

- boolean isResizable()
- void setResizable(boolean b)
- boolean isClosable()
- void setClosable(boolean b)
- boolean isMaximizable()
- void setMaximizable(boolean b)
- boolean isIconifiable()
- void setIconifiable(boolean b)

gets or sets the `resizable`, `closable`, `maximizable`, and `iconifiable` properties. When the property is `true`, an icon appears in the frame title to resize, close, maximize, or iconify the internal frame.

- boolean isIcon()
- void setIcon(boolean b)
- boolean isMaximum()
- void setMaximum(boolean b)
- boolean isClosed()
- void setClosed(boolean b)

gets or sets the `icon`, `maximum`, or `closed` property. When this property is `true`, the internal frame is iconified, maximized, or closed.

- boolean isSelected()
- void setSelected(boolean b)

gets or sets the `selected` property. When this property is `true`, the current internal frame becomes the selected frame on the desktop.

- void moveToFront()
- void moveToBack()

moves this internal frame to the front or the back of the desktop.

- void reshape(int x, int y, int width, int height)

moves and resizes this internal frame.

*Parameters:*      x, y                      The top left corner of the frame  
                              width, height      The width and height of the frame

- Container getContentPane()
- void setContentPane(Container c)

gets or sets the content pane of this internal frame.

(Continues)

***javax.swing.JInternalFrame 1.2 (Continued)***

- `JDesktopPane getDesktopPane()`  
gets the desktop pane of this internal frame.
- `Icon getFrameIcon()`
- `void setFrameIcon(Icon anIcon)`  
gets or sets the frame icon that is displayed in the title bar.
- `boolean isVisible()`
- `void setVisible(boolean b)`  
gets or sets the “visible” property.
- `void show()`  
makes this internal frame visible and brings it to the front.

***javax.swing.JComponent 1.2***

- `void addVetoableChangeListener(VetoableChangeListener listener)`  
adds a vetoable change listener that is notified when an attempt is made to change a constrained property.

***java.beans.VetoableChangeListener 1.1***

- `void vetoableChange(PropertyChangeEvent event)`  
is called when the set method of a constrained property notifies the vetoable change listeners.

***java.beans.PropertyChangeEvent 1.1***

- `String getPropertyName()`  
returns the name of the property that is about to be changed.
- `Object getNewValue()`  
returns the proposed new value for the property.

**java.beans.PropertyVetoException 1.1**

- `PropertyVetoException(String reason, PropertyChangeEvent event)`  
constructs a property veto exception.

*Parameters:*      `reason`                  The reason for the veto  
                      `event`                      The vetoed event

## 10.6.4 Layers

Java SE 1.7 introduces a feature that lets you place a layer over another component. You can paint on the layer and listen to events of the underlying component. You can use layers to add visual clues to your user interface. For example, you can decorate the current input, invalid inputs, or disabled components.

The `JLayer` class associates a component with a `LayerUI` object that is in charge of painting and event handling. The `LayerUI` class has a type parameter that must match the associated component. For example, here we add a layer to a `JPanel`:

```
JPanel panel = new JPanel();
LayerUI<JPanel> layerUI = new PanelLayer();
JLayer layer = new JLayer(panel, layerUI);
frame.add(layer);
```

Note that you add the layer, not the panel, to the parent. Here, `PanelLayer` is a subclass:

```
class PanelLayer extends LayerUI<Panel>
{
 public void paint(Graphics g, JComponent c)
 {
 ...
 }
 ...
}
```

In the `paint` method, you can paint anything you like. Be sure to call `super.paint` to have the component painted. Here, we draw a transparent color over the entire component:

```
public void paint(Graphics g, JComponent c)
{
 super.paint(g, c);

 Graphics2D g2 = (Graphics2D) g.create();
 g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, .3f));
 g2.setPaint(color);
 g2.fillRect(0, 0, c.getWidth(), c.getHeight());
```

```
 g2.dispose();
}
```

In order to listen to events from the associated component or any of its children, the `LayerUI` class must set the layer's event mask. This should be done in the `installUI` method, like this:

```
class PanelLayer extends LayerUI< JPanel >
{
 public void installUI(JComponent c)
 {
 super.installUI(c);
 ((JLayer<?>) c).setLayerEventMask(AWTEvent.KEY_EVENT_MASK | AWTEvent.FOCUS_EVENT_MASK);
 }

 public void uninstallUI(JComponent c)
 {
 ((JLayer<?>) c).setLayerEventMask(0);
 super.uninstallUI(c);
 }
 ...
}
```

Now you will receive events in the methods named `processXxxEvent`. For example, in our sample application, we repaint the layer after every keystroke:

```
public class PanelLayer extends LayerUI< JPanel >
{
 protected void processKeyEvent(KeyEvent e, JLayer<? extends JPanel> l)
 {
 l.repaint();
 }
}
```

Our sample program in Listing 10.32 has three input fields for the RGB values of a color. Whenever the user changes the values, the color is shown transparently over the panel. We also trap focus events and show the text of the focused component in a bold font.

#### **Listing 10.32** `layer/ColorFrame.java`

```
1 package layer;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import javax.swing.plaf.*;
```

(Continues)

**Listing 10.32 (Continued)**

```
8 /**
9 * A frame with three text fields to set the background color.
10 */
11 public class ColorFrame extends JFrame
12 {
13 private JPanel panel;
14 private JTextField redField;
15 private JTextField greenField;
16 private JTextField blueField;
17
18 public ColorFrame()
19 {
20 panel = new JPanel();
21
22 panel.add(new JLabel("Red:"));
23 redField = new JTextField("255", 3);
24 panel.add(redField);
25
26 panel.add(new JLabel("Green:"));
27 greenField = new JTextField("255", 3);
28 panel.add(greenField);
29
30 panel.add(new JLabel("Blue:"));
31 blueField = new JTextField("255", 3);
32 panel.add(blueField);
33
34 LayerUI<JPanel> layerUI = new PanelLayer();
35 JLayer<JPanel> layer = new JLayer<JPanel>(panel, layerUI);
36
37 add(layer);
38 pack();
39 }
40
41 class PanelLayer extends LayerUI<JPanel>
42 {
43 public void installUI(JComponent c)
44 {
45 super.installUI(c);
46 ((JLayer<?>) c).setLayerEventMask(AWTEvent.KEY_EVENT_MASK | AWTEvent.FOCUS_EVENT_MASK);
47 }
48
49 public void uninstallUI(JComponent c)
50 {
51 ((JLayer<?>) c).setLayerEventMask(0);
52 super.uninstallUI(c);
53 }
54 }
55 }
```

```
54
55 protected void processKeyEvent(KeyEvent e, JLayer<? extends JPanel> l)
56 {
57 l.repaint();
58 }
59
60 protected void processFocusEvent(FocusEvent e, JLayer<? extends JPanel> l)
61 {
62 if (e.getID() == FocusEvent.FOCUS_GAINED)
63 {
64 Component c = e.getComponent();
65 c.setFont(getFont().deriveFont(Font.BOLD));
66 }
67 if (e.getID() == FocusEvent.FOCUS_LOST)
68 {
69 Component c = e.getComponent();
70 c.setFont(getFont().deriveFont(Font.PLAIN));
71 }
72 }
73
74 public void paint(Graphics g, JComponent c)
75 {
76 super.paint(g, c);
77
78 Graphics2D g2 = (Graphics2D) g.create();
79 g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, .3f));
80 int red = Integer.parseInt(redField.getText().trim());
81 int green = Integer.parseInt(greenField.getText().trim());
82 int blue = Integer.parseInt(blueField.getText().trim());
83 g2.setPaint(new Color(red, green, blue));
84 g2.fillRect(0, 0, c.getWidth(), c.getHeight());
85 g2.dispose();
86 }
87 }
88 }
```

**javax.swing.JLayer<V extends Component> 7**

- `JLayer(V view, LayerUI<V> ui)`

constructs a layer over the given view, delegating painting and event handling to the `ui` object.

(Continues)

**javax.swing.JLayer<V extends Component> 7 (Continued)**

- void setLayerEventMask(long layerEventMask)

enables sending of all matching events, sent to the associated component or any of its descendants, to the associated LayerUI. For the event mask, combine any of the constants

|                             |                         |
|-----------------------------|-------------------------|
| COMPONENT_EVENT_MASK        | KEY_EVENT_MASK          |
| FOCUS_EVENT_MASK            | MOUSE_EVENT_MASK        |
| HIERARCHY_BOUNDS_EVENT_MASK | MOUSE_MOTION_EVENT_MASK |
| HIERARCHY_EVENT_MASK        | MOUSE_WHEEL_EVENT_MASK  |
| INPUT_METHOD_EVENT_MASK     |                         |

from the AWTEvent class.

**javax.swing.plaf.LayerUI<V extends Component> 7**

- void installUI(Component c)
- void uninstallUI(Component c)

Called when the LayerUI for the component c is installed or uninstalled. Override to set or clear the layer event mask.

- void paint(Graphics g, Component c)

Called when the decorated component is painted. Override to call super.paint and paint decorations.

- void processComponentEvent(ComponentEvent e, JLayer<? extends V> l)
- void processFocusEvent(FocusEvent e, JLayer<? extends V> l)
- void processHierarchyBoundsEvent(HierarchyEvent e, JLayer<? extends V> l)
- void processHierarchyEvent(HierarchyEvent e, JLayer<? extends V> l)
- void processInputMethodEvent(InputMethodEvent e, JLayer<? extends V> l)
- void processKeyEvent(KeyEvent e, JLayer<? extends V> l)
- void processMouseEvent(MouseEvent e, JLayer<? extends V> l)
- void processMouseMotionEvent(MouseEvent e, JLayer<? extends V> l)
- void processMouseWheelEvent(MouseWheelEvent e, JLayer<? extends V> l)

Called when the specified event is sent to this LayerUI.

You have now seen how to use the complex components that the Swing framework offers. In the next chapter, we will turn to advanced AWT issues: complex drawing operations, image manipulation, printing, and interfacing with the native windowing system.

# Advanced AWT

## In this chapter

- 11.1 The Rendering Pipeline, page 766
- 11.2 Shapes, page 769
- 11.3 Areas, page 786
- 11.4 Strokes, page 788
- 11.5 Paint, page 797
- 11.6 Coordinate Transformations, page 799
- 11.7 Clipping, page 805
- 11.8 Transparency and Composition, page 807
- 11.9 Rendering Hints, page 817
- 11.10 Readers and Writers for Images, page 823
- 11.11 Image Manipulation, page 834
- 11.12 Printing, page 851
- 11.13 The Clipboard, page 887
- 11.14 Drag and Drop, page 903
- 11.15 Platform Integration, page 921

You can use the methods of the `Graphics` class to create simple drawings. Those methods are sufficient for simple applets and applications, but they fall short when you need to create complex shapes or require complete control over the appearance of the graphics. The Java 2D API is a more sophisticated class library

that you can use to produce high-quality drawings. In this chapter, we will give you an overview of that API.

We'll then turn to the topic of printing and show how you can implement printing capabilities in your programs.

We will cover two techniques for transferring data between programs: the system clipboard and the drag-and-drop mechanism. You can use these techniques to transfer data between two Java applications or between a Java application and a native program. Finally, we cover techniques for making Java applications feel more like native applications, such as providing a splash screen and an icon in the system tray.

## 11.1 The Rendering Pipeline

The original JDK 1.0 had a very simple mechanism for drawing shapes. You selected color and paint mode, and called methods of the `Graphics` class such as `drawRect` or `filloval`. The Java 2D API supports many more options.

- You can easily produce a wide variety of *shapes*.
- You have control over the *stroke*—the pen that traces shape boundaries.
- You can *fill* shapes with solid colors, varying hues, and repeating patterns.
- You can use *transformations* to move, scale, rotate, or stretch shapes.
- You can *clip* shapes to restrict them to arbitrary areas.
- You can select *composition rules* to describe how to combine the pixels of a new shape with existing pixels.
- You can give *rendering hints* to make trade-offs between speed and drawing quality.

To draw a shape, go through the following steps:

1. Obtain an object of the `Graphics2D` class. This class is a subclass of the `Graphics` class. Ever since Java SE 1.2, methods such as `paint` and `paintComponent` automatically receive an object of the `Graphics2D` class. Simply use a cast, as follows:

```
public void paintComponent(Graphics g)
{
 Graphics2D g2 = (Graphics2D) g;
 ...
}
```

2. Use the `setRenderingHints` method to set *rendering hints*—trade-offs between speed and drawing quality.

```
RenderingHints hints = . . .;
g2.setRenderingHints(hints);
```

3. Use the `setStroke` method to set the *stroke*. The stroke draws the outline of the shape. You can select the thickness and choose among solid and dotted lines.

```
Stroke stroke = . . .;
g2.setStroke(stroke);
```

4. Use the `setPaint` method to set the *paint*. The paint fills areas such as the stroke path or the interior of a shape. You can create solid color paint, paint with changing hues, or tiled fill patterns.

```
Paint paint = . . .;
g2.setPaint(paint);
```

5. Use the `clip` method to set the *clipping region*.

```
Shape clip = . . .;
g2.clip(clip);
```

6. Use the `transform` method to set a *transformation* from user space to device space. Use transformations if it is easier for you to define your shapes in a custom coordinate system than by using pixel coordinates.

```
AffineTransform transform = . . .;
g2.transform(transform);
```

7. Use the `setComposite` method to set a *composition rule* that describes how to combine the new pixels with the existing pixels.

```
Composite composite = . . .;
g2.setComposite(composite);
```

8. Create a shape. The Java 2D API supplies many shape objects and methods to combine shapes.

```
Shape shape = . . .;
```

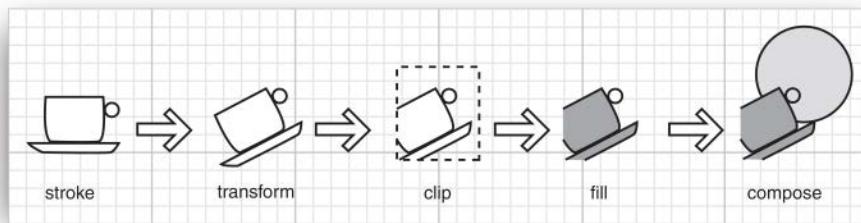
9. Draw or fill the shape. If you draw the shape, its outline is stroked. If you fill the shape, the interior is painted.

```
g2.draw(shape);
g2.fill(shape);
```

Of course, in many practical circumstances, you don't need all these steps. There are reasonable defaults for the settings of the 2D graphics context; change the settings only if you want to deviate from the defaults.

In the following sections, you will see how to describe shapes, strokes, paints, transformations, and composition rules.

The various `set` methods simply set the state of the 2D graphics context. They don't cause any drawing. Similarly, when you construct `Shape` objects, no drawing takes place. A shape is only rendered when you call `draw` or `fill`. At that time, the new shape is computed in a *rendering pipeline* (see Figure 11.1).



**Figure 11.1** The rendering pipeline

In the rendering pipeline, the following steps take place to render a shape:

1. The path of the shape is stroked.
2. The shape is transformed.
3. The shape is clipped. If there is no intersection between the shape and the clipping area, the process stops.
4. The remainder of the shape after clipping is filled.
5. The pixels of the filled shape are composed with the existing pixels. (In Figure 11.1, the circle is part of the existing pixels, and the cup shape is superimposed over it.)

In the next section, you will see how to define shapes. Then, we will turn to the 2D graphics context settings.

#### java.awt.Graphics2D 1.2

- `void draw(Shape s)`  
draws the outline of the given shape with the current paint.
- `void fill(Shape s)`  
fills the interior of the given shape with the current paint.

## 11.2 Shapes

Here are some of the methods in the `Graphics` class to draw shapes:

```
drawLine
drawRectangle
drawRoundRect
draw3DRect
drawPolygon
drawPolyline
drawOval
drawArc
```

There are also corresponding `fill` methods. These methods have been in the `Graphics` class ever since JDK 1.0. The Java 2D API uses a completely different, object-oriented approach. Instead of methods, there are classes:

```
Line2D
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
QuadCurve2D
CubicCurve2D
GeneralPath
```

These classes all implement the `Shape` interface, which we will examine in the following sections.

### 11.2.1 The Shape Class Hierarchy

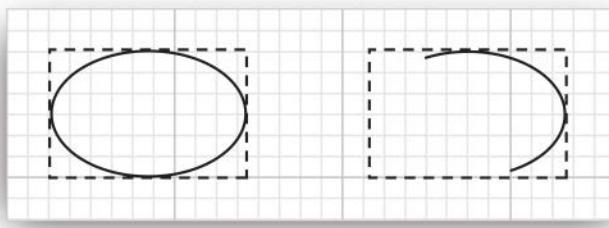
The `Line2D`, `Rectangle2D`, `RoundRectangle2D`, `Ellipse2D`, and `Arc2D` classes correspond to the `drawLine`, `drawRectangle`, `drawRoundRect`, `drawOval`, and `drawArc` methods. (The concept of a “3D rectangle” has died the death it so richly deserved—there is no analog to the `draw3DRect` method.) The Java 2D API supplies two additional classes, quadratic and cubic curves, that we will discuss in this section. There is no `Polygon2D` class; instead, the `GeneralPath` class describes paths made up from lines, quadratic and cubic curves. You can use a `GeneralPath` to describe a polygon; we’ll show you how later in this section.

To draw a shape, first create an object of a class that implements the `Shape` interface and then call the `draw` method of the `Graphics2D` class.

### The classes

```
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
```

all inherit from a common superclass `RectangularShape`. Admittedly, ellipses and arcs are not rectangular, but they have a *bounding rectangle* (see Figure 11.2).



**Figure 11.2** The bounding rectangle of an ellipse and an arc

Each of the classes with a name ending in “2D” has two subclasses for specifying coordinates as `float` or `double` quantities. In Volume I, you already encountered `Rectangle2D.Float` and `Rectangle2D.Double`.

The same scheme is used for the other classes, such as `Arc2D.Float` and `Arc2D.Double`.

Internally, all graphics classes use `float` coordinates because `float` numbers use less storage space but have sufficient precision for geometric computations. However, the Java programming language makes it a bit more tedious to manipulate `float` numbers. For that reason, most methods of the graphics classes use `double` parameters and return values. Only when constructing a 2D object you need to choose between the constructors with `float` and `double` coordinates. For example,

```
Rectangle2D floatRect = new Rectangle2D.Float(5F, 10F, 7.5F, 15F);
Rectangle2D doubleRect = new Rectangle2D.Double(5, 10, 7.5, 15);
```

The `Xxx2D.Float` and `Xxx2D.Double` classes are subclasses of the `Xxx2D` classes. After object construction, essentially no benefit accrues from remembering the subclass, and you can just store the constructed object in a superclass variable as in the code example above.

As you can see from the curious names, the `Xxx2D.Float` and `Xxx2D.Double` classes are also inner classes of the `Xxx2D` classes. That is just a minor syntactical convenience to avoid inflation of outer class names.

Finally, the `Point2D` class describes a point with an  $x$  and a  $y$  coordinate. Points are used to define shapes, but they aren't themselves shapes.

Figure 11.3 shows the relationships between the shape classes. However, the `Double` and `Float` subclasses are omitted. Legacy classes from the pre-2D library are marked with a gray fill.

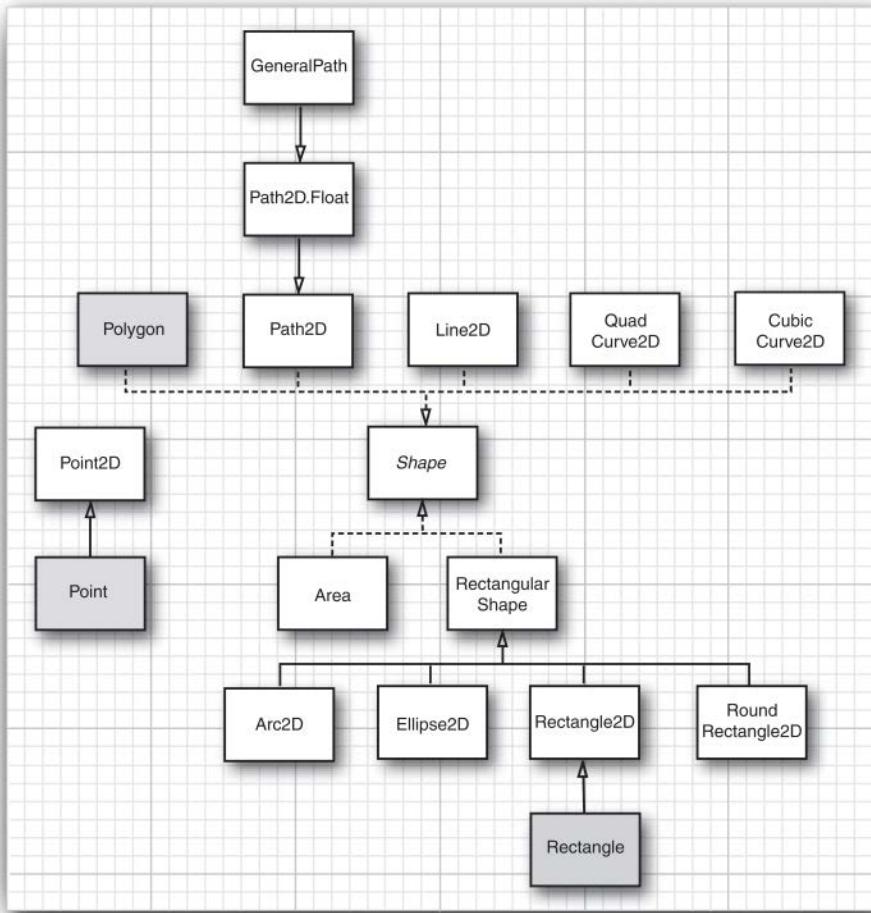


Figure 11.3 Relationships between the shape classes

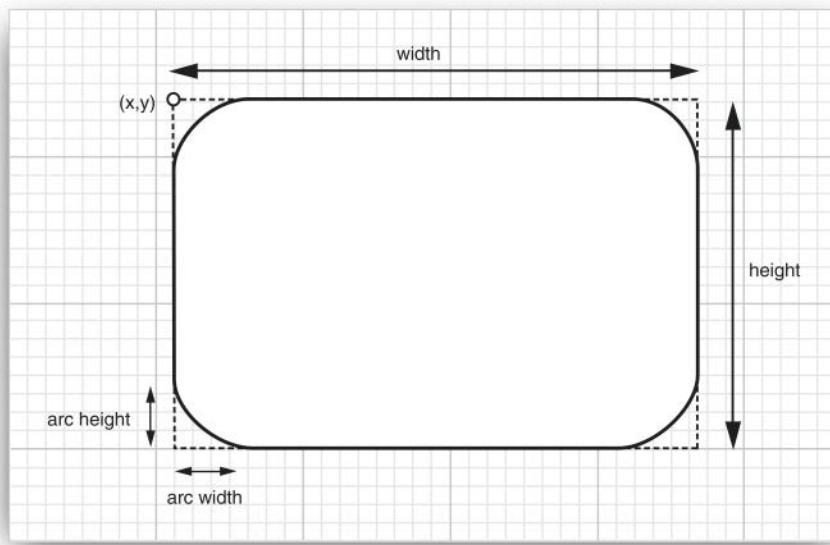
## 11.2.2 Using the Shape Classes

You already saw how to use the `Rectangle2D`, `Ellipse2D`, and `Line2D` classes in Volume I, Chapter 10. In this section, you will learn how to work with the remaining 2D shapes.

For the `RoundRectangle2D` shape, specify the top left corner, width, height, and the *x* and *y* dimensions of the corner area that should be rounded (see Figure 11.4). For example, the call

```
RoundRectangle2D r = new RoundRectangle2D.Double(150, 200, 100, 50, 20, 20);
```

produces a rounded rectangle with circles of radius 20 at each of the corners.



**Figure 11.4** Constructing a `RoundRectangle2D`

To construct an arc, specify the bounding box, the start angle, the angle swept out by the arc (see Figure 11.5), and the closure type—one of `Arc2D.OPEN`, `Arc2D.PIE`, or `Arc2D.CHORD`.

```
Arc2D a = new Arc2D(x, y, width, height, startAngle, arcAngle, closureType);
```

Figure 11.6 illustrates the arc types.

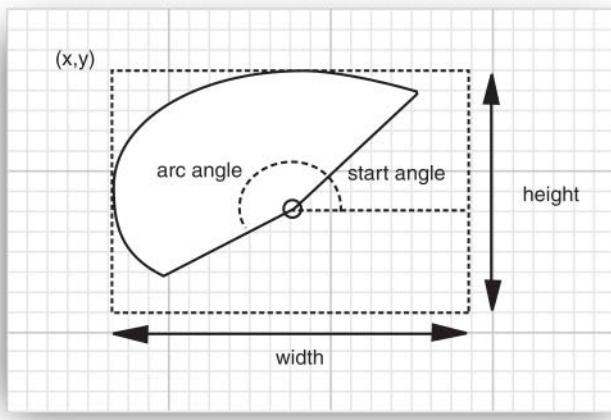


Figure 11.5 Constructing an elliptical arc



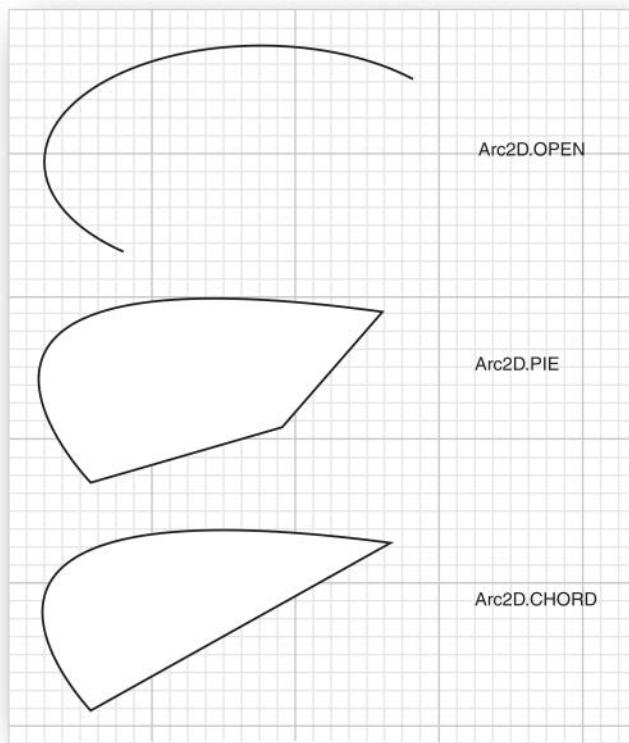
**CAUTION:** If the arc is elliptical, the computation of the arc angles is not at all straightforward. The API documentation states: “The angles are specified relative to the nonsquare framing rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the framing rectangle. As a result, if the framing rectangle is noticeably longer along one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the frame.” Unfortunately, the documentation is silent on how to compute this “skew.” Here are the details:

Suppose the center of the arc is the origin and the point  $(x, y)$  lies on the arc. You can get a skewed angle with the following formula:

```
skewedAngle = Math.toDegrees(Math.atan2(-y * height, x * width));
```

The result is a value between -180 and 180. Compute the skewed start and end angles in this way. Then, compute the difference between the two skewed angles. If the start angle or the difference is negative, add 360 to the start angle. Then, supply the start angle and the difference to the arc constructor.

If you run the example program at the end of this section, you can visually check that this calculation yields the correct values for the arc constructor (see Figure 11.9 on p. 777).



**Figure 11.6** Arc types

The Java 2D API supports *quadratic* and *cubic* curves. In this chapter, we do not get into the mathematics of these curves. We suggest you get a feel for how the curves look by running the program in Listing 11.1. As you can see in Figures 11.7 and 11.8, quadratic and cubic curves are specified by two *end points* and one or two *control points*. Moving the control points changes the shape of the curves.

To construct quadratic and cubic curves, give the coordinates of the end points and the control points. For example,

```
QuadCurve2D q = new QuadCurve2D.Double(startX, startY, controlX, controlY, endX, endY);
CubicCurve2D c = new CubicCurve2D.Double(startX, startY, control1X, control1Y,
 control2X, control2Y, endX, endY);
```

Quadratic curves are not very flexible, and they are not commonly used in practice. Cubic curves (such as the Bézier curves drawn by the `CubicCurve2D` class) are, however, very common. By combining many cubic curves so that the slopes at the connection points match, you can create complex, smooth-looking curved shapes.

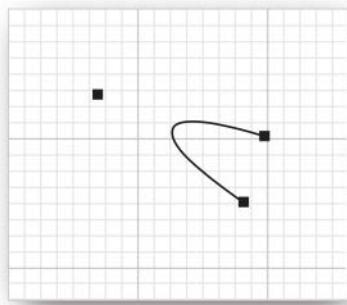


Figure 11.7 A quadratic curve

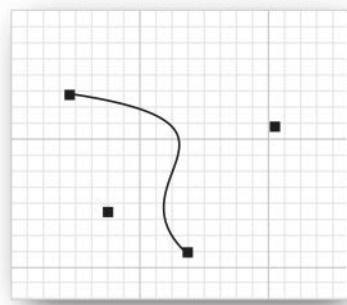


Figure 11.8 A cubic curve

For more information, we refer you to *Computer Graphics: Principles and Practice, Second Edition in C*, by James D. Foley, Andries van Dam, Steven K. Feiner, et al. (Addison-Wesley, 1995).

You can build arbitrary sequences of line segments, quadratic curves, and cubic curves, and store them in a `GeneralPath` object. Specify the first coordinate of the path with the `moveTo` method, for example:

```
GeneralPath path = new GeneralPath();
path.moveTo(10, 20);
```

You can then extend the path by calling one of the methods `lineTo`, `quadTo`, or `curveTo`. These methods extend the path by a line, a quadratic curve, or a cubic curve. To call `lineTo`, supply the end point. For the two curve methods, supply the control points, then the end point. For example,

```
path.lineTo(20, 30);
path.curveTo(control1X, control1Y, control2X, control2Y, endX, endY);
```

Close the path by calling the `closePath` method. It draws a line back to the starting point of the path.

To make a polygon, simply call `moveTo` to go to the first corner point, followed by repeated calls to `lineTo` to visit the other corner points. Finally, call `closePath` to close the polygon. The program in Listing 11.1 shows this in more detail.

A general path does not have to be connected. You can call `moveTo` at any time to start a new path segment.

Finally, you can use the `append` method to add arbitrary `Shape` objects to a general path. The outline of the shape is added to the end to the path. The second parameter of the `append` method is `true` if the new shape should be connected to the last point on the path, `false` otherwise. For example, the call

```
Rectangle2D r = . . .;
path.append(r, false);
```

appends the outline of a rectangle to the path without connecting it to the existing path. But

```
path.append(r, true);
```

adds a straight line from the end point of the path to the starting point of the rectangle, and then adds the rectangle outline to the path.

The program in Listing 11.1 lets you create sample paths. Figures 11.7 and 11.8 show sample runs of the program. You can pick a shape maker from the combo box. The program contains shape makers for

- Straight lines
- Rectangles, rounded rectangles, and ellipses
- Arcs (showing lines for the bounding rectangle and the start and end angles, in addition to the arc itself)
- Polygons (using a `GeneralPath`)
- Quadratic and cubic curves

Use the mouse to adjust the control points. As you move them, the shape continuously repaints itself.

The program is a bit complex because it handles multiple shapes and supports dragging of the control points.

An abstract superclass `ShapeMaker` encapsulates the commonality of the shape maker classes. Each shape has a fixed number of control points that the user can move around. The `getPointCount` method returns that value. The abstract method

```
Shape makeShape(Point2D[] points)
```

computes the actual shape, given the current positions of the control points. The `toString` method returns the class name so that the `ShapeMaker` objects can simply be dumped into a `JComboBox`.

To enable dragging of the control points, the `ShapePanel` class handles both mouse and mouse motion events. If the mouse is pressed on top of a rectangle, subsequent mouse drags move the rectangle.

The majority of the shape maker classes are simple—their `makeShape` methods just construct and return the requested shapes. However, the `ArcMaker` class needs to compute the distorted start and end angles. Furthermore, to demonstrate that the computation is indeed correct, the returned shape is a `GeneralPath` containing the arc itself, the bounding rectangle, and the lines from the center of the arc to the angle control points (see Figure 11.9).

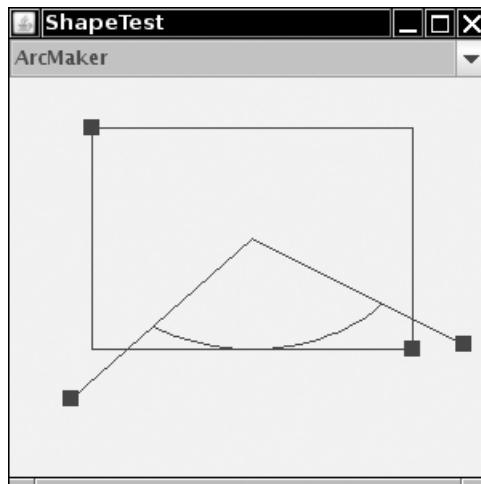


Figure 11.9 The `ShapeTest` program

#### **Listing 11.1** `shape/ShapeTest.java`

```
1 package shape;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.geom.*;
6 import java.util.*;
7 import javax.swing.*;
8
```

(Continues)

**Listing 11.1 (Continued)**

```
9 /**
10 * This program demonstrates the various 2D shapes.
11 * @version 1.03 2016-05-10
12 * @author Cay Horstmann
13 */
14 public class ShapeTest
15 {
16 public static void main(String[] args)
17 {
18 EventQueue.invokeLater(() ->
19 {
20 JFrame frame = new ShapeTestFrame();
21 frame.setTitle("ShapeTest");
22 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23 frame.setVisible(true);
24 });
25 }
26 }
27
28 /**
29 * This frame contains a combo box to select a shape and a component to draw it.
30 */
31 class ShapeTestFrame extends JFrame
32 {
33 public ShapeTestFrame()
34 {
35 final ShapeComponent comp = new ShapeComponent();
36 add(comp, BorderLayout.CENTER);
37 final JComboBox<ShapeMaker> comboBox = new JComboBox<>();
38 comboBox.addItem(new LineMaker());
39 comboBox.addItem(new RectangleMaker());
40 comboBox.addItem(new RoundRectangleMaker());
41 comboBox.addItem(new EllipseMaker());
42 comboBox.addItem(new ArcMaker());
43 comboBox.addItem(new PolygonMaker());
44 comboBox.addItem(new QuadCurveMaker());
45 comboBox.addItem(new CubicCurveMaker());
46 comboBox.addActionListener(event ->
47 {
48 ShapeMaker shapeMaker = comboBox.getItemAt(comboBox.getSelectedIndex());
49 comp.setShapeMaker(shapeMaker);
50 });
51 add(comboBox, BorderLayout.NORTH);
52 comp.setShapeMaker((ShapeMaker) comboBox.getItemAt(0));
53 pack();
54 }
55 }
```

```
57 /**
58 * This component draws a shape and allows the user to move the points that define it.
59 */
60 class ShapeComponent extends JPanel
61 {
62 private static final Dimension PREFERRED_SIZE = new Dimension(300, 200);
63 private Point2D[] points;
64 private static Random generator = new Random();
65 private static int SIZE = 10;
66 private int current;
67 private ShapeMaker shapeMaker;
68
69 public ShapeComponent()
70 {
71 addMouseListener(new MouseAdapter()
72 {
73 public void mousePressed(MouseEvent event)
74 {
75 Point p = event.getPoint();
76 for (int i = 0; i < points.length; i++)
77 {
78 double x = points[i].getX() - SIZE / 2;
79 double y = points[i].getY() - SIZE / 2;
80 Rectangle2D r = new Rectangle2D.Double(x, y, SIZE, SIZE);
81 if (r.contains(p))
82 {
83 current = i;
84 return;
85 }
86 }
87 }
88
89 public void mouseReleased(MouseEvent event)
90 {
91 current = -1;
92 }
93 });
94 addMouseMotionListener(new MouseMotionAdapter()
95 {
96 public void mouseDragged(MouseEvent event)
97 {
98 if (current == -1) return;
99 points[current] = event.getPoint();
100 repaint();
101 }
102 });
103 current = -1;
104 }
105}
```

(Continues)

**Listing 11.1 (Continued)**

```
106 /**
107 * Set a shape maker and initialize it with a random point set.
108 * @param aShapeMaker a shape maker that defines a shape from a point set
109 */
110 public void setShapeMaker(ShapeMaker aShapeMaker)
111 {
112 shapeMaker = aShapeMaker;
113 int n = shapeMaker.getPointCount();
114 points = new Point2D[n];
115 for (int i = 0; i < n; i++)
116 {
117 double x = generator.nextDouble() * getWidth();
118 double y = generator.nextDouble() * getHeight();
119 points[i] = new Point2D.Double(x, y);
120 }
121 repaint();
122 }
123
124 public void paintComponent(Graphics g)
125 {
126 if (points == null) return;
127 Graphics2D g2 = (Graphics2D) g;
128 for (int i = 0; i < points.length; i++)
129 {
130 double x = points[i].getX() - SIZE / 2;
131 double y = points[i].getY() - SIZE / 2;
132 g2.fill(new Rectangle2D.Double(x, y, SIZE, SIZE));
133 }
134
135 g2.draw(shapeMaker.makeShape(points));
136 }
137
138 public Dimension getPreferredSize() { return PREFERRED_SIZE; }
139 }
140
141 /**
142 * A shape maker can make a shape from a point set. Concrete subclasses must return a shape in the
143 * makeShape method.
144 */
145 abstract class ShapeMaker
146 {
147 private int pointCount;
148
149 /**
150 * Constructs a shape maker.
151 * @param pointCount the number of points needed to define this shape.
152 */
153 }
```

```
153 public ShapeMaker(int pointCount)
154 {
155 this.pointCount = pointCount;
156 }
157
158 /**
159 * Gets the number of points needed to define this shape.
160 * @return the point count
161 */
162 public int getPointCount()
163 {
164 return pointCount;
165 }
166
167 /**
168 * Makes a shape out of the given point set.
169 * @param p the points that define the shape
170 * @return the shape defined by the points
171 */
172 public abstract Shape makeShape(Point2D[] p);
173
174 public String toString()
175 {
176 return getClass().getName();
177 }
178 }
179
180 /**
181 * Makes a line that joins two given points.
182 */
183 class LineMaker extends ShapeMaker
184 {
185 public LineMaker()
186 {
187 super(2);
188 }
189
190 public Shape makeShape(Point2D[] p)
191 {
192 return new Line2D.Double(p[0], p[1]);
193 }
194 }
195
196 /**
197 * Makes a rectangle that joins two given corner points.
198 */
199 class RectangleMaker extends ShapeMaker
200 {
```

(Continues)

**Listing 11.1 (Continued)**

```
201 public RectangleMaker()
202 {
203 super(2);
204 }
205
206 public Shape makeShape(Point2D[] p)
207 {
208 Rectangle2D s = new Rectangle2D.Double();
209 s setFrameFromDiagonal(p[0], p[1]);
210 return s;
211 }
212 }
213
214 /**
215 * Makes a round rectangle that joins two given corner points.
216 */
217 class RoundRectangleMaker extends ShapeMaker
218 {
219 public RoundRectangleMaker()
220 {
221 super(2);
222 }
223
224 public Shape makeShape(Point2D[] p)
225 {
226 RoundRectangle2D s = new RoundRectangle2D.Double(0, 0, 0, 0, 20, 20);
227 s setFrameFromDiagonal(p[0], p[1]);
228 return s;
229 }
230 }
231
232 /**
233 * Makes an ellipse contained in a bounding box with two given corner points.
234 */
235 class EllipseMaker extends ShapeMaker
236 {
237 public EllipseMaker()
238 {
239 super(2);
240 }
241
242 public Shape makeShape(Point2D[] p)
243 {
244 Ellipse2D s = new Ellipse2D.Double();
245 s setFrameFromDiagonal(p[0], p[1]);
246 return s;
247 }
248 }
```

```
249
250 /**
251 * Makes an arc contained in a bounding box with two given corner points, and with starting and
252 * ending angles given by lines emanating from the center of the bounding box and ending in two
253 * given points. To show the correctness of the angle computation, the returned shape contains the
254 * arc, the bounding box, and the lines.
255 */
256 class ArcMaker extends ShapeMaker
257 {
258 public ArcMaker()
259 {
260 super(4);
261 }
262
263 public Shape makeShape(Point2D[] p)
264 {
265 double centerX = (p[0].getX() + p[1].getX()) / 2;
266 double centerY = (p[0].getY() + p[1].getY()) / 2;
267 double width = Math.abs(p[1].getX() - p[0].getX());
268 double height = Math.abs(p[1].getY() - p[0].getY());
269
270 double skewedStartAngle = Math.toDegrees(Math.atan2(-(p[2].getY() - centerY) * width,
271 (p[2].getX() - centerX) * height));
272 double skewedEndAngle = Math.toDegrees(Math.atan2(-(p[3].getY() - centerY) * width,
273 (p[3].getX() - centerX) * height));
274 double skewedAngleDifference = skewedEndAngle - skewedStartAngle;
275 if (skewedStartAngle < 0) skewedStartAngle += 360;
276 if (skewedAngleDifference < 0) skewedAngleDifference += 360;
277
278 Arc2D s = new Arc2D.Double(0, 0, 0, 0, skewedStartAngle, skewedAngleDifference, Arc2D.OPEN);
279 s.setFrameFromDiagonal(p[0], p[1]);
280
281 GeneralPath g = new GeneralPath();
282 g.append(s, false);
283 Rectangle2D r = new Rectangle2D.Double();
284 r.setFrameFromDiagonal(p[0], p[1]);
285 g.append(r, false);
286 Point2D center = new Point2D.Double(centerX, centerY);
287 g.append(new Line2D.Double(center, p[2]), false);
288 g.append(new Line2D.Double(center, p[3]), false);
289 return g;
290 }
291 }
292
293 /**
294 * Makes a polygon defined by six corner points.
295 */
```

(Continues)

**Listing 11.1 (Continued)**

```
296 class PolygonMaker extends ShapeMaker
297 {
298 public PolygonMaker()
299 {
300 super(6);
301 }
302
303 public Shape makeShape(Point2D[] p)
304 {
305 GeneralPath s = new GeneralPath();
306 s.moveTo((float) p[0].getX(), (float) p[0].getY());
307 for (int i = 1; i < p.length; i++)
308 s.lineTo((float) p[i].getX(), (float) p[i].getY());
309 s.closePath();
310 return s;
311 }
312 }
313
314 /**
315 * Makes a quad curve defined by two end points and a control point.
316 */
317 class QuadCurveMaker extends ShapeMaker
318 {
319 public QuadCurveMaker()
320 {
321 super(3);
322 }
323
324 public Shape makeShape(Point2D[] p)
325 {
326 return new QuadCurve2D.Double(p[0].getX(), p[0].getY(), p[1].getX(), p[1].getY(),
327 p[2].getX(), p[2].getY());
328 }
329 }
330
331 /**
332 * Makes a cubic curve defined by two end points and two control points.
333 */
334 class CubicCurveMaker extends ShapeMaker
335 {
336 public CubicCurveMaker()
337 {
338 super(4);
339 }
340 }
```

```
341 public Shape makeShape(Point2D[] p)
342 {
343 return new CubicCurve2D.Double(p[0].getX(), p[0].getY(), p[1].getX(), p[1].getY(), p[2]
344 .getX(), p[2].getY(), p[3].getX(), p[3].getY());
345 }
346 }
```

#### java.awt.geom.RoundRectangle2D.Double 1.2

- RoundRectangle2D.Double(double x, double y, double width, double height, double arcWidth, double arcHeight)

constructs a rounded rectangle with the given bounding rectangle and arc dimensions. See Figure 11.4 for an explanation of the arcWidth and arcHeight parameters.

#### java.awt.geom.Arc2D.Double 1.2

- Arc2D.Double(double x, double y, double w, double h, double startAngle, double arcAngle, int type)

constructs an arc with the given bounding rectangle, start and arc angle, and arc type. The startAngle and arcAngle are explained on p. 773. The type is one of Arc2D.OPEN, Arc2D.PIE, and Arc2D.CHORD.

#### java.awt.geom.QuadCurve2D.Double 1.2

- QuadCurve2D.Double(double x1, double y1, double ctrlx, double ctrly, double x2, double y2)

constructs a quadratic curve from a start point, a control point, and an end point.

#### java.awt.geom.CubicCurve2D.Double 1.2

- CubicCurve2D.Double(double x1, double y1, double ctrlx1, double ctrly1, double ctrlx2, double ctrly2, double x2, double y2)

constructs a cubic curve from a start point, two control points, and an end point.

#### java.awt.geom.GeneralPath 1.2

- GeneralPath()

constructs an empty general path.

**java.awt.geom.Path2D.Float 6**

- `void moveTo(float x, float y)`  
makes (x, y) the *current point*—that is, the starting point of the next segment.
- `void lineTo(float x, float y)`
- `void quadTo(float ctrlx, float ctrly, float x, float y)`
- `void curveTo(float ctrl1x, float ctrl1y, float ctrl2x, float ctrl2y, float x, float y)`  
draws a line, quadratic curve, or cubic curve from the current point to the end point (x, y), and makes that end point the current point.

**java.awt.geom.Path2D 6**

- `void append(Shape s, boolean connect)`  
adds the outline of the given shape to the general path. If `connect` is true, the current point of the general path is connected to the starting point of the added shape by a straight line.
- `void closePath()`  
closes the path by drawing a straight line from the current point to the first point in the path.

## 11.3 Areas

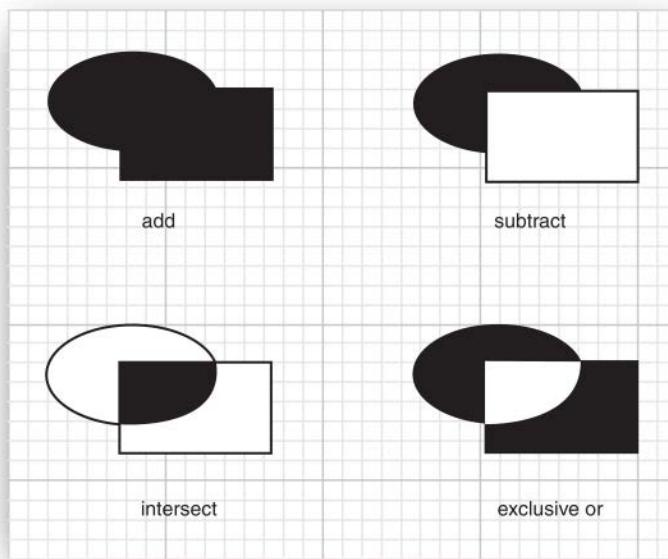
In the preceding section, you saw how you can specify complex shapes by constructing general paths composed of lines and curves. By using a sufficient number of lines and curves, you can draw essentially any shape. For example, the shapes of characters in the fonts that you see on the screen and on your printouts are all made up of lines and cubic curves.

Occasionally, it is easier to describe a shape by composing it from *areas*, such as rectangles, polygons, or ellipses. The Java 2D API supports four *constructive area geometry* operations that combine two areas into a new area.

- `add`: The combined area contains all points that are in the first or the second area.
- `subtract`: The combined area contains all points that are in the first but not the second area.
- `intersect`: The combined area contains all points that are in the first and the second area.

- `exclusiveOr`: The combined area contains all points that are in either the first or the second area, but not in both.

Figure 11.10 shows these operations.



**Figure 11.10** Constructive area geometry operations

To construct a complex area, start with a default area object.

```
Area a = new Area();
```

Then, combine the area with any shape.

```
a.add(new Rectangle2D.Double(. . .));
a.subtract(path);
. . .
```

The `Area` class implements the `Shape` interface. You can stroke the boundary of the area with the `draw` method or paint the interior with the `fill` method of the `Graphics2D` class.

**java.awt.geom.Area**

- void add(Area other)
- void subtract(Area other)
- void intersect(Area other)
- void exclusiveOr(Area other)

carries out the constructive area geometry operation with this area and the other area and sets this area to the result.

## 11.4 Strokes

The `draw` operation of the `Graphics2D` class draws the boundary of a shape by using the currently selected *stroke*. By default, the stroke is a solid line that is 1 pixel wide. You can select a different stroke by calling the `setStroke` method and supplying an object of a class that implements the `Stroke` interface. The Java 2D API defines only one such class, called `BasicStroke`. In this section, we'll look at the capabilities of the `BasicStroke` class.

You can construct strokes of arbitrary thickness. For example, here is how to draw lines that are 10 pixels wide:

```
g2.setStroke(new BasicStroke(10.0F));
g2.draw(new Line2D.Double(. . .));
```

When a stroke is more than a pixel thick, the *end* of the stroke can have different styles. Figure 11.11 shows these so-called end cap styles. You have three choices:

- A *butt cap* simply ends the stroke at its end point.
- A *round cap* adds a half-circle to the end of the stroke.
- A *square cap* adds a half-square to the end of the stroke.

When two thick strokes meet, there are three choices for the *join style* (see Figure 11.12).

- A *bevel join* joins the strokes with a straight line that is perpendicular to the bisector of the angle between the two strokes.
- A *round join* extends each stroke to have a round cap.
- A *miter join* extends both strokes by adding a “spike.”

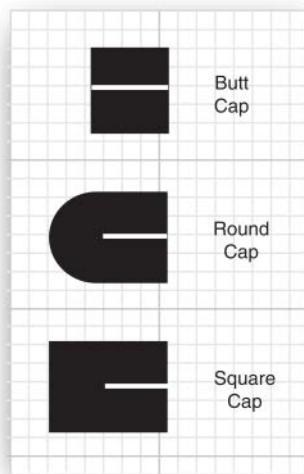


Figure 11.11 End cap styles

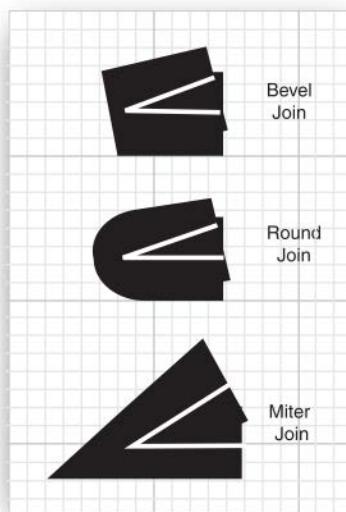


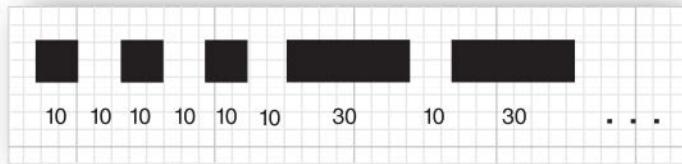
Figure 11.12 Join styles

The miter join is not suitable for lines that meet at small angles. If two lines join with an angle that is less than the *miter limit*, a bevel join is used instead, which prevents extremely long spikes. By default, the miter limit is 10 degrees.

You can specify these choices in the `BasicStroke` constructor, for example:

```
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER,
 15.0F /* miter limit */));
```

Finally, you can create dashed lines by setting a *dash pattern*. In the program in Listing 11.2, you can select a dash pattern that spells out SOS in Morse code. The dash pattern is a `float[]` array that contains the lengths of the “on” and “off” intervals (see Figure 11.13).



**Figure 11.13** A dash pattern

You can specify the dash pattern and a *dash phase* when constructing the `BasicStroke`. The dash phase indicates where in the dash pattern each line should start. Normally, you set this value to 0.

```
float[] dashPattern = { 10, 10, 10, 10, 10, 10, 30, 10, 30, . . . };
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER,
 10.0F /* miter limit */, dashPattern, 0 /* dash phase */));
```

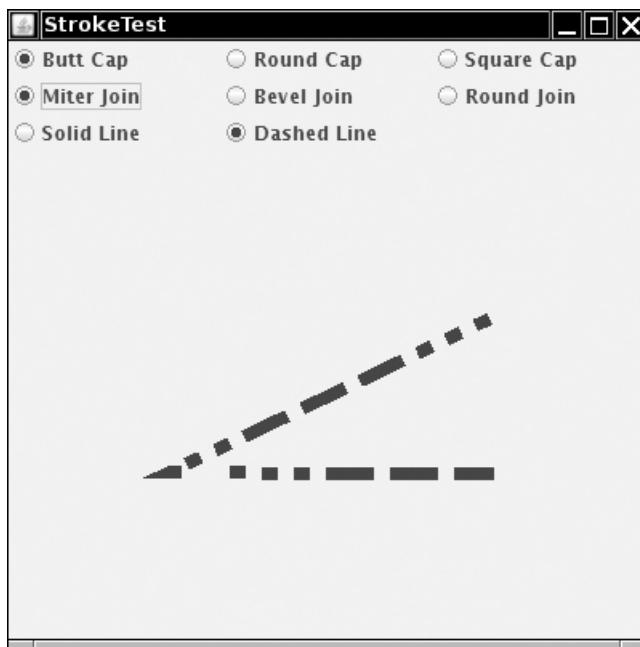
---

**NOTE:** End cap styles are applied to the ends of *each dash* in a dash pattern.

---

The program in Listing 11.2 lets you specify end cap styles, join styles, and dashed lines (see Figure 11.14). You can move the ends of the line segments to test the miter limit: Select the miter join, then move the line segment to form a very acute angle. You will see the miter join turn into a bevel join.

The program is similar to the program in Listing 11.1. The mouse listener remembers your click on the end point of a line segment, and the mouse motion listener monitors the dragging of the end point. A set of radio buttons signal the user



**Figure 11.14** The `StrokeTest` program

choices for the end cap style, join style, and solid or dashed line. The `paintComponent` method of the `StrokePanel` class constructs a `GeneralPath` consisting of the two line segments that join the three points which the user can move with the mouse. It then constructs a `BasicStroke`, according to the selections the user made, and finally draws the path.

**Listing 11.2** `stroke/StrokeTest.java`

```
1 package stroke;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.geom.*;
6 import javax.swing.*;
7
8 /**
9 * This program demonstrates different stroke types.
10 * @version 1.04 2016-05-10
11 * @author Cay Horstmann
12 */
```

(Continues)

**Listing 11.2 (Continued)**

```
13 public class StrokeTest
14 {
15 public static void main(String[] args)
16 {
17 EventQueue.invokeLater(() ->
18 {
19 JFrame frame = new StrokeTestFrame();
20 frame.setTitle("StrokeTest");
21 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22 frame.setVisible(true);
23 });
24 }
25 }
26 /**
27 * This frame lets the user choose the cap, join, and line style, and shows the resulting stroke.
28 */
29
30 class StrokeTestFrame extends JFrame
31 {
32 private StrokeComponent canvas;
33 private JPanel buttonPanel;
34
35 public StrokeTestFrame()
36 {
37 canvas = new StrokeComponent();
38 add(canvas, BorderLayout.CENTER);
39
40 buttonPanel = new JPanel();
41 buttonPanel.setLayout(new GridLayout(3, 3));
42 add(buttonPanel, BorderLayout.NORTH);
43
44 ButtonGroup group1 = new ButtonGroup();
45 makeCapButton("Butt Cap", BasicStroke.CAP_BUTT, group1);
46 makeCapButton("Round Cap", BasicStroke.CAP_ROUND, group1);
47 makeCapButton("Square Cap", BasicStroke.CAP_SQUARE, group1);
48
49 ButtonGroup group2 = new ButtonGroup();
50 makeJoinButton("Miter Join", BasicStroke.JOIN_MITER, group2);
51 makeJoinButton("Bevel Join", BasicStroke.JOIN_BEVEL, group2);
52 makeJoinButton("Round Join", BasicStroke.JOIN_ROUND, group2);
53
54 ButtonGroup group3 = new ButtonGroup();
55 makeDashButton("Solid Line", false, group3);
56 makeDashButton("Dashed Line", true, group3);
57 }
58
59 /**
60 * Makes a radio button to change the cap style.
```

```
61 * @param label the button label
62 * @param style the cap style
63 * @param group the radio button group
64 */
65 private void makeCapButton(String label, final int style, ButtonGroup group)
66 {
67 // select first button in group
68 boolean selected = group.getButtonCount() == 0;
69 JRadioButton button = new JRadioButton(label, selected);
70 buttonPanel.add(button);
71 group.add(button);
72 button.addActionListener(event -> canvas.setCap(style));
73 pack();
74 }
75
76 /**
77 * Makes a radio button to change the join style.
78 * @param label the button label
79 * @param style the join style
80 * @param group the radio button group
81 */
82 private void makeJoinButton(String label, final int style, ButtonGroup group)
83 {
84 // select first button in group
85 boolean selected = group.getButtonCount() == 0;
86 JRadioButton button = new JRadioButton(label, selected);
87 buttonPanel.add(button);
88 group.add(button);
89 button.addActionListener(event -> canvas.setJoin(style));
90 }
91
92 /**
93 * Makes a radio button to set solid or dashed lines
94 * @param label the button label
95 * @param style false for solid, true for dashed lines
96 * @param group the radio button group
97 */
98 private void makeDashButton(String label, final boolean style, ButtonGroup group)
99 {
100 // select first button in group
101 boolean selected = group.getButtonCount() == 0;
102 JRadioButton button = new JRadioButton(label, selected);
103 buttonPanel.add(button);
104 group.add(button);
105 button.addActionListener(event -> canvas.setDash(style));
106 }
107 }
108
109 /**
```

(Continues)

**Listing 11.2 (Continued)**

```
110 * This component draws two joined lines, using different stroke objects, and allows the user to
111 * drag the three points defining the lines.
112 */
113 class StrokeComponent extends JPanel
114 {
115 private static final Dimension PREFERRED_SIZE = new Dimension(400, 400);
116 private static int SIZE = 10;
117
118 private Point2D[] points;
119 private int current;
120 private float width;
121 private int cap;
122 private int join;
123 private boolean dash;
124
125 public StrokeComponent()
126 {
127 addMouseListener(new MouseAdapter()
128 {
129 public void mousePressed(MouseEvent event)
130 {
131 Point p = event.getPoint();
132 for (int i = 0; i < points.length; i++)
133 {
134 double x = points[i].getX() - SIZE / 2;
135 double y = points[i].getY() - SIZE / 2;
136 Rectangle2D r = new Rectangle2D.Double(x, y, SIZE, SIZE);
137 if (r.contains(p))
138 {
139 current = i;
140 return;
141 }
142 }
143 }
144
145 public void mouseReleased(MouseEvent event)
146 {
147 current = -1;
148 }
149 });
150
151 addMouseMotionListener(new MouseMotionAdapter()
152 {
153 public void mouseDragged(MouseEvent event)
154 {
155 if (current == -1) return;
156 points[current] = event.getPoint();
157 repaint();
158 }
159 });
160 }
161 }
```

```
158 }
159 });
160
161 points = new Point2D[3];
162 points[0] = new Point2D.Double(200, 100);
163 points[1] = new Point2D.Double(100, 200);
164 points[2] = new Point2D.Double(200, 200);
165 current = -1;
166 width = 8.0F;
167 }
168
169 public void paintComponent(Graphics g)
170 {
171 Graphics2D g2 = (Graphics2D) g;
172 GeneralPath path = new GeneralPath();
173 path.moveTo((float) points[0].getX(), (float) points[0].getY());
174 for (int i = 1; i < points.length; i++)
175 path.lineTo((float) points[i].getX(), (float) points[i].getY());
176 BasicStroke stroke;
177 if (dash)
178 {
179 float miterLimit = 10.0F;
180 float[] dashPattern = { 10F, 10F, 10F, 10F, 10F, 10F, 30F, 10F, 30F, 10F, 10F,
181 10F, 10F, 10F, 10F, 30F };
182 float dashPhase = 0;
183 stroke = new BasicStroke(width, cap, join, miterLimit, dashPattern, dashPhase);
184 }
185 else stroke = new BasicStroke(width, cap, join);
186 g2.setStroke(stroke);
187 g2.draw(path);
188 }
189 /**
190 * Sets the join style.
191 * @param j the join style
192 */
193 public void setJoin(int j)
194 {
195 join = j;
196 repaint();
197 }
198 /**
199 * Sets the cap style.
200 * @param c the cap style
201 */
202 public void setCap(int c)
203 {
204 cap = c;
```

(Continues)

**Listing 11.2 (Continued)**

```
207 repaint();
208 }
209
210 /**
211 * Sets solid or dashed lines.
212 * @param d false for solid, true for dashed lines
213 */
214 public void setDash(boolean d)
215 {
216 dash = d;
217 repaint();
218 }
219
220 public Dimension getPreferredSize() { return PREFERRED_SIZE; }
221 }
```

---

**java.awt.Graphics2D 1.2**

- `void setStroke(Stroke s)`

sets the stroke of this graphics context to the given object that implements the `Stroke` interface.

**java.awt.BasicStroke 1.2**

- `BasicStroke(float width)`
- `BasicStroke(float width, int cap, int join)`
- `BasicStroke(float width, int cap, int join, float miterlimit)`
- `BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dashPhase)`

constructs a stroke object with the given attributes.

|                    |                         |                                                                                                                                                             |
|--------------------|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Parameters:</i> | <code>width</code>      | The width of the pen                                                                                                                                        |
|                    | <code>cap</code>        | The end cap style—one of <code>CAP_BUTT</code> , <code>CAP_ROUND</code> , and <code>CAP_SQUARE</code>                                                       |
|                    | <code>join</code>       | The join style—one of <code>JOIN_BEVEL</code> , <code>JOIN_MITER</code> , and <code>JOIN_ROUND</code>                                                       |
|                    | <code>miterlimit</code> | The angle, in degrees, below which a miter join is rendered as a bevel join                                                                                 |
|                    | <code>dash</code>       | An array of the lengths of the alternating filled and blank portions of a dashed stroke                                                                     |
|                    | <code>dashPhase</code>  | The “phase” of the dash pattern; a segment of this length, preceding the starting point of the stroke, is assumed to have the dash pattern already applied. |

## 11.5 Paint

When you fill a shape, its inside is covered with *paint*. Use the `setPaint` method to set the paint style to an object whose class implements the `Paint` interface. The Java 2D API provides three such classes:

- The `Color` class implements the `Paint` interface. To fill shapes with a solid color, simply call `setPaint` with a `Color` object, such as

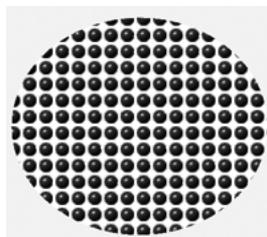
```
g2.setPaint(Color.red);
```

- The `GradientPaint` class varies colors by interpolating between two given color values (see Figure 11.15).



**Figure 11.15** Gradient paint

- The `TexturePaint` class fills an area with repetitions of an image (see Figure 11.16).



**Figure 11.16** Texture paint

You can construct a `GradientPaint` object by specifying two points and the colors that you want at these two points.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW));
```

Colors are interpolated along the line joining the two points. Colors are constant along lines perpendicular to that joining line. Points beyond an end point of the line are given the color at the end point.

Alternatively, if you call the `GradientPaint` constructor with `true` for the `cyclic` parameter,

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW, true));
```

then the color variation *cycles* and keeps varying beyond the end points.

To construct a `TexturePaint` object, specify a `BufferedImage` and an *anchor* rectangle.

```
g2.setPaint(new TexturePaint(bufferedImage, anchorRectangle));
```

We will introduce the `BufferedImage` class later in this chapter when we discuss images in detail. The simplest way of obtaining a buffered image is to read an image file:

```
bufferedImage = ImageIO.read(new File("blue-ball.gif"));
```

The anchor rectangle is extended indefinitely in *x* and *y* directions to tile the entire coordinate plane. The image is scaled to fit into the anchor and then replicated into each tile.

#### java.awt.Graphics2D 1.2

- `void setPaint(Paint s)`

sets the paint of this graphics context to the given object that implements the `Paint` interface.

#### java.awt.GradientPaint 1.2

- `GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2)`
- `GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2, boolean cyclic)`
- `GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2)`
- `GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2, boolean cyclic)`

constructs a gradient paint object that fills shapes with color such that the start point is colored with `color1`, the end point is colored with `color2`, and the colors in between are linearly interpolated. Colors are constant along lines perpendicular to the line joining the start and the end point. By default, the gradient paint is not cyclic—that is, points beyond the start and end points are colored with the same color as the start and end point. If the gradient paint is *cyclic*, then colors continue to be interpolated, first returning to the starting point color and then repeating indefinitely in both directions.

**java.awt.TexturePaint 1.2**

- `TexturePaint(BufferedImage texture, Rectangle2D anchor)`

creates a texture paint object. The anchor rectangle defines the tiling of the space to be painted; it is repeated indefinitely in *x* and *y* directions, and the texture image is scaled to fill each tile.

## 11.6 Coordinate Transformations

Suppose you need to draw an object, such as an automobile. You know, from the manufacturer's specifications, the height, wheelbase, and total length. You could, of course, figure out all pixel positions, assuming some number of pixels per meter. However, there is an easier way: You can ask the graphics context to carry out the conversion for you.

```
g2.scale(pixelsPerMeter, pixelsPerMeter);
g2.draw(new Line2D.Double(coordinates in meters)); // converts to pixels and draws scaled line
```

The `scale` method of the `Graphics2D` class sets the *coordinate transformation* of the graphics context to a scaling transformation. That transformation changes *user coordinates* (user-specified units) to *device coordinates* (pixels). Figure 11.17 shows how the transformation works.

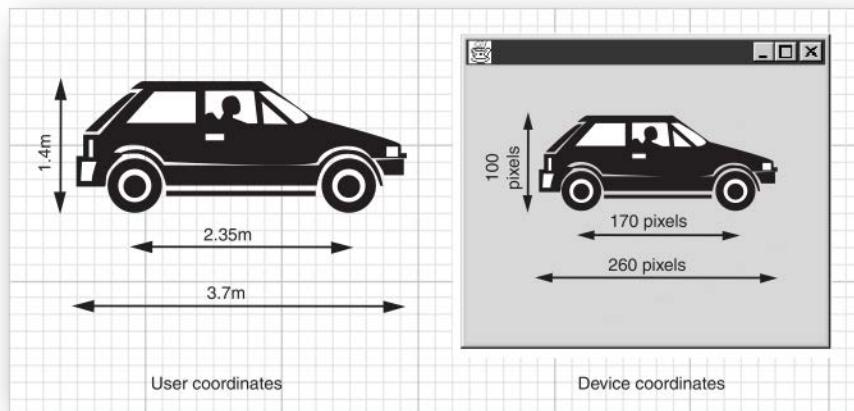


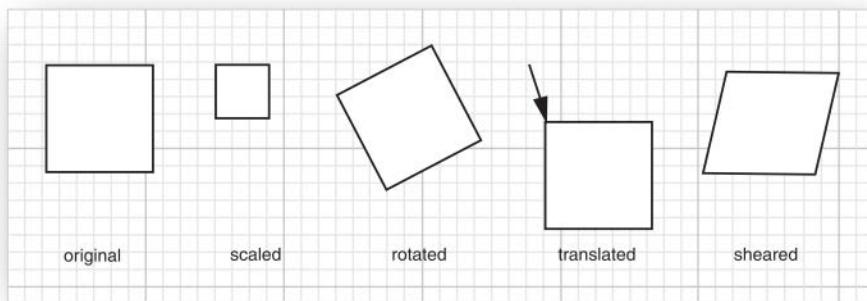
Figure 11.17 User and device coordinates

Coordinate transformations are very useful in practice. They allow you to work with convenient coordinate values. The graphics context takes care of the dirty work of transforming them to pixels.

There are four fundamental transformations.

- Scaling: blowing up, or shrinking, all distances from a fixed point
- Rotation: rotating all points around a fixed center
- Translation: moving all points by a fixed amount
- Shear: leaving one line fixed and “sliding” the lines parallel to it by an amount that is proportional to the distance from the fixed line

Figure 11.18 shows how these four fundamental transformations act on a unit square.



**Figure 11.18** The fundamental transformations

The `scale`, `rotate`, `translate`, and `shear` methods of the `Graphics2D` class set the coordinate transformation of the graphics context to one of these fundamental transformations.

You can *compose* the transformations. For example, you might want to rotate shapes *and* double their size; supply both a rotation and a scaling transformation:

```
g2.rotate(angle);
g2.scale(2, 2);
g2.draw(. . .);
```

In this case, it does not matter in which order you supply the transformations. However, with most transformations, order does matter. For example, if you want to rotate and shear, then it makes a difference which of the transformations you supply first. You need to figure out what your intention is. The graphics

context will apply the transformations in the order opposite to that in which you supplied them—that is, the last transformation you supply is applied first.

You can supply as many transformations as you like. For example, consider the following sequence of transformations:

```
g2.translate(x, y);
g2.rotate(a);
g2.translate(-x, -y);
```

The last transformation (which is applied first) moves the point  $(x, y)$  to the origin. The second transformation rotates with an angle  $a$  around the origin. The final transformation moves the origin back to  $(x, y)$ . The overall effect is a rotation with center point  $(x, y)$ —see Figure 11.19. Since rotating about a point other than the origin is such a common operation, there is a shortcut:

```
g2.rotate(a, x, y);
```

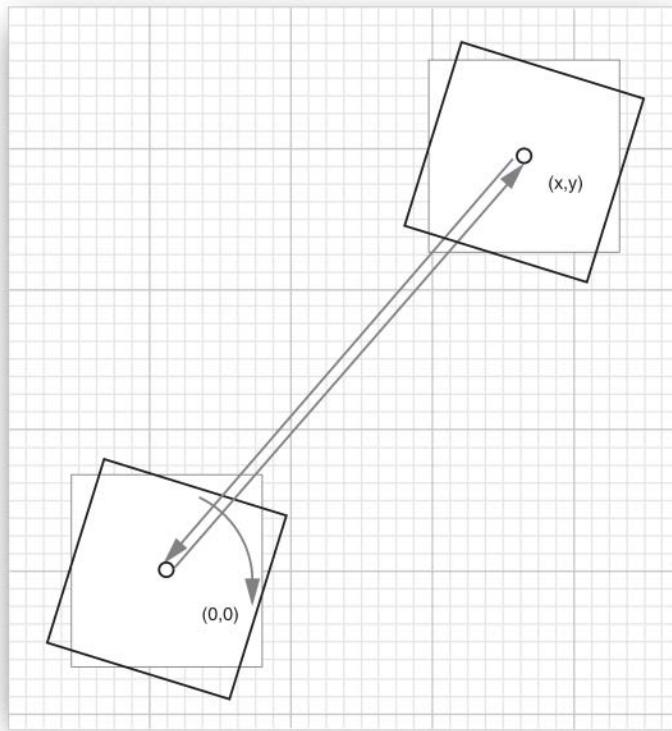


Figure 11.19 Composing transformations

If you know some matrix theory, you are probably aware that all rotations, translations, scalings, shears, and their compositions can be expressed by transformation matrices of the form:

$$\begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Such a transformation is called an *affine transformation*. In the Java 2D API, the `AffineTransform` class describes such a transformation. If you know the components of a particular transformation matrix, you can construct it directly as

```
AffineTransform t = new AffineTransform(a, b, c, d, e, f);
```

Additionally, the factory methods `getRotateInstance`, `getScaleInstance`, `getTranslateInstance`, and `getShearInstance` construct the matrices that represent these transformation types. For example, the call

```
t = AffineTransform.getScaleInstance(2.0F, 0.5F);
```

returns a transformation that corresponds to the matrix

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Finally, the instance methods `setToRotation`, `setToScale`, `setToTranslation`, and `setToShear` set a transformation object to a new type. Here is an example:

```
t.setToRotation(angle); // sets t to a rotation
```

You can set the coordinate transformation of the graphics context to an `AffineTransform` object.

```
g2.setTransform(t); // replaces current transformation
```

However, in practice, you shouldn't call the `setTransform` operation, as it replaces any existing transformation that the graphics context may have. For example, a graphics context for printing in landscape mode already contains a 90-degree rotation transformation. If you call `setTransform`, you obliterate that rotation. Instead, call the `transform` method.

```
g2.transform(t); // composes current transformation with t
```

It composes the existing transformation with the new `AffineTransform` object.

If you just want to apply a transformation temporarily, first get the old transformation, compose it with your new transformation, and finally restore the old transformation when you are done.

```
AffineTransform oldTransform = g2.getTransform(); // save old transform
g2.transform(t); // apply temporary transform
draw on g2
g2.setTransform(oldTransform); // restore old transform
```

**java.awt.geom.AffineTransform 1.2**

- `AffineTransform(double a, double b, double c, double d, double e, double f)`
- `AffineTransform(float a, float b, float c, float d, float e, float f)`

constructs the affine transform with matrix

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

- `AffineTransform(double[] m)`
- `AffineTransform(float[] m)`

constructs the affine transform with matrix

$$\begin{bmatrix} m[0] & m[2] & m[4] \\ m[1] & m[3] & m[5] \\ 0 & 0 & 1 \end{bmatrix}$$

- `static AffineTransform getRotateInstance(double a)`

creates a rotation around the origin by the angle a (in radians). The transformation matrix is

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If a is between 0 and  $\pi/2$ , the rotation moves the positive  $x$  axis toward the positive  $y$  axis.

- `static AffineTransform getRotateInstance(double a, double x, double y)`

creates a rotation around the point  $(x,y)$  by the angle a (in radians).

- `static AffineTransform getScaleInstance(double sx, double sy)`

creates a scaling transformation that scales the  $x$  axis by sx and the  $y$  axis by sy. The transformation matrix is

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

*(Continues)*

**java.awt.geom.AffineTransform 1.2 (Continued)**

- static AffineTransform getShearInstance(double shx, double shy)

creates a shear transformation that shears the  $x$  axis by  $shx$  and the  $y$  axis by  $shy$ . The transformation matrix is

$$\begin{bmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- static AffineTransform getTranslateInstance(double tx, double ty)

creates a translation that moves the  $x$  axis by  $tx$  and the  $y$  axis by  $ty$ . The transformation matrix is

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

- void setToRotation(double a)
- void setToRotation(double a, double x, double y)
- void setToScale(double sx, double sy)
- void setToShear(double sx, double sy)
- void setToTranslation(double tx, double ty)

sets this affine transformation to a basic transformation with the given parameters. See the `getXxxInstance` methods for an explanation of the basic transformations and their parameters.

**java.awt.Graphics2D 1.2**

- void setTransform(AffineTransform t)

replaces the existing coordinate transformation of this graphics context with  $t$ .

- void transform(AffineTransform t)

composes the existing coordinate transformation of this graphics context with  $t$ .

- void rotate(double a)
- void rotate(double a, double x, double y)
- void scale(double sx, double sy)
- void shear(double sx, double sy)
- void translate(double tx, double ty)

composes the existing coordinate transformation of this graphics context with a basic transformation with the given parameters. See the `AffineTransform.getXxxInstance` methods for an explanation of the basic transformations and their parameters.

## 11.7 Clipping

By setting a *clipping shape* in the graphics context, you constrain all drawing operations to the interior of that clipping shape.

```
g2.setClip(clipShape); // but see below
g2.draw(shape); // draws only the part that falls inside the clipping shape
```

However, in practice, you don't want to call the `setClip` operation because it replaces any existing clipping shape that the graphics context might have. For example, as you will see later in this chapter, a graphics context for printing comes with a clip rectangle that ensures that you don't draw on the margins. Instead, call the `clip` method.

```
g2.clip(clipShape); // better
```

The `clip` method intersects the existing clipping shape with the new one that you supply.

If you just want to apply a clipping area temporarily, you should first get the old clip, add your new clip, and finally restore the old clip when you are done:

```
Shape oldClip = g2.getClip(); // save old clip
g2.clip(clipShape); // apply temporary clip
draw on g2
g2.setClip(oldClip); // restore old clip
```

In Figure 11.20, we show off the clipping capability with a rather dramatic drawing of a line pattern clipped by a complex shape—namely, the outline of a set of letters.



Figure 11.20 Using letter shapes to clip a line pattern

To obtain the character outlines, you need a *font render context*. Use the `getFontRenderContext` method of the `Graphics2D` class.

```
FontRenderContext context = g2.getFontRenderContext();
```

Next, using a string, a font, and the font render context, create a `TextLayout` object:

```
TextLayout layout = new TextLayout("Hello", font, context);
```

This text layout object describes the layout of a sequence of characters, as rendered by a particular font render context. The layout depends on the font render context—the same characters will look different on a screen and on a printer.

More important for our application, the `getOutline` method returns a `Shape` object that describes the shape of the outline of the characters in the text layout. The outline shape starts at the origin (0, 0), which might not be what you want. In that case, supply an affine transform to the `getOutline` operation to specify where you would like the outline to appear.

```
AffineTransform transform = AffineTransform.getTranslateInstance(0, 100);
Shape outline = layout.getOutline(transform);
```

Then, append the outline to the clipping shape.

```
GeneralPath clipShape = new GeneralPath();
clipShape.append(outline, false);
```

Finally, set the clipping shape and draw a set of lines. The lines appear only inside the character boundaries.

```
g2.setClip(clipShape);
Point2D p = new Point2D.Double(0, 0);
for (int i = 0; i < NLINES; i++)
{
 double x = . . .;
 double y = . . .;
 Point2D q = new Point2D.Double(x, y);
 g2.draw(new Line2D.Double(p, q)); // lines are clipped
}
```

#### java.awt.Graphics 1.0

- `void setClip(Shape s)` **1.2**

sets the current clipping shape to the shape `s`.

- `Shape getClip()` **1.2**

returns the current clipping shape.

**java.awt.Graphics2D 1.2**

- void clip(Shape s)  
intersects the current clipping shape with the shape s.
- FontRenderContext getFontRenderContext()  
returns a font render context that is necessary for constructing TextLayout objects.

**java.awt.font.TextLayout 1.2**

- TextLayout(String s, Font f, FontRenderContext context)  
constructs a text layout object from a given string and font, using the font render context to obtain font properties for a particular device.
- float getAdvance()  
returns the width of this text layout.
- float getAscent()
- float getDescent()  
returns the height of this text layout above and below the baseline.
- float getLeading()  
returns the distance between successive lines in the font used by this text layout.

## 11.8 Transparency and Composition

In the standard RGB color model, every color is described by its red, green, and blue components. However, it is also convenient to describe areas of an image that are *transparent* or partially transparent. When you superimpose an image onto an existing drawing, the transparent pixels do not obscure the pixels under them at all, whereas partially transparent pixels are mixed with the pixels under them. Figure 11.21 shows the effect of overlaying a partially transparent rectangle on an image. You can still see the details of the image shine through from under the rectangle.

In the Java 2D API, transparency is described by an *alpha channel*. Each pixel has, in addition to its red, green, and blue color components, an alpha value between 0 (fully transparent) and 1 (fully opaque). For example, the rectangle in Figure 11.21 was filled with a pale yellow color with 50% transparency:

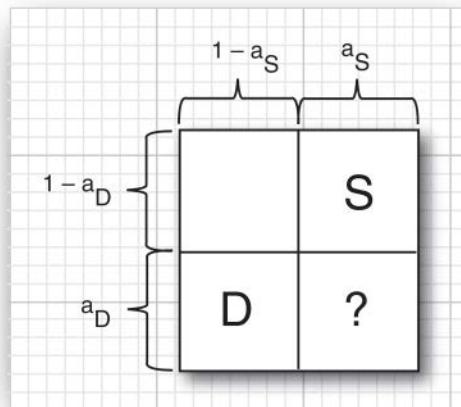
```
new Color(0.7F, 0.7F, 0.0F, 0.5F);
```



**Figure 11.21** Overlaying a partially transparent rectangle on an image

Now let us look at what happens if you superimpose two shapes. You need to blend or *compose* the colors and alpha values of the source and destination pixels. Porter and Duff, two researchers in the field of computer graphics, have formulated 12 possible *composition rules* for this blending process. The Java 2D API implements all of these rules. Before going any further, we'd like to point out that only two of these rules have practical significance. If you find the rules arcane or confusing, just use the `SRC_OVER` rule. It is the default rule for a `Graphics2D` object, and it gives the most intuitive results.

Here is the theory behind the rules. Suppose you have a *source pixel* with alpha value  $a_S$ . In the image, there is already a *destination pixel* with alpha value  $a_D$ . You want to compose the two. The diagram in Figure 11.22 shows how to design a composition rule.



**Figure 11.22** Designing a composition rule

Porter and Duff consider the alpha value as the probability that the pixel color should be used. From the perspective of the source, there is a probability  $a_S$  that it wants to use the source color and a probability of  $1 - a_S$  that it doesn't care. The same holds for the destination. When composing the colors, let us assume that the probabilities are independent. Then there are four cases, as shown in Figure 11.22. If the source wants to use the source color and the destination doesn't care, then it seems reasonable to let the source have its way. That's why the upper right corner of the diagram is labeled "S". The probability for that event is  $a_S \cdot (1 - a_D)$ . Similarly, the lower left corner is labeled "D". What should one do if both destination and source would like to select their color? That's where the Porter–Duff rules come in. If we decide that the source is more important, we label the lower right corner with an "S" as well. That rule is called `SRC_OVER`. In that rule, you combine the source colors with a weight of  $a_S$  and the destination colors with a weight of  $(1 - a_S) \cdot a_D$ .

The visual effect is a blending of the source and destination, with preference given to the source. In particular, if  $a_S$  is 1, then the destination color is not taken into account at all. If  $a_S$  is 0, then the source pixel is completely transparent and the destination color is unchanged.

The other rules depend on what letters you put in the boxes of the probability diagram. Table 11.1 and Figure 11.23 show all rules that are supported by the Java 2D API. The images in the figure show the results of the rules when a rectangular source region with an alpha of 0.75 is combined with an elliptical destination region with an alpha of 1.0.

As you can see, most of the rules aren't very useful. Consider, as an extreme case, the `DST_IN` rule. It doesn't take the source color into account at all, but it uses the alpha of the source to affect the destination. The `SRC` rule is potentially useful—it forces the source color to be used, turning off blending with the destination.

For more information on the Porter–Duff rules, see, for example, *Computer Graphics: Principles and Practice, Second Edition in C*, by James D. Foley, Andries van Dam, Steven K. Feiner, et al.

Use the `setComposite` method of the `Graphics2D` class to install an object of a class that implements the `Composite` interface. The Java 2D API supplies one such class, `AlphaComposite`, that implements all the Porter–Duff rules in Figure 11.23.

The factory method `getInstance` of the `AlphaComposite` class yields an `AlphaComposite` object. You supply the rule and the alpha value to be used for source pixels. For example, consider the following code:

**Table 11.1** The Porter–Duff Composition Rules

| Rule     | Explanation                                                                   |
|----------|-------------------------------------------------------------------------------|
| CLEAR    | Source clears destination.                                                    |
| SRC      | Source overwrites destination and empty pixels.                               |
| DST      | Source does not affect destination.                                           |
| SRC_OVER | Source blends with destination and overwrites empty pixels.                   |
| DST_OVER | Source does not affect destination and overwrites empty pixels.               |
| SRC_IN   | Source overwrites destination.                                                |
| SRC_OUT  | Source clears destination and overwrites empty pixels.                        |
| DST_IN   | Source alpha modifies destination.                                            |
| DST_OUT  | Source alpha complement modifies destination.                                 |
| SRC_ATOP | Source blends with destination.                                               |
| DST_ATOP | Source alpha modifies destination. Source overwrites empty pixels.            |
| XOR      | Source alpha complement modifies destination. Source overwrites empty pixels. |

```

int rule = AlphaComposite.SRC_OVER;
float alpha = 0.5f;
g2.setComposite(AlphaComposite.getInstance(rule, alpha));
g2.setPaint(Color.blue);
g2.fill(rectangle);

```

The rectangle is then painted with blue color and an alpha value of 0.5. Since the composition rule is SRC\_OVER, it is transparently overlaid on the existing image.

The program in Listing 11.3 lets you explore these composition rules. Pick a rule from the combo box and use the slider to set the alpha value of the `AlphaComposite` object.

Furthermore, the program displays a verbal description of each rule. Note that the descriptions are computed from the composition rule diagrams. For example, a "DS" in the second row stands for "blends with destination."

The program has one important twist. There is no guarantee that the graphics context that corresponds to the screen has an alpha channel. (In fact, it generally does not.) When pixels are deposited to a destination without an alpha channel, the pixel colors are multiplied with the alpha value and the alpha value is

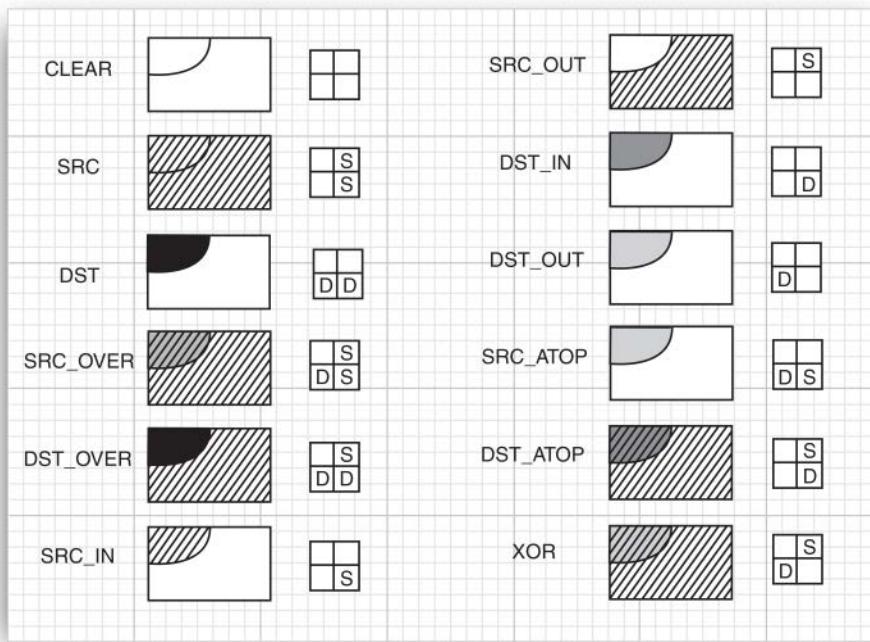
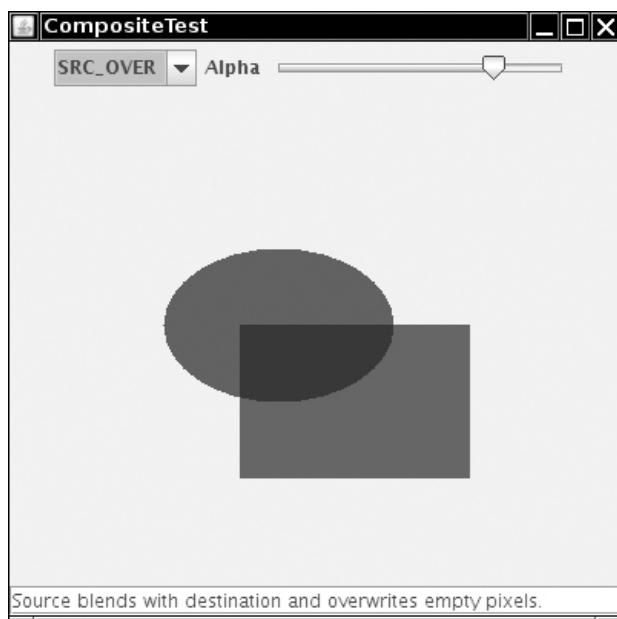


Figure 11.23 Porter–Duff composition rules

discarded. Now, several of the Porter–Duff rules use the alpha values of the destination, which means a destination alpha channel is important. For that reason, we use a buffered image with the ARGB color model to compose the shapes. After the images have been composed, we draw the resulting image to the screen.

```
BufferedImage image = new BufferedImage(getWidth(), getHeight(), BufferedImage.TYPE_INT_ARGB);
Graphics2D gImage = image.createGraphics();
// now draw to gImage
g2.drawImage(image, null, 0, 0);
```

Listings 11.3 and 11.4 show the frame and component class. The Rule class in Listing 11.5 provides a brief explanation for each rule—see Figure 11.24. As you run the program, move the alpha slider from left to right to see the effect on the composed shapes. In particular, note that the only difference between the DST\_IN and DST\_OUT rules is how the destination (!) color changes when you change the source alpha.



**Figure 11.24** The CompositeTest program

**Listing 11.3** composite/CompositeTestFrame.java

```
1 package composite;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6
7 /**
8 * This frame contains a combo box to choose a composition rule, a slider to change the source
9 * alpha channel, and a component that shows the composition.
10 */
11 class CompositeTestFrame extends JFrame
12 {
13 private static final int DEFAULT_WIDTH = 400;
14 private static final int DEFAULT_HEIGHT = 400;
15 }
```

```
16 private CompositeComponent canvas;
17 private JComboBox<Rule> ruleCombo;
18 private JSlider alphaSlider;
19 private JTextField explanation;
20
21 public CompositeTestFrame()
22 {
23 setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
24
25 canvas = new CompositeComponent();
26 add(canvas, BorderLayout.CENTER);
27
28 ruleCombo = new JComboBox<>(new Rule[] { new Rule("CLEAR", " ", " "),
29 new Rule("SRC", " S", " S"), new Rule("DST", " ", "DD"),
30 new Rule("SRC_OVER", " S", "DS"), new Rule("DST_OVER", " S", "DD"),
31 new Rule("SRC_IN", " ", " S"), new Rule("SRC_OUT", " S", " "),
32 new Rule("DST_IN", " ", " D"), new Rule("DST_OUT", " ", "D "),
33 new Rule("SRC_ATOP", " ", "DS"), new Rule("DST_ATOP", " S", " D"),
34 new Rule("XOR", " S", "D "), }));
35 ruleCombo.addActionListener(event ->
36 {
37 Rule r = (Rule) ruleCombo.getSelectedItem();
38 canvas.setRule(r.getValue());
39 explanation.setText(r.getExplanation());
40 });
41
42 alphaSlider = new JSlider(0, 100, 75);
43 alphaSlider.addChangeListener(event -> canvas.setAlpha(alphaSlider.getValue()));
44 JPanel panel1 = new JPanel();
45 panel1.add(ruleCombo);
46 panel1.add(new JLabel("Alpha"));
47 panel1.add(alphaSlider);
48 add(panel1, BorderLayout.NORTH);
49
50 explanation = new JTextField();
51 add(explanation, BorderLayout.SOUTH);
52
53 canvas.setAlpha(alphaSlider.getValue());
54 Rule r = ruleCombo.getItemAt(ruleCombo.getSelectedIndex());
55 canvas.setRule(r.getValue());
56 explanation.setText(r.getExplanation());
57 }
58 }
```

**Listing 11.4** composite/CompositeComponent.java

```
1 package composite;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.image.*;
6 import javax.swing.*;
7
8 /**
9 * This component draws two shapes, composed with a composition rule.
10 */
11 class CompositeComponent extends JPanel
12 {
13 private int rule;
14 private Shape shape1;
15 private Shape shape2;
16 private float alpha;
17
18 public CompositeComponent()
19 {
20 shape1 = new Ellipse2D.Double(100, 100, 150, 100);
21 shape2 = new Rectangle2D.Double(150, 150, 150, 100);
22 }
23
24 public void paintComponent(Graphics g)
25 {
26 Graphics2D g2 = (Graphics2D) g;
27
28 BufferedImage image = new BufferedImage(getWidth(), getHeight(),
29 BufferedImage.TYPE_INT_ARGB);
30 Graphics2D gImage = image.createGraphics();
31 gImage.setPaint(Color.red);
32 gImage.fill(shape1);
33 AlphaComposite composite = AlphaComposite.getInstance(rule, alpha);
34 gImage.setComposite(composite);
35 gImage.setPaint(Color.blue);
36 gImage.fill(shape2);
37 g2.drawImage(image, null, 0, 0);
38 }
39
40 /**
41 * Sets the composition rule.
42 * @param r the rule (as an AlphaComposite constant)
43 */
44 public void setRule(int r)
45 {
46 rule = r;
47 repaint();
48 }
```

```
49
50 /**
51 * Sets the alpha of the source.
52 * @param a the alpha value between 0 and 100
53 */
54 public void setAlpha(int a)
55 {
56 alpha = (float) a / 100.0F;
57 repaint();
58 }
59 }
```

**Listing 11.5** composite/Rule.java

```
1 package composite;
2
3 import java.awt.*;
4
5 /**
6 * This class describes a Porter-Duff rule.
7 */
8 class Rule
9 {
10 private String name;
11 private String porterDuff1;
12 private String porterDuff2;
13
14 /**
15 * Constructs a Porter-Duff rule.
16 * @param n the rule name
17 * @param pd1 the first row of the Porter-Duff square
18 * @param pd2 the second row of the Porter-Duff square
19 */
20 public Rule(String n, String pd1, String pd2)
21 {
22 name = n;
23 porterDuff1 = pd1;
24 porterDuff2 = pd2;
25 }
26
27 /**
28 * Gets an explanation of the behavior of this rule.
29 * @return the explanation
30 */
31 public String getExplanation()
32 {
```

(Continues)

**Listing 11.5 (Continued)**

```
33 StringBuilder r = new StringBuilder("Source ");
34 if (porterDuff2.equals(" ")) r.append("clears");
35 if (porterDuff2.equals(" S")) r.append("overwrites");
36 if (porterDuff2.equals("DS")) r.append("blends with");
37 if (porterDuff2.equals(" D")) r.append("alpha modifies");
38 if (porterDuff2.equals("D ")) r.append("alpha complement modifies");
39 if (porterDuff2.equals("DD")) r.append("does not affect");
40 r.append(" destination");
41 if (porterDuff1.equals(" S")) r.append(" and overwrites empty pixels");
42 r.append(".");
43 return r.toString();
44 }
45
46 public String toString()
47 {
48 return name;
49 }
50 /**
51 * Gets the value of this rule in the AlphaComposite class.
52 * @return the AlphaComposite constant value, or -1 if there is no matching constant
53 */
54 public int getValue()
55 {
56 try
57 {
58 return (Integer) AlphaComposite.class.getField(name).get(null);
59 }
60 catch (Exception e)
61 {
62 return -1;
63 }
64 }
65 }
66 }
```

---

**java.awt.Graphics2D 1.2**

- **void setComposite(Composite s)**

sets the composite of this graphics context to the given object that implements the **Composite** interface.

**java.awt.AlphaComposite 1.2**

- static AlphaComposite getInstance(int rule)
- static AlphaComposite getInstance(int rule, float sourceAlpha)

constructs an alpha composite object. The rule is one of CLEAR, SRC, SRC\_OVER, DST\_OVER, SRC\_IN, SRC\_OUT, DST\_IN, DST\_OUT, DST, DST\_ATOP, SRC\_ATOP, XOR.

## 11.9 Rendering Hints

In the preceding sections you have seen that the rendering process is quite complex. Although the Java 2D API is surprisingly fast in most cases, sometimes you would like to have control over trade-offs between speed and quality. You can achieve this by setting *rendering hints*. The `setRenderingHint` method of the `Graphics2D` class lets you set a single hint. The hints' keys and values are declared in the `RenderingHints` class. Table 11.2 summarizes the choices. The values that end in `_DEFAULT` denote the defaults that are chosen by a particular implementation as a good trade-off between performance and quality.

**Table 11.2** Rendering Hints

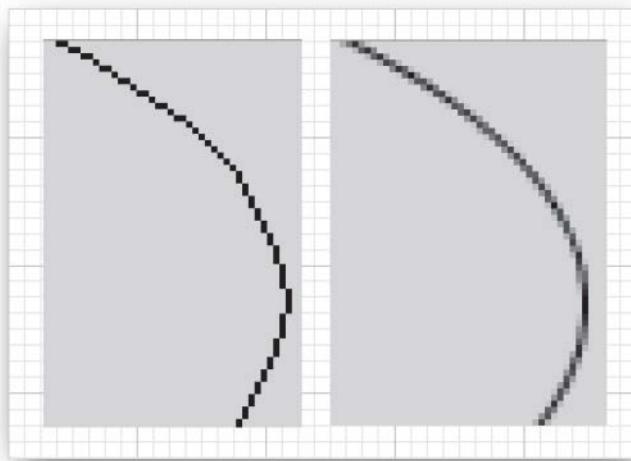
| Key                   | Value                                                                                                                                                                                                                                                          | Explanation                                                                                                                                                                                                                                                                                |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| KEY_ANTIALIASING      | VALUE_ANTIALIAS_ON<br>VALUE_ANTIALIAS_OFF<br>VALUE_ANTIALIAS_DEFAULT                                                                                                                                                                                           | Turn antialiasing for shapes on or off.                                                                                                                                                                                                                                                    |
| KEY_TEXT_ANTIALIASING | VALUE_TEXT_ANTIALIAS_ON<br>VALUE_TEXT_ANTIALIAS_OFF<br>VALUE_TEXT_ANTIALIAS_DEFAULT<br>VALUE_TEXT_ANTIALIAS_GASP 6<br>VALUE_TEXT_ANTIALIAS_LCD_HRGB 6<br>VALUE_TEXT_ANTIALIAS_LCD_HBGR 6<br>VALUE_TEXT_ANTIALIAS_LCD_VRGB 6<br>VALUE_TEXT_ANTIALIAS_LCD_VBGR 6 | Turn antialiasing for fonts on or off. When using the value <code>VALUE_TEXT_ANTIALIAS_GASP</code> , the “gasp table” of the font is consulted to decide whether a particular size of a font should be antialiased. The LCD values force subpixel rendering for a particular display type. |
| KEY_FRACTIONALMETRICS | VALUE_FRACTIONALMETRICS_ON<br>VALUE_FRACTIONALMETRICS_OFF<br>VALUE_FRACTIONALMETRICS_DEFAULT                                                                                                                                                                   | Turn the computation of fractional character dimensions on or off. Fractional character dimensions lead to better placement of characters.                                                                                                                                                 |

(Continues)

**Table 11.2** (Continued)

| Key                     | Value                                                                                                     | Explanation                                                                                                                                                                                                                 |
|-------------------------|-----------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| KEY_RENDERING           | VALUE_RENDER_QUALITY<br>VALUE_RENDER_SPEED<br>VALUE_RENDER_DEFAULT                                        | When available, select the rendering algorithm for greater quality or speed.                                                                                                                                                |
| KEY_STROKE_CONTROL 1.3  | VALUE_STROKE_NORMALIZE<br>VALUE_STROKE_PURE<br>VALUE_STROKE_DEFAULT                                       | Select whether the placement of strokes is controlled by the graphics accelerator (which may move it by up to half a pixel) or is computed by the “pure” rule that mandates that strokes run through the centers of pixels. |
| KEY_DITHERING           | VALUE_DITHER_ENABLE<br>VALUE_DITHER_DISABLE<br>VALUE_DITHER_DEFAULT                                       | Turn dithering for colors on or off. Dithering approximates color values by drawing groups of pixels of similar colors. (Note that antialiasing can interfere with dithering.)                                              |
| KEY_ALPHA_INTERPOLATION | VALUE_ALPHA_INTERPOLATION_QUALITY<br>VALUE_ALPHA_INTERPOLATION_SPEED<br>VALUE_ALPHA_INTERPOLATION_DEFAULT | Turn precise computation of alpha composites on or off.                                                                                                                                                                     |
| KEY_COLOR_RENDERING     | VALUE_COLOR_RENDER_QUALITY<br>VALUE_COLOR_RENDER_SPEED<br>VALUE_COLOR_RENDER_DEFAULT                      | Select quality or speed for color rendering. This is only an issue when you use different color spaces.                                                                                                                     |
| KEY_INTERPOLATION       | VALUE_INTERPOLATION_NEAREST_NEIGHBOR<br>VALUE_INTERPOLATION_BILINEAR<br>VALUE_INTERPOLATION_BICUBIC       | Select a rule for interpolating pixels when scaling or rotating images.                                                                                                                                                     |

The most useful of these settings involves *antialiasing*. This technique removes the “jaggies” from slanted lines and curves. As you can see in Figure 11.25, a slanted line must be drawn as a “staircase” of pixels. Especially on low-resolution screens, this line can look ugly. But if, instead of drawing each pixel completely on or off, you color in the pixels that are partially covered with the color value proportional to the area of the pixel that the line covers, then the result looks much smoother. This technique is called antialiasing. Of course, antialiasing takes a bit longer because it has to compute all those color values.



**Figure 11.25** Antialiasing

For example, here is how you can request the use of antialiasing:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
```

It also makes sense to use antialiasing for fonts.

```
g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
 RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
```

The other rendering hints are not as commonly used.

You can also put a bunch of key/value hint pairs into a map and set them all at once by calling the `setRenderingHints` method. Any collection class implementing the map interface will do, but you might as well use the `RenderingHints` class itself. It implements the `Map` interface and supplies a default map implementation if you pass `null` to the constructor. For example,

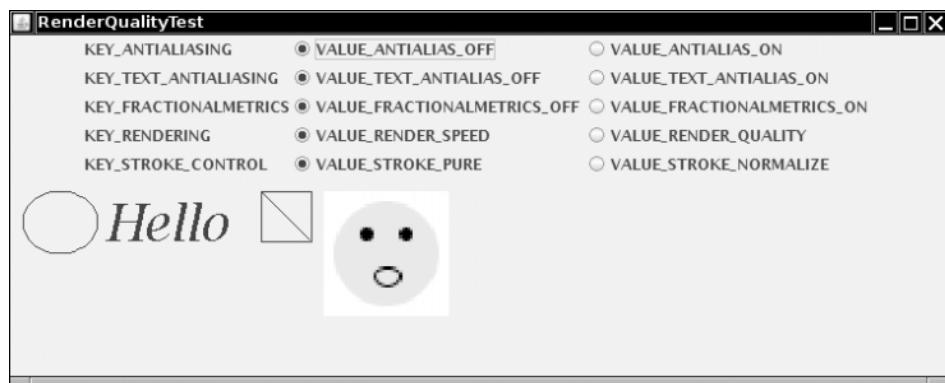
```
RenderingHints hints = new RenderingHints(null);
hints.put(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
hints.put(RenderingHints.KEY_TEXT_ANTIALIASING, RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
g2.setRenderingHints(hints);
```

That is the technique we use in Listing 11.6. The program shows several rendering hints that we found beneficial. Note the following:

- Antialiasing smooths the ellipse.
- Text antialiasing smooths the text.

- On some platforms, fractional text metrics move the letters a bit closer together.
- Selecting `VALUE_RENDER_QUALITY` smooths the scaled image. (You would get the same effect by setting `KEY_INTERPOLATION` to `VALUE_INTERPOLATION_BICUBIC`.)
- When antialiasing is turned off, selecting `VALUE_STROKE_NORMALIZE` changes the appearance of the ellipse and the placement of the diagonal line in the square.

Figure 11.26 shows a screen capture of the program.



**Figure 11.26** Testing the effect of rendering hints

---

**Listing 11.6** renderQuality/RenderQualityTestFrame.java

---

```
1 package renderQuality;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5
6 import javax.swing.*;
7
8 /**
9 * This frame contains buttons to set rendering hints and an image that is drawn with the selected
10 * hints.
11 */
12 public class RenderQualityTestFrame extends JFrame
13 {
14 private RenderQualityComponent canvas;
15 private JPanel buttonBox;
16 private RenderingHints hints;
17 private int r;
18 }
```

```
19 public RenderQualityTestFrame()
20 {
21 buttonBox = new JPanel();
22 buttonBox.setLayout(new GridBagLayout());
23 hints = new RenderingHints(null);
24
25 makeButtons("KEY_ANTIALIASING", "VALUE_ANTIALIAS_OFF", "VALUE_ANTIALIAS_ON");
26 makeButtons("KEY_TEXT_ANTIALIASING", "VALUE_TEXT_ANTIALIAS_OFF", "VALUE_TEXT_ANTIALIAS_ON");
27 makeButtons("KEY_FRACTIONALMETRICS", "VALUE_FRACTIONALMETRICS_OFF",
28 "VALUE_FRACTIONALMETRICS_ON");
29 makeButtons("KEY_RENDERING", "VALUE_RENDER_SPEED", "VALUE_RENDER_QUALITY");
30 makeButtons("KEY_STROKE_CONTROL", "VALUE_STROKE_PURE", "VALUE_STROKE_NORMALIZE");
31 canvas = new RenderQualityComponent();
32 canvas.setRenderingHints(hints);
33
34 add(canvas, BorderLayout.CENTER);
35 add(buttonBox, BorderLayout.NORTH);
36 pack();
37 }
38
39 /**
40 * Makes a set of buttons for a rendering hint key and values.
41 * @param key the key name
42 * @param value1 the name of the first value for the key
43 * @param value2 the name of the second value for the key
44 */
45 void makeButtons(String key, String value1, String value2)
46 {
47 try
48 {
49 final RenderingHints.Key k =
50 (RenderingHints.Key) RenderingHints.class.getField(key).get(null);
51 final Object v1 = RenderingHints.class.getField(value1).get(null);
52 final Object v2 = RenderingHints.class.getField(value2).get(null);
53 JLabel label = new JLabel(key);
54
55 buttonBox.add(label, new GBC(0, r).setAnchor(GBC.WEST));
56 ButtonGroup group = new ButtonGroup();
57 JRadioButton b1 = new JRadioButton(value1, true);
58
59 buttonBox.add(b1, new GBC(1, r).setAnchor(GBC.WEST));
60 group.add(b1);
61 b1.addActionListener(event ->
62 {
63 hints.put(k, v1);
64 canvas.setRenderingHints(hints);
65 });
66 JRadioButton b2 = new JRadioButton(value2, false);
```

(Continues)

**Listing 11.6 (Continued)**

```
67 buttonBox.add(b2, new GBC(2, r).setAnchor(GBC.WEST));
68 group.add(b2);
69 b2.addActionListener(event ->
70 {
71 hints.put(k, v2);
72 canvas.setRenderingHints(hints);
73 });
74 hints.put(k, v1);
75 r++;
76 }
77 catch (Exception e)
78 {
79 e.printStackTrace();
80 }
81 }
82 }
83 }
84 /**
85 * This component produces a drawing that shows the effect of rendering hints.
86 */
87 class RenderQualityComponent extends JPanel
88 {
89 private static final Dimension PREFERRED_SIZE = new Dimension(750, 150);
90 private RenderingHints hints = new RenderingHints(null);
91 private Image image;
92
93 public RenderQualityComponent()
94 {
95 image = new ImageIcon(getClass().getResource("face.gif")).getImage();
96 }
97
98 public void paintComponent(Graphics g)
99 {
100 Graphics2D g2 = (Graphics2D) g;
101 g2.setRenderingHints(hints);
102
103 g2.draw(new Ellipse2D.Double(10, 10, 60, 50));
104 g2.setFont(new Font("Serif", Font.ITALIC, 40));
105 g2.drawString("Hello", 75, 50);
106
107 g2.draw(new Rectangle2D.Double(200, 10, 40, 40));
108 g2.draw(new Line2D.Double(201, 11, 239, 49));
109
110 g2.drawImage(image, 250, 10, 100, 100, null);
111 }
112 }
```

```
113
114 /**
115 * Sets the hints and repaints.
116 * @param h the rendering hints
117 */
118 public void setRenderingHints(RenderingHints h)
119 {
120 hints = h;
121 repaint();
122 }
123
124 public Dimension getPreferredSize() { return PREFERRED_SIZE; }
125 }
```

**java.awt.Graphics2D 1.2**

- `void setRenderingHint(RenderingHints.Key key, Object value)`  
sets a rendering hint for this graphics context.
- `void setRenderingHints(Map m)`  
sets all rendering hints whose key/value pairs are stored in the map.

**java.awt.RenderingHints 1.2**

- `RenderingHints(Map<RenderingHints.Key, ?> m)`  
constructs a rendering hints map for storing rendering hints. If `m` is `null`, a default map implementation is provided.

## 11.10 Readers and Writers for Images

The `javax.imageio` package contains “out of the box” support for reading and writing several common file formats, as well as a framework that enables third parties to add readers and writers for other formats. The GIF, JPEG, PNG, BMP (Windows bitmap), and WBMP (wireless bitmap) file formats are supported.

The basics of the library are extremely straightforward. To load an image, use the static `read` method of the `ImageIO` class:

```
File f = . . .;
BufferedImage image = ImageIO.read(f);
```

The `ImageIO` class picks an appropriate reader, based on the file type. It may consult the file extension and the “magic number” at the beginning of the file for

that purpose. If no suitable reader can be found or the reader can't decode the file contents, the `read` method returns `null`.

Writing an image to a file is just as simple:

```
File f = . . .;
String format = . . .;
ImageIO.write(image, format, f);
```

Here the format string is a string identifying the image format, such as "JPEG" or "PNG". The `ImageIO` class picks an appropriate writer and saves the file.

### 11.10.1 Obtaining Readers and Writers for Image File Types

For more advanced image reading and writing operations that go beyond the static `read` and `write` methods of the `ImageIO` class, you first need to get the appropriate `ImageReader` and `ImageWriter` objects. The `ImageIO` class enumerates readers and writers that match one of the following:

- An image format (such as "JPEG")
- A file suffix (such as "jpg")
- A MIME type (such as "image/jpeg")

---

**NOTE:** MIME is the Multipurpose Internet Mail Extensions standard. The MIME standard defines common data formats such as "image/jpeg" and "application/pdf".

---

For example, you can obtain a reader that reads JPEG files as follows:

```
ImageReader reader = null;
Iterator<ImageReader> iter = ImageIO.getImageReadersByFormatName("JPEG");
if (iter.hasNext()) reader = iter.next();
```

The `getImageReadersBySuffix` and `getImageReadersByMIMEType` methods enumerate readers that match a file extension or MIME type.

It is possible that the `ImageIO` class can locate multiple readers that can all read a particular image type. In that case, you have to pick one of them, but it isn't clear how you can decide which one is the best. To find out more information about a reader, obtain its *service provider interface*:

```
ImageReaderSpi spi = reader.getOriginatingProvider();
```

Then you can get the vendor name and version number:

```
String vendor = spi.getVendor();
String version = spi.getVersion();
```

Perhaps that information can help you decide among the choices, or you might just present a list of readers to your program users and let them choose. For now, we assume that the first enumerated reader is adequate.

In the sample program in Listing 11.7, we want to find all file suffixes of all available readers so that we can use them in a file filter. Use the static `ImageIO.getReaderFileSuffixes` method for this purpose:

```
String[] extensions = ImageIO.getReaderFileSuffixes();
chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
```

For saving files, we have to work harder. We'd like to present the user with a menu of all supported image types. Unfortunately, the `getWriterFormatNames` of the `ImageIO` class returns a rather curious list with redundant names, such as

```
jpg, BMP, bmp, JPG, jpeg, wbmp, png, JPEG, PNG, WBMP, GIF, gif
```

That's not something one would want to present in a menu. What is needed is a list of "preferred" format names. We supply a helper method `getWriterFormats` for this purpose (see Listing 11.7). We look up the first writer associated with each format name. Then we ask it what its format names are, in the hope that it will list the most popular one first. Indeed, for the JPEG writer, this works fine—it lists "JPEG" before the other options. (The PNG writer, on the other hand, lists "png" in lower case before "PNG". We hope this behavior will be addressed at some point in the future. For now, we force all-lowercase names to upper case.) Once we pick a preferred name, we remove all alternate names from the original set. We keep going until all format names are handled.

## 11.10.2 Reading and Writing Files with Multiple Images

Some files—in particular, animated GIF files—contain multiple images. The `read` method of the `ImageIO` class reads a single image. To read multiple images, turn the input source (for example, an input stream or file) into an `ImageInputStream`.

```
InputStream in = . . .;
ImageInputStream imageIn = ImageIO.createImageInputStream(in);
```

Then, attach the image input stream to the reader:

```
reader.setInput(imageIn, true);
```

The second parameter indicates that the input is in “seek forward only” mode. Otherwise, random access is used, either by buffering stream input as it is read or by using random file access. Random access is required for certain operations. For example, to find out the number of images in a GIF file, you need to read the entire file. If you then want to fetch an image, the input must be read again.

This consideration is only important if you read from a stream, if the input contains multiple images, and if the image format doesn’t have the information that you request (such as the image count) in the header. If you read from a file, simply use

```
File f = . . .;
ImageInputStream imageIn = ImageIO.createImageInputStream(f);
reader.setInputStream(imageIn);
```

Once you have a reader, you can read the images in the input by calling

```
BufferedImage image = reader.read(index);
```

where `index` is the image index, starting with 0.

If the input is in the “seek forward only” mode, you keep reading images until the `read` method throws an `IndexOutOfBoundsException`. Otherwise, you can call the `getNumImages` method:

```
int n = reader.getNumImages(true);
```

Here, the parameter indicates that you allow a search of the input to determine the number of images. That method throws an `IllegalStateException` if the input is in the “seek forward only” mode. Alternatively, you can set the “allow search” parameter to `false`. Then the `getNumImages` method returns -1 if it can’t determine the number of images without a search. In that case, you’ll have to switch to Plan B and keep reading images until you get an `IndexOutOfBoundsException`.

Some files contain thumbnails—smaller versions of an image for preview purposes. You can get the number of thumbnails of an image with the call

```
int count = reader.getNumThumbnails(index);
```

Then you get a particular index as

```
BufferedImage thumbnail = reader.getThumbnail(index, thumbnailIndex);
```

Sometimes you may want to get the image size before actually getting the image—in particular, if the image is huge or comes from a slow network connection. Use the calls

```
int width = reader.getWidth(index);
int height = reader.getHeight(index);
```

to get the dimensions of an image with a given index.

To write a file with multiple images, you first need an `ImageWriter`. The `ImageIO` class can enumerate the writers capable of writing a particular image format:

```
String format = . . .;
ImageWriter writer = null;
Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatName(format);
if (iter.hasNext()) writer = iter.next();
```

Next, turn an output stream or file into an `ImageOutputStream` and attach it to the writer. For example,

```
File f = . . .;
ImageOutputStream imageOut = ImageIO.createImageOutputStream(f);
writer.setOutput(imageOut);
```

You must wrap each image into an `IIOImage` object. You can optionally supply a list of thumbnails and image metadata (such as compression algorithms and color information). In this example, we just use `null` for both; see the API documentation for additional information.

```
IIOImage iioImage = new IIOImage(images[i], null, null);
```

To write out the *first* image, use the `write` method:

```
writer.write(new IIOImage(images[0], null, null));
```

For subsequent images, use

```
if (writer.canInsertImage(i))
 writer.writeInsert(i, iioImage, null);
```

The third parameter can contain an `ImageWriteParam` object to set image writing details such as tiling and compression; use `null` for default values.

Not all file formats can handle multiple images. In that case, the `canInsertImage` method returns `false` for  $i > 0$ , and only a single image is saved.

The program in Listing 11.7 lets you load and save files in the formats for which the Java library supplies readers and writers. The program displays multiple images (see Figure 11.27), but not thumbnails.

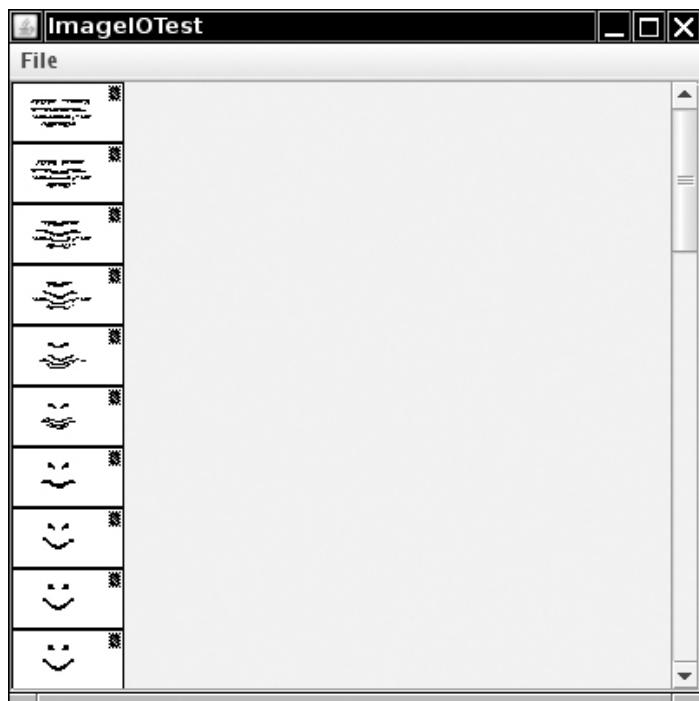


Figure 11.27 An animated GIF image

---

**Listing 11.7** `imageIO/ImageIOFrame.java`

---

```
1 package imageIO;
2
3 import java.awt.image.*;
4 import java.io.*;
5 import java.util.*;
6
7 import javax.imageio.*;
8 import javax.imageio.stream.*;
9 import javax.swing.*;
10 import javax.swing.filechooser.*;
11
12 /**
13 * This frame displays the loaded images. The menu has items for loading and saving files.
14 */
15 public class ImageIOFrame extends JFrame
16 {
17 private static final int DEFAULT_WIDTH = 400;
18 private static final int DEFAULT_HEIGHT = 400;
```

```
19 private static Set<String> writerFormats = getWriterFormats();
20
21 private BufferedImage[] images;
22
23 public ImageIOFrame()
24 {
25 setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
26
27 JMenu fileMenu = new JMenu("File");
28 JMenuItem openItem = new JMenuItem("Open");
29 openItem.addActionListener(event -> openFile());
30 fileMenu.add(openItem);
31
32 JMenu saveMenu = new JMenu("Save");
33 fileMenu.add(saveMenu);
34 Iterator<String> iter = writerFormats.iterator();
35 while (iter.hasNext())
36 {
37 final String formatName = iter.next();
38 JMenuItem formatItem = new JMenuItem(formatName);
39 saveMenu.add(formatItem);
40 formatItem.addActionListener(event -> saveFile(formatName));
41 }
42
43 JMenuItem exitItem = new JMenuItem("Exit");
44 exitItem.addActionListener(event -> System.exit(0));
45 fileMenu.add(exitItem);
46
47 JMenuBar menuBar = new JMenuBar();
48 menuBar.add(fileMenu);
49 setJMenuBar(menuBar);
50 }
51
52
53 /**
54 * Open a file and load the images.
55 */
56 public void openFile()
57 {
58 JFileChooser chooser = new JFileChooser();
59 chooser.setCurrentDirectory(new File("."));
60 String[] extensions = ImageIO.getReaderFileSuffixes();
61 chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
62 int r = chooser.showOpenDialog(this);
63 if (r != JFileChooser.APPROVE_OPTION) return;
64 File f = chooser.getSelectedFile();
65 Box box = Box.createVerticalBox();
66 try
67 {
```

(Continues)

**Listing 11.7 (Continued)**

```
68 String name = f.getName();
69 String suffix = name.substring(name.lastIndexOf('.') + 1);
70 Iterator<ImageReader> iter = ImageIO.getImageReadersBySuffix(suffix);
71 ImageReader reader = iter.next();
72 ImageInputStream imageIn = ImageIO.createImageInputStream(f);
73 reader.setInput(imageIn);
74 int count = reader.getNumImages(true);
75 images = new BufferedImage[count];
76 for (int i = 0; i < count; i++)
77 {
78 images[i] = reader.read(i);
79 box.add(new JLabel(new ImageIcon(images[i])));
80 }
81 }
82 catch (IOException e)
83 {
84 JOptionPane.showMessageDialog(this, e);
85 }
86 setContentPane(new JScrollPane(box));
87 validate();
88 }
89 /**
90 * Save the current image in a file.
91 * @param formatName the file format
92 */
93 public void saveFile(final String formatName)
94 {
95 if (images == null) return;
96 Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatName(formatName);
97 ImageWriter writer = iter.next();
98 JFileChooser chooser = new JFileChooser();
99 chooser.setCurrentDirectory(new File("."));
100 String[] extensions = writer.getOriginatingProvider().getFileSuffixes();
101 chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
102
103 int r = chooser.showSaveDialog(this);
104 if (r != JFileChooser.APPROVE_OPTION) return;
105 File f = chooser.getSelectedFile();
106 try
107 {
108 ImageOutputStream imageOut = ImageIO.createImageOutputStream(f);
109 writer.setOutput(imageOut);
110
111 writer.write(new IIOImage(images[0], null, null));
112 for (int i = 1; i < images.length; i++)
113 {
114
```

```
115 IIOImage iioImage = new IIOImage(images[i], null, null);
116 if (writer.canInsertImage(i)) writer.writeInsert(i, iioImage, null);
117 }
118 }
119 catch (IOException e)
120 {
121 JOptionPane.showMessageDialog(this, e);
122 }
123 }
124
125 /**
126 * Gets a set of "preferred" format names of all image writers. The preferred format name is
127 * the first format name that a writer specifies.
128 * @return the format name set
129 */
130 public static Set<String> getWriterFormats()
131 {
132 Set<String> writerFormats = new TreeSet<>();
133 Set<String> formatNames = new TreeSet<>(
134 Arrays.asList(ImageIO.getWriterFormatNames()));
135 while (formatNames.size() > 0)
136 {
137 String name = formatNames.iterator().next();
138 Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatName(name);
139 ImageWriter writer = iter.next();
140 String[] names = writer.getOriginatingProvider().getFormatNames();
141 String format = names[0];
142 if (format.equals(format.toLowerCase())) format = format.toUpperCase();
143 writerFormats.add(format);
144 formatNames.removeAll(Arrays.asList(names));
145 }
146 return writerFormats;
147 }
148 }
```

**javax.imageio.ImageIO 1.4**

- static BufferedImage read(File input)
- static BufferedImage read(InputStream input)
- static BufferedImage read(URL input)  
reads an image from input.
- static boolean write(RenderedImage image, String formatName, File output)
- static boolean write(RenderedImage image, String formatName, OutputStream output)  
writes an image in the given format to output. Returns false if no appropriate writer was found.

*(Continues)*

**`javax.imageio.ImageIO 1.4 (Continued)`**

- `static Iterator<ImageReader> getImageReadersByFormatName(String formatName)`
- `static Iterator<ImageReader> getImageReadersBySuffix(String fileSuffix)`
- `static Iterator<ImageReader> getImageReadersByMIMEType(String mimeType)`
- `static Iterator<ImageWriter> getImageWritersByFormatName(String formatName)`
- `static Iterator<ImageWriter> getImageWritersBySuffix(String fileSuffix)`
- `static Iterator<ImageWriter> getImageWritersByMIMEType(String mimeType)`

gets all readers and writers that are able to handle the given format (e.g., "JPEG"), file suffix (e.g., "jpg"), or MIME type (e.g., "image/jpeg").

- `static String[] getReaderFormatNames()`
- `static String[] getReaderMIMETypes()`
- `static String[] getWriterFormatNames()`
- `static String[] getWriterMIMETypes()`
- `static String[] getReaderFileSuffixes() 6`
- `static String[] getWriterFileSuffixes() 6`

gets all format names, MIME type names, and file suffixes supported by readers and writers.

- `ImageInputStream createImageInputStream(Object input)`
- `ImageOutputStream createImageOutputStream(Object output)`

creates an image input or image output stream from the given object. The object can be a file, a stream, a `RandomAccessFile`, or another object for which a service provider exists. Returns `null` if no registered service provider can handle the object.

**`javax.imageio.ImageReader 1.4`**

- `void setInput(Object input)`
- `void setInput(Object input, boolean seekForwardOnly)`

sets the input source of the reader.

*Parameters:*      `input`      An `ImageInputStream` object or another object that this reader can accept.

`seekForwardOnly`      `true` if the reader should read forward only. By default, the reader uses random access and, if necessary, buffers image data.

- `BufferedImage read(int index)`

reads the image with the given image index (starting at 0). Throws an `IndexOutOfBoundsException` if no such image is available.

*(Continues)*

**javax.imageio.ImageReader 1.4 (Continued)**

- `int getNumImages(boolean allowSearch)`  
gets the number of images in this reader. If `allowSearch` is `false` and the number of images cannot be determined without reading forward, then `-1` is returned. If `allowSearch` is `true` and the reader is in the “seek forward only” mode, then an `IllegalStateException` is thrown.
- `int getNumThumbnails(int index)`  
gets the number of thumbnails of the image with the given `index`.
- `BufferedImage readThumbnail(int index, int thumbnailIndex)`  
gets the thumbnail with index `thumbnailIndex` of the image with the given `index`.
- `int getWidth(int index)`
- `int getHeight(int index)`  
gets the image width and height. Throws an `IndexOutOfBoundsException` if no such image is available.
- `ImageReaderSpi getOriginatingProvider()`  
gets the service provider that constructed this reader.

**javax.imageio.spi.IIOServiceProvider 1.4**

- `String getVendorName()`
- `String getVersion()`  
gets the vendor name and version of this service provider.

**javax.imageio.spi.ImageReaderWriterSpi 1.4**

- `String[] getFormatNames()`
- `String[] getFileSuffixes()`
- `String[] getMIMETypes()`  
gets the format names, file suffixes, and MIME types supported by the readers or writers that this service provider creates.

**javax.imageio.ImageWriter 1.4**

- void setOutput(Object output)

sets the output target of this writer.

*Parameters:*      output      An `ImageOutputStream` object or another object that this writer can accept

- void write(IIOImage image)

- void write(RenderedImage image)

writes a single image to the output

- void writeInsert(int index, IIOImage image, ImageWriteParam param)

writes an image into a multi-image file.

- boolean canInsertImage(int index)

returns true if it is possible to insert an image at the given index.

- ImageWriterSpi getOriginatingProvider()

gets the service provider that constructed this writer.

**javax.imageio.IIOImage 1.4**

- IIOImage(RenderedImage image, List thumbnails, IIOMetadata metadata)

constructs an `IIOImage` from an image, optional thumbnails, and optional metadata.

## 11.11 Image Manipulation

Suppose you have an image and you would like to improve its appearance. You then need to access the individual pixels of the image and replace them with other pixels. Or perhaps you want to compute the pixels of an image from scratch—for example, to show the result of physical measurements or a mathematical computation. The `BufferedImage` class gives you control over the pixels in an image, and the classes that implement the `BufferedImageOp` interface let you transform images.

---

**NOTE:** JDK 1.0 had a completely different, and far more complex, imaging framework that was optimized for *incremental rendering* of images downloaded from the Web, a scan line at a time. However, it was difficult to manipulate those images. We do not discuss that framework in this book.

---

### 11.11.1 Constructing Raster Images

Most of the images that you manipulate are simply read in from an image file—they were either produced by a device such as a digital camera or scanner, or constructed by a drawing program. In this section, we'll show you a different technique for constructing an image—namely, building it up a pixel at a time.

To create an image, construct a `BufferedImage` object in the usual way.

```
image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
```

Now, call the `getRaster` method to obtain an object of type `WritableRaster`. You will use this object to access and modify the pixels of the image.

```
WritableRaster raster = image.getRaster();
```

The `setPixel` method lets you set an individual pixel. The complexity here is that you can't simply set the pixel to a `Color` value. You must know how the buffered image specifies color values. That depends on the *type* of the image. If your image has a type of `TYPE_INT_ARGB`, then each pixel is described by four values—red, green, blue, and alpha, each between 0 and 255. You have to supply them in an array of four integers:

```
int[] black = { 0, 0, 0, 255 };
raster.setPixel(i, j, black);
```

In the lingo of the Java 2D API, these values are called the *sample values* of the pixel.



**CAUTION:** There are also `setPixel` methods that take array parameters of types `float[]` and `double[]`. However, the values that you need to place into these arrays are *not* normalized color values between 0.0 and 1.0.

```
float[] red = { 1.0F, 0.0F, 0.0F, 1.0F };
raster.setPixel(i, j, red); // ERROR
```

You need to supply values between 0 and 255, no matter what the type of the array is.

You can supply batches of pixels with the `setPixels` method. Specify the starting pixel position and the width and height of the rectangle that you want to set. Then, supply an array that contains the sample values for all pixels. For example, if your buffered image has a type of `TYPE_INT_ARGB`, supply the red, green, blue, and alpha values of the first pixel, then the red, green, blue, and alpha values for the second pixel, and so on.

```
int[] pixels = new int[4 * width * height];
pixels[0] = . . . // red value for first pixel
pixels[1] = . . . // green value for first pixel
pixels[2] = . . . // blue value for first pixel
pixels[3] = . . . // alpha value for first pixel
. .
raster.setPixels(x, y, width, height, pixels);
```

Conversely, to read a pixel, use the `getPixel` method. Supply an array of four integers to hold the sample values.

```
int[] sample = new int[4];
raster.getPixel(x, y, sample);
Color c = new Color(sample[0], sample[1], sample[2], sample[3]);
```

You can read multiple pixels with the `getPixels` method.

```
raster.getPixels(x, y, width, height, samples);
```

If you use an image type other than `TYPE_INT_ARGB` and you know how that type represents pixel values, you can still use the `getPixel`/`setPixel` methods. However, you have to know the encoding of the sample values in the particular image type.

If you need to manipulate an image with an arbitrary, unknown image type, then you have to work a bit harder. Every image type has a *color model* that can translate between sample value arrays and the standard RGB color model.

---

**NOTE:** The RGB color model isn't as standard as you might think. The exact look of a color value depends on the characteristics of the imaging device. Digital cameras, scanners, monitors, and LCD displays all have their own idiosyncrasies. As a result, the same RGB value can look quite different on different devices. The International Color Consortium ([www.color.org](http://www.color.org)) recommends that all color data be accompanied by an *ICC profile* that specifies how the colors map to a standard form such as the 1931 CIE XYZ color specification. That specification was designed by the Commission Internationale de l'Eclairage, or CIE ([www.cie.co.at](http://www.cie.co.at)), the international organization in charge of providing technical guidance in all matters of illumination and color. The specification is a standard method for representing any color that the human eye can perceive as a triplet of coordinates called X, Y, Z. (See, for example, *Computer Graphics: Principles and Practice, Second Edition in C*, by James D. Foley, Andries van Dam, Steven K. Feiner, et al., Chapter 13, for more information on the 1931 CIE XYZ specification.)

---

ICC profiles are complex, however. A simpler proposed standard, called sRGB ([www.w3.org/Graphics/Color/sRGB.html](http://www.w3.org/Graphics/Color/sRGB.html)), specifies an exact mapping between RGB values and the 1931 CIE XYZ values that was designed to work well with typical color monitors. The Java 2D API uses that mapping when converting between RGB and other color spaces.

---

The `getColorModel` method returns the color model:

```
ColorModel model = image.getColorModel();
```

To find the color value of a pixel, call the `getDataElements` method of the `Raster` class. That call returns an `Object` that contains a color-model-specific description of the color value.

```
Object data = raster.getDataElements(x, y, null);
```

---

**NOTE:** The object that is returned by the `getDataElements` method is actually an array of sample values. You don't need to know this to process the object, but it explains why the method is called `getDataElements`.

---

The color model can translate the object to standard ARGB values. The `getRGB` method returns an `int` value that has the alpha, red, green, and blue values packed in four blocks of eight bits each. You can construct a `Color` value out of that integer with the `Color(int argb, boolean hasAlpha)` constructor:

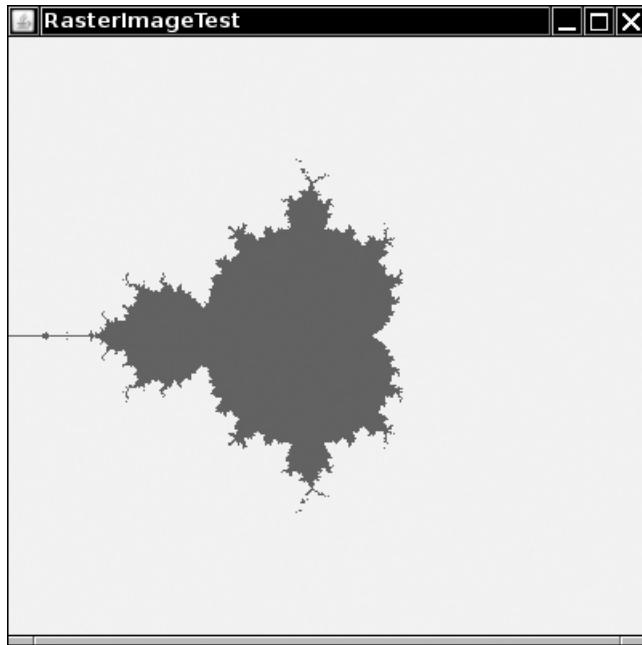
```
int argb = model.getRGB(data);
Color color = new Color(argb, true);
```

To set a pixel to a particular color, reverse these steps. The `getRGB` method of the `Color` class yields an `int` value with the alpha, red, green, and blue values. Supply that value to the `getDataElements` method of the `ColorModel` class. The return value is an `Object` that contains the color-model-specific description of the color value. Pass the object to the `setDataElements` method of the `WritableRaster` class.

```
int argb = color.getRGB();
Object data = model.getDataElements(argb, null);
raster.setDataElements(x, y, data);
```

To illustrate how to use these methods to build an image from individual pixels, we bow to tradition and draw a Mandelbrot set, as shown in Figure 11.28.

The idea of the Mandelbrot set is that each point of the plane is associated with a sequence of numbers. If that sequence stays bounded, you color the point. If it “escapes to infinity,” you leave it transparent.



**Figure 11.28** A Mandelbrot set

Here is how you can construct the simplest Mandelbrot set. For each point  $(a, b)$ , look at sequences that start with  $(x, y) = (0, 0)$  and iterate:

$$x_{\text{new}} = x^2 - y^2 + a$$

$$y_{\text{new}} = 2 \cdot x \cdot y + b$$

It turns out that if  $x$  or  $y$  ever gets larger than 2, then the sequence escapes to infinity. Only the pixels that correspond to points  $(a, b)$  leading to a bounded sequence are colored. (The formulas for the number sequences come ultimately from the mathematics of complex numbers; we'll just take them for granted. For more on the mathematics of fractals, see, for example, <http://classes.yale.edu/fractals.>)

Listing 11.8 shows the code. In this program, we demonstrate how to use the `ColorModel` class for translating `Color` values into pixel data. That process is independent of the image type. Just for fun, change the color type of the buffered image to `TYPE_BYTE_GRAY`. You don't need to change any other code—the color model of the image automatically takes care of the conversion from colors to sample values.

**Listing 11.8** rasterImage/RasterImageFrame.java

```
1 package rasterImage;
2
3 import java.awt.*;
4 import java.awt.image.*;
5 import javax.swing.*;
6
7 /**
8 * This frame shows an image with a Mandelbrot set.
9 */
10 public class RasterImageFrame extends JFrame
11 {
12 private static final double XMIN = -2;
13 private static final double XMAX = 2;
14 private static final double YMIN = -2;
15 private static final double YMAX = 2;
16 private static final int MAX_ITERATIONS = 16;
17 private static final int IMAGE_WIDTH = 400;
18 private static final int IMAGE_HEIGHT = 400;
19
20 public RasterImageFrame()
21 {
22 BufferedImage image = makeMandelbrot(IMAGE_WIDTH, IMAGE_HEIGHT);
23 add(new JLabel(new ImageIcon(image)));
24 pack();
25 }
26
27 /**
28 * Makes the Mandelbrot image.
29 * @param width the width
30 * @param height the height
31 * @return the image
32 */
33 public BufferedImage makeMandelbrot(int width, int height)
34 {
35 BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
36 WritableRaster raster = image.getRaster();
37 ColorModel model = image.getColorModel();
38
39 Color fractalColor = Color.red;
40 int argb = fractalColor.getRGB();
41 Object colorData = model.getDataElements(argb, null);
42
43 for (int i = 0; i < width; i++)
44 for (int j = 0; j < height; j++)
45 {
```

*(Continues)*

**Listing 11.8 (Continued)**

```
46 double a = XMIN + i * (XMAX - XMIN) / width;
47 double b = YMIN + j * (YMAX - YMIN) / height;
48 if (!escapesToInfinity(a, b)) raster.setDataElements(i, j, colorData);
49 }
50 return image;
51 }
52
53 private boolean escapesToInfinity(double a, double b)
54 {
55 double x = 0.0;
56 double y = 0.0;
57 int iterations = 0;
58 while (x <= 2 && y <= 2 && iterations < MAX_ITERATIONS)
59 {
60 double xnew = x * x - y * y + a;
61 double ynew = 2 * x * y + b;
62 x = xnew;
63 y = ynew;
64 iterations++;
65 }
66 return x > 2 || y > 2;
67 }
68 }
```

---

**java.awt.image.BufferedImage 1.2**

- `BufferedImage(int width, int height, int imageType)`

constructs a buffered image object.

*Parameters:*      `width, height`

The image dimensions

`imageType`

The image type. The most common types are `TYPE_INT_RGB`, `TYPE_INT_ARGB`, `TYPE_BYTE_GRAY`, and `TYPE_BYTE_INDEXED`.

- `ColorModel getColorModel()`

returns the color model of this buffered image.

- `WritableRaster getRaster()`

gets the raster for accessing and modifying pixels of this buffered image.

**java.awt.image.Raster 1.2**

- `Object getDataElements(int x, int y, Object data)`

returns the sample data for a raster point, in an array whose element type and length depend on the color model. If `data` is not `null`, it is assumed to be an array that is appropriate for holding sample data, and it is filled. If `data` is `null`, a new array is allocated. Its element type and length depend on the color model.

- `int[] getPixel(int x, int y, int[] sampleValues)`
- `float[] getPixel(int x, int y, float[] sampleValues)`
- `double[] getPixel(int x, int y, double[] sampleValues)`
- `int[] getPixels(int x, int y, int width, int height, int[] sampleValues)`
- `float[] getPixels(int x, int y, int width, int height, float[] sampleValues)`
- `double[] getPixels(int x, int y, int width, int height, double[] sampleValues)`

returns the sample values for a raster point, or a rectangle of raster points, in an array whose length depends on the color model. If `sampleValues` is not `null`, it is assumed to be sufficiently long for holding the sample values, and it is filled. If `sampleValues` is `null`, a new array is allocated. These methods are only useful if you know the meaning of the sample values for a color model.

**java.awt.image.WritableRaster 1.2**

- `void setDataElements(int x, int y, Object data)`

sets the sample data for a raster point. `data` is an array filled with the sample data for a pixel. Its element type and length depend on the color model.

- `void setPixel(int x, int y, int[] sampleValues)`
- `void setPixel(int x, int y, float[] sampleValues)`
- `void setPixel(int x, int y, double[] sampleValues)`
- `void setPixels(int x, int y, int width, int height, int[] sampleValues)`
- `void setPixels(int x, int y, int width, int height, float[] sampleValues)`
- `void setPixels(int x, int y, int width, int height, double[] sampleValues)`

sets the sample values for a raster point or a rectangle of raster points. These methods are only useful if you know the encoding of the sample values for a color model.

**java.awt.image.ColorModel 1.2**

- `int getRGB(Object data)`  
returns the ARGB value that corresponds to the sample data passed in the data array. Its element type and length depend on the color model.
- `Object getDataElements(int argb, Object data);`  
returns the sample data for a color value. If `data` is not `null`, it is assumed to be an array that is appropriate for holding sample data, and it is filled. If `data` is `null`, a new array is allocated. `data` is an array filled with the sample data for a pixel. Its element type and length depend on the color model.

**java.awt.Color 1.0**

- `Color(int argb, boolean hasAlpha) 1.2`  
creates a color with the specified combined ARGB value if `hasAlpha` is `true`, or the specified RGB value if `hasAlpha` is `false`.
- `int getRGB()`  
returns the ARGB color value corresponding to this color.

### 11.11.2 Filtering Images

In the preceding section, you saw how to build up an image from scratch. However, often you want to access image data for a different reason: You already have an image and you want to improve it in some way.

Of course, you can use the `getPixel/getDataElements` methods that you saw in the preceding section to read the image data, manipulate them, and write them back. But fortunately, the Java 2D API already supplies a number of *filters* that carry out common image processing operations for you.

The image manipulations all implement the `BufferedImageOp` interface. After you construct the operation, you simply call the `filter` method to transform an image into another.

```
BufferedImageOp op = . . .;
BufferedImage filteredImage =
 new BufferedImage(image.getWidth(), image.getHeight(), image.getType());
op.filter(image, filteredImage);
```

Some operations can transform an image in place (`op.filter(image, image)`), but most can't.

Five classes implement the `BufferedImageOp` interface:

```
AffineTransformOp
RescaleOp
LookupOp
ColorConvertOp
ConvolveOp
```

The `AffineTransformOp` carries out an affine transformation on the pixels. For example, here is how you can rotate an image about its center:

```
AffineTransform transform = AffineTransform.getRotateInstance(Math.toRadians(angle),
 image.getWidth() / 2, image.getHeight() / 2);
AffineTransformOp op = new AffineTransformOp(transform, interpolation);
op.filter(image, filteredImage);
```

The `AffineTransformOp` constructor requires an affine transform and an *interpolation* strategy. Interpolation is necessary to determine pixels in the target image if the source pixels are transformed somewhere between target pixels. For example, if you rotate source pixels, then they will generally not fall exactly onto target pixels. There are two interpolation strategies: `AffineTransformOp.TYPE_BILINEAR` and `AffineTransformOp.TYPE_NEAREST_NEIGHBOR`. Bilinear interpolation takes a bit longer but looks better.

The program in Listing 11.9 lets you rotate an image by 5 degrees (see Figure 11.29).

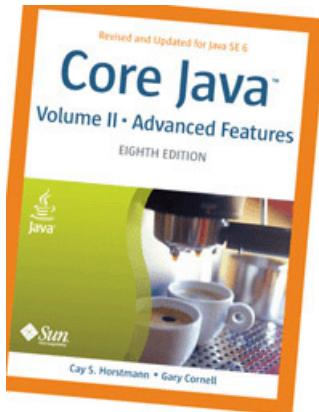


Figure 11.29 A rotated image

The `RescaleOp` carries out a rescaling operation

$$x_{\text{new}} = a \cdot x + b$$

for each of the color components in the image. (Alpha components are not affected.) The effect of rescaling with  $a > 1$  is to brighten the image. Construct the `RescaleOp` by specifying the scaling parameters and optional rendering hints. In Listing 11.9, we use:

```
float a = 1.1f;
float b = 20.0f;
RescaleOp op = new RescaleOp(a, b, null);
```

You can also supply separate scaling values for each color component—see the API notes.

The `LookupOp` operation lets you specify an arbitrary mapping of sample values. Supply a table that specifies how each value should be mapped. In the example program, we compute the *negative* of all colors, changing the color  $c$  to  $255 - c$ .

The `LookupOp` constructor requires an object of type `LookupTable` and a map of optional hints. The `LookupTable` class is abstract, with two concrete subclasses: `ByteLookupTable` and `ShortLookupTable`. Since RGB color values are bytes, a `ByteLookupTable` should suffice. However, because of the bug described in [http://bugs.sun.com/bugdatabase/view\\_bug.do?bug\\_id=6183251](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6183251), we will use a `ShortLookupTable` instead. Here is how we construct the `LookupOp` for the example program:

```
short negative[] = new short[256];
for (int i = 0; i < 256; i++) negative[i] = (short) (255 - i);
ShortLookupTable table = new ShortLookupTable(0, negative);
LookupOp op = new LookupOp(table, null);
```

The lookup is applied to each color component separately, but not to the alpha component. You can also supply different lookup tables for each color component—see the API notes.

---

**NOTE:** You cannot apply a `LookupOp` to an image with an indexed color model.  
(In those images, each sample value is an offset into a color palette.)

---

The `ColorConvertOp` is useful for color space conversions. We do not discuss it here.

The most powerful of the transformations is the `ConvolveOp`, which carries out a mathematical *convolution*. We won't get too deeply into the mathematical details, but the basic idea is simple. Consider, for example, the *blur filter* (see Figure 11.30).

The blurring is achieved by replacing each pixel with the *average* value from the pixel and its eight neighbors. Intuitively, it makes sense why this operation would blur out the picture. Mathematically, the averaging can be expressed as a convolution operation with the following *kernel*:



Figure 11.30 Blurring an image

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

The kernel of a convolution is a matrix that tells what weights should be applied to the neighboring values. The kernel above produces a blurred image. A different kernel carries out *edge detection*, locating the areas of color changes:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Edge detection is an important technique for analyzing photographic images (see Figure 11.31).

To construct a convolution operation, first set up an array of the values for the kernel and construct a `Kernel` object. Then, construct a `ConvolveOp` object from the kernel and use it for filtering.

```
float[] elements =
{
 0.0f, -1.0f, 0.0f,
 -1.0f, 4.0f, -1.0f,
 0.0f, -1.0f, 0.0f
};
Kernel kernel = new Kernel(3, 3, elements);
ConvolveOp op = new ConvolveOp(kernel);
op.filter(image, filteredImage);
```



**Figure 11.31** Edge detection and inversion

The program in Listing 11.9 allows a user to load in a GIF or JPEG image and carry out the image manipulations that we discussed. Thanks to the power of the operations provided by Java 2D API, the program is very simple.

---

**Listing 11.9** `imageProcessing/ImageProcessingFrame.java`

---

```
1 package imageProcessing;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.image.*;
6 import java.io.*;
7
8 import javax.imageio.*;
9 import javax.swing.*;
10 import javax.swing.filechooser.*;
11
12 /**
13 * This frame has a menu to load an image and to specify various transformations, and a component
14 * to show the resulting image.
15 */
16 public class ImageProcessingFrame extends JFrame
17 {
18 private static final int DEFAULT_WIDTH = 400;
19 private static final int DEFAULT_HEIGHT = 400;
20
21 private BufferedImage image;
22
23 public ImageProcessingFrame()
24 {
```

```
25 setTitle("ImageProcessingTest");
26 setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
27
28 add(new JComponent()
29 {
30 public void paintComponent(Graphics g)
31 {
32 if (image != null) g.drawImage(image, 0, 0, null);
33 }
34 });
35
36 JMenu fileMenu = new JMenu("File");
37 JMenuItem openItem = new JMenuItem("Open");
38 openItem.addActionListener(event -> openFileDialog());
39 fileMenu.add(openItem);
40
41 JMenuItem exitItem = new JMenuItem("Exit");
42 exitItem.addActionListener(event -> System.exit(0));
43 fileMenu.add(exitItem);
44
45 JMenu editMenu = new JMenu("Edit");
46 JMenuItem blurItem = new JMenuItem("Blur");
47 blurItem.addActionListener(event ->
48 {
49 float weight = 1.0f / 9.0f;
50 float[] elements = new float[9];
51 for (int i = 0; i < 9; i++)
52 elements[i] = weight;
53 convolve(elements);
54 });
55 editMenu.add(blurItem);
56
57 JMenuItem sharpenItem = new JMenuItem("Sharpen");
58 sharpenItem.addActionListener(event ->
59 {
60 float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 5.f, -1.0f, 0.0f, -1.0f, 0.0f };
61 convolve(elements);
62 });
63 editMenu.add(sharpenItem);
64
65 JMenuItem brightenItem = new JMenuItem("Brighten");
66 brightenItem.addActionListener(event ->
67 {
68 float a = 1.1f;
69 float b = 20.0f;
70 RescaleOp op = new RescaleOp(a, b, null);
71 filter(op);
72 });
73 editMenu.add(brightenItem);
```

(Continues)

**Listing 11.9 (Continued)**

```
74
75 JMenuItem edgeDetectItem = new JMenuItem("Edge detect");
76 edgeDetectItem.addActionListener(event ->
77 {
78 float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 4.f, -1.0f, 0.0f, -1.0f, 0.0f };
79 convolve(elements);
80 });
81 editMenu.add(edgeDetectItem);
82
83 JMenuItem negativeItem = new JMenuItem("Negative");
84 negativeItem.addActionListener(event ->
85 {
86 short[] negative = new short[256 * 1];
87 for (int i = 0; i < 256; i++)
88 negative[i] = (short) (255 - i);
89 ShortLookupTable table = new ShortLookupTable(0, negative);
90 LookupOp op = new LookupOp(table, null);
91 filter(op);
92 });
93 editMenu.add(negativeItem);
94
95 JMenuItem rotateItem = new JMenuItem("Rotate");
96 rotateItem.addActionListener(event ->
97 {
98 if (image == null) return;
99 AffineTransform transform = AffineTransform.getRotateInstance(Math.toRadians(5),
100 image.getWidth() / 2, image.getHeight() / 2);
101 AffineTransformOp op = new AffineTransformOp(transform,
102 AffineTransformOp.TYPE_BICUBIC);
103 filter(op);
104 });
105 editMenu.add(rotateItem);
106
107 JMenuBar menuBar = new JMenuBar();
108 menuBar.add(fileMenu);
109 menuBar.add(editMenu);
110 setJMenuBar(menuBar);
111 }
112
113 /**
114 * Open a file and load the image.
115 */
```

```
116 public void openFile()
117 {
118 JFileChooser chooser = new JFileChooser(".");
119 chooser.setCurrentDirectory(new File(getClass().getPackage().getName()));
120 String[] extensions = ImageIO.getReaderFileSuffixes();
121 chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
122 int r = chooser.showOpenDialog(this);
123 if (r != JFileChooser.APPROVE_OPTION) return;
124
125 try
126 {
127 Image img = ImageIO.read(chooser.getSelectedFile());
128 image = new BufferedImage(img.getWidth(null), img.getHeight(null),
129 BufferedImage.TYPE_INT_RGB);
130 image.getGraphics().drawImage(img, 0, 0, null);
131 }
132 catch (IOException e)
133 {
134 JOptionPane.showMessageDialog(this, e);
135 }
136 repaint();
137 }
138 /**
139 * Apply a filter and repaint.
140 * @param op the image operation to apply
141 */
142 private void filter(BufferedImageOp op)
143 {
144 if (image == null) return;
145 image = op.filter(image, null);
146 repaint();
147 }
148 /**
149 * Apply a convolution and repaint.
150 * @param elements the convolution kernel (an array of 9 matrix elements)
151 */
152 private void convolve(float[] elements)
153 {
154 Kernel kernel = new Kernel(3, 3, elements);
155 ConvolveOp op = new ConvolveOp(kernel);
156 filter(op);
157 }
158 }
159 }
160 }
```

**java.awt.image(BufferedImageOp 1.2**

- `BufferedImage filter(BufferedImage source, BufferedImage dest)`

applies the image operation to the source image and stores the result in the destination image. If `dest` is `null`, a new destination image is created. The destination image is returned.

**java.awt.image(AffineTransformOp 1.2**

- `AffineTransformOp(AffineTransform t, int interpolationType)`

constructs an affine transform operator. The interpolation type is one of `TYPE_BILINEAR`, `TYPE_BICUBIC`, or `TYPE_NEAREST_NEIGHBOR`.

**java.awt.image.RescaleOp 1.2**

- `RescaleOp(float a, float b, RenderingHints hints)`
- `RescaleOp(float[] as, float[] bs, RenderingHints hints)`

constructs a rescale operator that carries out the scaling operation  $x_{\text{new}} = a \cdot x + b$ . When using the first constructor, all color components (but not the alpha component) are scaled with the same coefficients. When using the second constructor, you supply either the values for each color component, in which case the alpha component is unaffected, or the values for both alpha and color components.

**java.awt.image.LookupOp 1.2**

- `LookupOp(LookupTable table, RenderingHints hints)`

constructs a lookup operator for the given lookup table.

**java.awt.image.ByteLookupTable 1.2**

- `ByteLookupTable(int offset, byte[] data)`
- `ByteLookupTable(int offset, byte[][] data)`

constructs a lookup table for converting byte values. The offset is subtracted from the input before the lookup. The values in the first constructor are applied to all color components but not the alpha component. When using the second constructor, you supply either the values for each color component, in which case the alpha component is unaffected, or the values for both alpha and color components.

**java.awt.image.ShortLookupTable 1.2**

- `ShortLookupTable(int offset, short[] data)`
- `ShortLookupTable(int offset, short[][] data)`

constructs a lookup table for converting short values. The offset is subtracted from the input before the lookup. The values in the first constructor are applied to all color components but not the alpha component. When using the second constructor, you supply either the values for each color component, in which case the alpha component is unaffected, or the values for both alpha and color components.

**java.awt.image.ConvolveOp 1.2**

- `ConvolveOp(Kernel kernel)`
- `ConvolveOp(Kernel kernel, int edgeCondition, RenderingHints hints)`

constructs a convolution operator. The edge condition specified is one of `EDGE_NO_OP` and `EDGE_ZERO_FILL`. Edge values need to be treated specially because they don't have sufficient neighboring values to compute the convolution. The default is `EDGE_ZERO_FILL`.

**java.awt.image.Kernel 1.2**

- `Kernel(int width, int height, float[] matrixElements)`

constructs a kernel for the given matrix.

## 11.12 Printing

The original JDK had no support for printing at all. It was not possible to print from applets, and you had to get a third-party library if you wanted to print from an application. JDK 1.1 introduced very lightweight printing support, just enough to produce simple printouts, as long as you were not too particular about the print quality. The 1.1 printing model was designed to allow browser vendors to print the surface of an applet as it appears on a web page (which, however, the browser vendors have not embraced).

Java SE 1.2 introduced the beginnings of a robust printing model that is fully integrated with 2D graphics. Java SE 1.4 added important enhancements, such as discovery of printer features and streaming print jobs for server-side print management.

In this section, we will show you how you can easily print a drawing on a single sheet of paper, how you can manage a multipage printout, and how you can benefit from the elegance of the Java 2D imaging model and easily generate a print preview dialog box.

### 11.12.1 Graphics Printing

In this section, we will tackle what is probably the most common printing situation: printing a 2D graphic. Of course, the graphic can contain text in various fonts or even consist entirely of text.

To generate a printout, you have to take care of these two tasks:

- Supply an object that implements the `Printable` interface
- Start a print job

The `Printable` interface has a single method:

```
int print(Graphics g, PageFormat format, int page)
```

That method is called whenever the print engine needs to have a page formatted for printing. Your code draws the text and the images to be printed onto the graphics context. The page format tells you the paper size and the print margins. The page number tells you which page to render.

To start a print job, use the `PrinterJob` class. First, call the static `getPrinterJob` method to get a print job object. Then set the `Printable` object that you want to print.

```
Printable canvas = . . .;
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(canvas);
```



**CAUTION:** The class `PrintJob` handles JDK 1.1-style printing. That class is now obsolete. Do not confuse it with the `PrinterJob` class.

---

Before starting the print job, you should call the `printDialog` method to display a print dialog box (see Figure 11.32). That dialog box gives the user a chance to select the printer to be used (in case multiple printers are available), the page range that should be printed, and various printer settings.

Collect printer settings in an object of a class that implements the `PrintRequestAttributeSet` interface, such as the `HashPrintRequestAttributeSet` class.

```
HashPrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
```

Add attribute settings and pass the `attributes` object to the `printDialog` method.

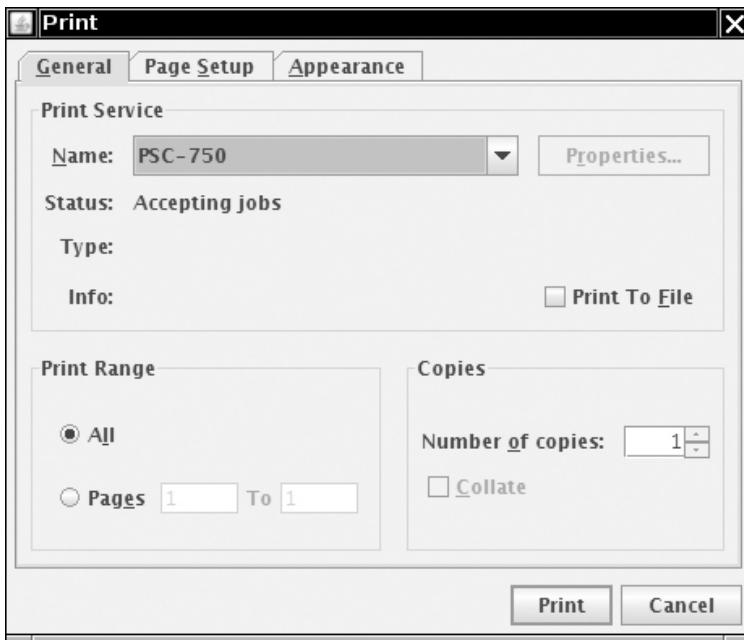


Figure 11.32 A cross-platform print dialog box

The `printDialog` method returns `true` if the user clicked OK and `false` if the user canceled the dialog box. If the user accepted, call the `print` method of the `PrinterJob` class to start the printing process. The `print` method might throw a `PrinterException`. Here is the outline of the printing code:

```
if (job.printDialog(attributes))
{
 try
 {
 job.print(attributes);
 }
 catch (PrinterException exception)
 {
 ...
 }
}
```

---

**NOTE:** Prior to JDK 1.4, the printing system used the native print and page setup dialog boxes of the host platform. To show a native print dialog box, call the `printDialog` method with no parameters. (There is no way to collect user settings in an attribute set.)

---

During printing, the `print` method of the `PrinterJob` class makes repeated calls to the `print` method of the `Printable` object associated with the job.

Since the job does not know how many pages you want to print, it simply keeps calling the `print` method. As long as the `print` method returns the value `Printable.PAGE_EXISTS`, the print job keeps producing pages. When the `print` method returns `Printable.NO_SUCH_PAGE`, the print job stops.



**CAUTION:** The page numbers that the print job passes to the `print` method start with page 0.

---

Therefore, the print job doesn't have an accurate page count until after the printout is complete. For that reason, the print dialog box can't display the correct page range—instead it displays “Pages 1 to 1.” You will see in the next section how to avoid this blemish by supplying a `Book` object to the print job.

During the printing process, the print job repeatedly calls the `print` method of the `Printable` object. The print job is allowed to make multiple calls *for the same page*. You should therefore not count pages inside the `print` method but always rely on the page number parameter. There is a good reason why the print job might call the `print` method repeatedly for the same page. Some printers, in particular dot-matrix and inkjet printers, use *banding*. They print one band at a time, advance the paper, and then print the next band. The print job might use banding even for laser printers that print a full page at a time—it gives the print job a way of managing the size of the spool file.

If the print job needs the `Printable` object to print a band, it sets the clip area of the graphics context to the requested band and calls the `print` method. Its drawing operations are clipped against the band rectangle, and only those drawing elements that show up in the band are rendered. Your `print` method need not be aware of that process, with one caveat: It should *not* interfere with the clip area.



**CAUTION:** The `Graphics` object that your `print` method gets is also clipped against the page margins. If you replace the clip area, you can draw outside the margins. Especially in a printer graphics context, the clipping area must be respected. Call `clip`, not `setClip`, to further restrict the clipping area. If you must remove a clip area, make sure to call `getClip` at the beginning of your `print` method and restore that clip area.

---

The `PageFormat` parameter of the `print` method contains information about the printed page. The methods `getWidth` and `getHeight` return the paper size, measured in *points*.

One point is 1/72 of an inch. (An inch equals 25.4 millimeters.) For example, A4 paper is approximately  $595 \times 842$  points, and US Letter paper is  $612 \times 792$  points.

Points are a common measurement in the printing trade in the United States. Much to the chagrin of the rest of the world, the printing package uses point units. There are two purposes for that: paper sizes and paper margins are measured in points, and points are the default unit for all print graphics contexts. You can verify that in the example program at the end of this section. The program prints two lines of text that are 72 units apart. Run the example program and measure the distance between the baselines; they are exactly 1 inch or 25.4 millimeters apart.

The `getWidth` and `getHeight` methods of the `PageFormat` class give you the complete paper size. Not all of the paper area is printable. Users typically select margins, and even if they don't, printers need to somehow grip the sheets of paper on which they print and therefore have a small unprintable area around the edges.

The methods `getImageableWidth` and `getImageableHeight` tell you the dimensions of the area that you can actually fill. However, the margins need not be symmetrical, so you must also know the top left corner of the imageable area (see Figure 11.33), which you obtain by the methods `getImageableX` and `getImageableY`.

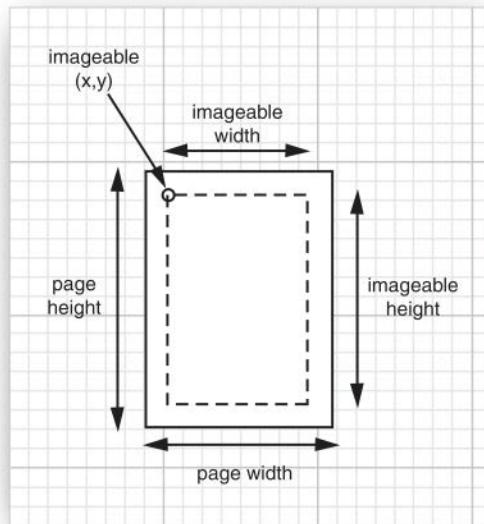


Figure 11.33 Page format measurements



**TIP:** The graphics context that you receive in the print method is clipped to exclude the margins, but the origin of the coordinate system is nevertheless the top left corner of the paper. It makes sense to translate the coordinate system to the top left corner of the imageable area. Simply start your print method with

```
g.translate(pageFormat.getImageableX(), pageFormat.getImageableY());
```

---

If you want your users to choose the settings for the page margins or to switch between portrait and landscape orientation without setting other printing attributes, call the `pageDialog` method of the `PrinterJob` class:

```
PageFormat format = job.pageDialog(attributes);
```

---

**NOTE:** One of the tabs of the print dialog box contains the page setup dialog (see Figure 11.34). You might still want to give users an option to set the page format before printing, especially if your program presents a “what you see is what you get” display of the pages to be printed. The `pageDialog` method returns a `PageFormat` object with the user settings.

---

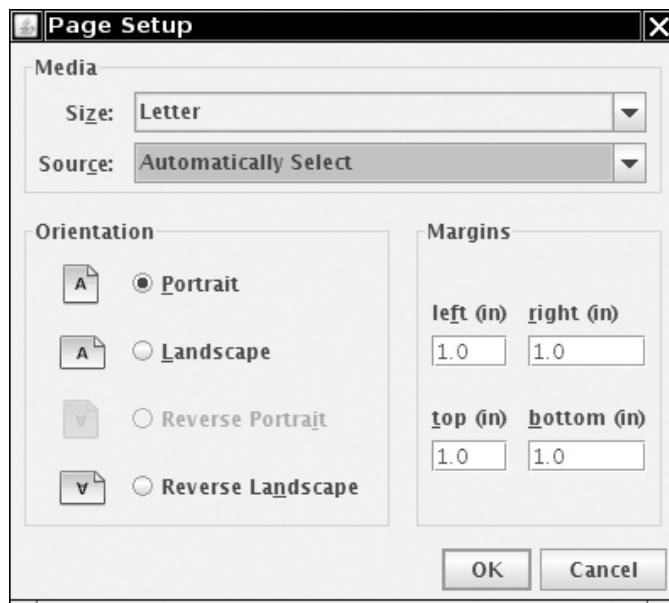


Figure 11.34 A cross-platform page setup dialog

The program in Listings 11.10 and 11.11 shows how to render the same set of shapes on the screen and on the printed page. A subclass of JPanel implements the Printable interface. Both the paintComponent and the print methods call the same method to carry out the actual drawing.

```
class PrintPanel extends JPanel implements Printable
{
 public void paintComponent(Graphics g)
 {
 super.paintComponent(g);
 Graphics2D g2 = (Graphics2D) g;
 drawPage(g2);
 }

 public int print(Graphics g, PageFormat pf, int page)
 throws PrinterException
 {
 if (page >= 1) return Printable.NO_SUCH_PAGE;
 Graphics2D g2 = (Graphics2D) g;
 g2.translate(pf.getImageableX(), pf.getImageableY());
 drawPage(g2);
 return Printable.PAGE_EXISTS;
 }

 public void drawPage(Graphics2D g2)
 {
 // shared drawing code goes here
 . .
 }
 . .
}
```

This example displays and prints the image shown in Figure 11.20 on p. 805—namely, the outline of the message “Hello, World” used as a clipping area for a pattern of lines.

Click the Print button to start printing, or click the Page setup button to open the page setup dialog box. Listing 11.10 shows the code.

---

**NOTE:** To show a native page setup dialog box, pass a default PageFormat object to the pageDialog method. The method clones that object, modifies it according to the user selections in the dialog box, and returns the cloned object.

```
PageFormat defaultFormat = printJob.defaultPage();
PageFormat selectedFormat = printJob.pageDialog(defaultFormat);
```

---

**Listing 11.10** print/PrintTestFrame.java

```
1 package print;
2
3 import java.awt.*;
4 import java.awt.print.*;
5
6 import javax.print.attribute.*;
7 import javax.swing.*;
8
9 /**
10 * This frame shows a panel with 2D graphics and buttons to print the graphics and to set up the
11 * page format.
12 */
13 public class PrintTestFrame extends JFrame
14 {
15 private PrintComponent canvas;
16 private PrintRequestAttributeSet attributes;
17
18 public PrintTestFrame()
19 {
20 canvas = new PrintComponent();
21 add(canvas, BorderLayout.CENTER);
22
23 attributes = new HashPrintRequestAttributeSet();
24
25 JPanel buttonPanel = new JPanel();
26 JButton printButton = new JButton("Print");
27 buttonPanel.add(printButton);
28 printButton.addActionListener(event ->
29 {
30 try
31 {
32 PrinterJob job = PrinterJob.getPrinterJob();
33 job.setPrintable(canvas);
34 if (job.printDialog(attributes)) job.print(attributes);
35 }
36 catch (PrinterException ex)
37 {
38 JOptionPane.showMessageDialog(PrintTestFrame.this, ex);
39 }
40 });
41
42 JButton pageSetupButton = new JButton("Page setup");
43 buttonPanel.add(pageSetupButton);
44 pageSetupButton.addActionListener(event ->
45 {
46 PrinterJob job = PrinterJob.getPrinterJob();
47 job.pageDialog(attributes);
48 });
49
50 JPanel previewPanel = new JPanel();
51 previewPanel.setLayout(new GridLayout(1, 1));
52 previewPanel.add(canvas);
53
54 add(buttonPanel, "North");
55 add(previewPanel, "Center");
56 }
57
58 protected void paint(Graphics g)
59 {
60 super.paint(g);
61 canvas.paint(g);
62 }
63
64 public static void main(String[] args)
65 {
66 PrintTestFrame frame = new PrintTestFrame();
67 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
68 frame.setSize(400, 300);
69 frame.setVisible(true);
70 }
71 }
```

```
49 add(buttonPanel, BorderLayout.NORTH);
50 pack();
51 }
52 }
53 }
```

**Listing 11.11** print/PrintComponent.java

```
1 package print;
2
3 import java.awt.*;
4 import java.awt.font.*;
5 import java.awt.geom.*;
6 import java.awt.print.*;
7 import javax.swing.*;
8
9 /**
10 * This component generates a 2D graphics image for screen display and printing.
11 */
12 public class PrintComponent extends JComponent implements Printable
13 {
14 private static final Dimension PREFERRED_SIZE = new Dimension(300, 300);
15 public void paintComponent(Graphics g)
16 {
17 Graphics2D g2 = (Graphics2D) g;
18 drawPage(g2);
19 }
20
21 public int print(Graphics g, PageFormat pf, int page) throws PrinterException
22 {
23 if (page >= 1) return Printable.NO_SUCH_PAGE;
24 Graphics2D g2 = (Graphics2D) g;
25 g2.translate(pf.getImageableX(), pf.getImageableY());
26 g2.draw(new Rectangle2D.Double(0, 0, pf.getImageableWidth(), pf.getImageableHeight()));
27
28 drawPage(g2);
29 return Printable.PAGE_EXISTS;
30 }
31
32 /**
33 * This method draws the page both on the screen and the printer graphics context.
34 * @param g2 the graphics context
35 */
36 public void drawPage(Graphics2D g2)
37 {
```

(Continues)

**Listing 11.11 (Continued)**

```
38 FontRenderContext context = g2.getFontRenderContext();
39 Font f = new Font("Serif", Font.PLAIN, 72);
40 GeneralPath clipShape = new GeneralPath();
41
42 TextLayout layout = new TextLayout("Hello", f, context);
43 AffineTransform transform = AffineTransform.getTranslateInstance(0, 72);
44 Shape outline = layout.getOutline(transform);
45 clipShape.append(outline, false);
46
47 layout = new TextLayout("World", f, context);
48 transform = AffineTransform.getTranslateInstance(0, 144);
49 outline = layout.getOutline(transform);
50 clipShape.append(outline, false);
51
52 g2.draw(clipShape);
53 g2.clip(clipShape);
54
55 final int NLINES = 50;
56 Point2D p = new Point2D.Double(0, 0);
57 for (int i = 0; i < NLINES; i++)
58 {
59 double x = (2 * getWidth() * i) / NLINES;
60 double y = (2 * getHeight() * (NLINES - 1 - i)) / NLINES;
61 Point2D q = new Point2D.Double(x, y);
62 g2.draw(new Line2D.Double(p, q));
63 }
64 }
65
66 public Dimension getPreferredSize() { return PREFERRED_SIZE; }
67 }
```

---

**java.awt.print.Printable 1.2**

- `int print(Graphics g, PageFormat format, int pageNumber)`

renders a page and returns PAGE\_EXISTS, or returns NO\_SUCH\_PAGE.

*Parameters:*

`g`

The graphics context onto which the page is rendered

`format`

The format of the page to draw on

`pageNumber`

The number of the requested page

**java.awt.print.PrinterJob 1.2**

- static PrinterJob getPrinterJob()  
returns a printer job object.
- PageFormat defaultPage()  
returns the default page format for this printer.
- boolean printDialog(PrintRequestAttributeSet attributes)
- boolean printDialog()  
opens a print dialog box to allow a user to select the pages to be printed and to change print settings. The first method displays a cross-platform dialog box, the second a native dialog box. The first method modifies the attributes object to reflect the user settings. Both methods return true if the user accepts the dialog box.
- PageFormat pageDialog(PrintRequestAttributeSet attributes)
- PageFormat pageDialog(PageFormat defaults)  
displays a page setup dialog box. The first method displays a cross-platform dialog box, the second a native dialog box. Both methods return a PageFormat object with the format that the user requested in the dialog box. The first method modifies the attributes object to reflect the user settings. The second method does not modify the defaults object.
- void setPrintable(Printable p)
- void setPrintable(Printable p, PageFormat format)  
sets the Printable of this print job and an optional page format.
- void print()
- void print(PrintRequestAttributeSet attributes)  
prints the current Printable by repeatedly calling its print method and sending the rendered pages to the printer, until no more pages are available.

**java.awt.print.PageFormat 1.2**

- double getWidth()  
• double getHeight()  
returns the width and height of the page.
- double getImageableWidth()  
• double getImageableHeight()  
returns the width and height of the imageable area of the page.

(Continues)

**java.awt.print.PageFormat 1.2 (Continued)**

- `double getImageableX()`
- `double getImageableY()`  
returns the position of the top left corner of the imageable area.
- `int getOrientation()`  
returns one of `PORTRAIT`, `LANDSCAPE`, or `REVERSE_LANDSCAPE`. Page orientation is transparent to programmers because the page format and graphics context settings automatically reflect the page orientation.

### 11.12.2 Multiple-Page Printing

In practice, you usually shouldn't pass a raw `Printable` object to a print job. Instead, you should obtain an object of a class that implements the `Pageable` interface. The Java platform supplies one such class, called `Book`. A book is made up of sections, each of which is a `Printable` object. To make a book, add `Printable` objects and their page counts.

```
Book book = new Book();
Printable coverPage = . . .;
Printable bodyPages = . . .;
book.append(coverPage, pageFormat); // append 1 page
book.append(bodyPages, pageCount);
```

Then, use the `setPageable` method to pass the `Book` object to the print job.

```
printJob.setPageable(book);
```

Now the print job knows exactly how many pages to print, so the print dialog box displays an accurate page range and the user can select the entire range or subranges.



**CAUTION:** When the print job calls the `print` methods of the `Printable` sections, it passes the current page number of the *book*, and not of each *section*, as the current page number. That is a huge pain—each section must know the page counts of the preceding sections to make sense of the page number parameter.

From your perspective as a programmer, the biggest challenge of using the `Book` class is that you must know how many pages each section will have when you print it. Your `Printable` class needs a *layout algorithm* that computes the layout of the material on the printed pages. Before printing starts, invoke that algorithm

to compute the page breaks and the page count. You can retain the layout information so you have it handy during the printing process.

You must guard against the possibility that the user has changed the page format. If that happens, you must recompute the layout, even if the information that you want to print has not changed.

Listing 11.13 shows how to produce a multipage printout. This program prints a message in very large characters on a number of pages (see Figure 11.35). You can then trim the margins and tape the pages together to form a banner.

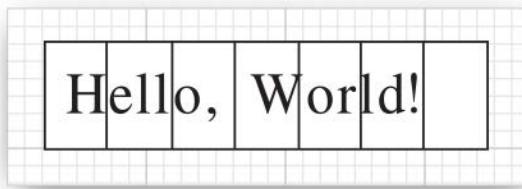


Figure 11.35 A banner

The `layoutPages` method of the `Banner` class computes the layout. We first lay out the message string in a 72-point font. We then compute the height of the resulting string and compare it with the imageable height of the page. We derive a scale factor from these two measurements. When printing the string, we magnify it by that scale factor.

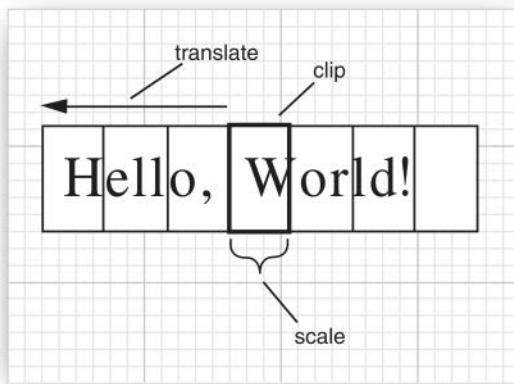


**CAUTION:** To lay out your information precisely, you usually need access to the printer graphics context. Unfortunately, there is no way to obtain that graphics context before printing actually starts. In our example program, we make do with the screen graphics context and hope that the font metrics of the screen and printer match.

The `getPageCount` method of the `Banner` class first calls the `layout` method. Then it scales up the width of the string and divides it by the imageable width of each page. The quotient, rounded up to the next integer, is the page count.

It sounds like it might be difficult to print the banner because characters can be broken across multiple pages. However, thanks to the power of the Java 2D API, this turns out not to be a problem at all. When a particular page is requested, we simply use the `translate` method of the `Graphics2D` class to shift the top left corner of the string to the left. Then, we set a clip rectangle that equals the current page

(see Figure 11.36). Finally, we scale the graphics context with the scale factor that the layout method computed.



**Figure 11.36** Printing a page of a banner

This example shows the power of transformations. The drawing code is kept simple, and the transformation does all the work of placing the drawing in the appropriate place. Finally, the clip cuts away the part of the image that falls outside the page. In the next section, you will see another compelling use of transformations—to display a print preview.

### 11.12.3 Print Preview

Most professional programs have a print preview mechanism that lets you look at your pages on the screen so you won't waste paper on a printout that you don't like. The printing classes of the Java platform do not supply a standard "print preview" dialog box, but it is easy to design your own (see Figure 11.37). In this section, we'll show you how. The `PrintPreviewDialog` class in Listing 11.14 is completely generic—you can reuse it to preview any kind of printout.

To construct a `PrintPreviewDialog`, supply either a `Printable` or a `Book`, together with a `PageFormat` object. The dialog box contains a `PrintPreviewCanvas` (see Listing 11.15). As you use the Next and Previous buttons to flip through the pages, the `paintComponent` method calls the `print` method of the `Printable` object for the requested page.

Normally, the `print` method draws the page context on a printer graphics context. However, we supply the screen graphics context, suitably scaled so that the entire printed page fits inside a small screen rectangle.

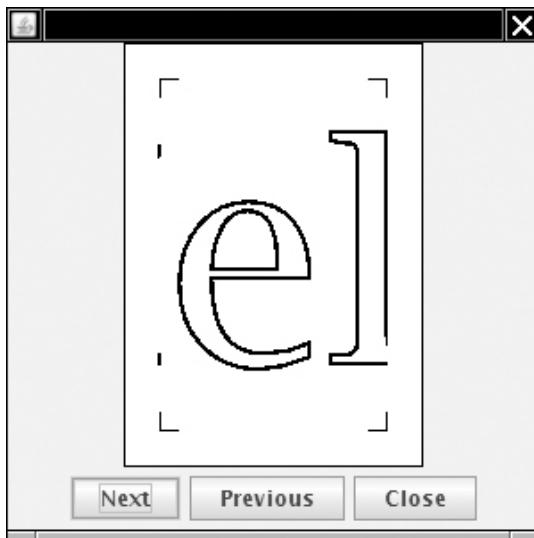


Figure 11.37 The print preview dialog, showing a banner page

```
float xoff = . . .; // left of page
float yoff = . . .; // top of page
float scale = . . .; // to fit printed page onto screen
g2.translate(xoff, yoff);
g2.scale(scale, scale);
Printable printable = book.getPrintable(currentPage);
printable.print(g2, pageFormat, currentPage);
```

The print method never knows that it doesn't actually produce printed pages. It simply draws onto the graphics context, producing a microscopic print preview on the screen. This is a compelling demonstration of the power of the Java 2D imaging model.

Listing 11.12 contains the code for the banner printing program. Type "Hello, World!" into the text field and look at the print preview, then print the banner.

---

**Listing 11.12** book/BookTestFrame.java

---

```
1 package book;
2
3 import java.awt.*;
4 import java.awt.print.*;
5
6 import javax.print.attribute.*;
```

(Continues)

**Listing 11.12 (Continued)**

```
7 import javax.swing.*;
8 /**
9 * This frame has a text field for the banner text and buttons for printing, page setup, and print
10 * preview.
11 */
12 public class BookTestFrame extends JFrame
13 {
14 private JTextField text;
15 private PageFormat pageFormat;
16 private PrintRequestAttributeSet attributes;
17
18 public BookTestFrame()
19 {
20 text = new JTextField();
21 add(text, BorderLayout.NORTH);
22
23 attributes = new HashPrintRequestAttributeSet();
24
25 JPanel buttonPanel = new JPanel();
26
27 JButton printButton = new JButton("Print");
28 buttonPanel.add(printButton);
29 printButton.addActionListener(event ->
30 {
31 try
32 {
33 PrinterJob job = PrinterJob.getPrinterJob();
34 job.setPageable(makeBook());
35 if (job.printDialog(attributes))
36 {
37 job.print(attributes);
38 }
39 }
40 catch (PrinterException e)
41 {
42 JOptionPane.showMessageDialog(BookTestFrame.this, e);
43 }
44 });
45
46 JButton pageSetupButton = new JButton("Page setup");
47 buttonPanel.add(pageSetupButton);
48 pageSetupButton.addActionListener(event ->
49 {
50 PrinterJob job = PrinterJob.getPrinterJob();
51 pageFormat = job.pageDialog(attributes);
52 });
53
54 }
```

```
55 JButton printPreviewButton = new JButton("Print preview");
56 buttonPanel.add(printPreviewButton);
57 printPreviewButton.addActionListener(event ->
58 {
59 PrintPreviewDialog dialog = new PrintPreviewDialog(makeBook());
60 dialog.setVisible(true);
61 });
62
63 add(buttonPanel, BorderLayout.SOUTH);
64 pack();
65 }
66
67 /**
68 * Makes a book that contains a cover page and the pages for the banner.
69 */
70 public Book makeBook()
71 {
72 if (pageFormat == null)
73 {
74 PrinterJob job = PrinterJob.getPrinterJob();
75 pageFormat = job.defaultPage();
76 }
77 Book book = new Book();
78 String message = text.getText();
79 Banner banner = new Banner(message);
80 int pageCount = banner.getPageCount((Graphics2D) getGraphics(), pageFormat);
81 book.append(new CoverPage(message + " (" + pageCount + " pages)", pageFormat));
82 book.append(banner, pageFormat, pageCount);
83 return book;
84 }
85 }
```

---

**Listing 11.13** book/Banner.java

---

```
1 package book;
2
3 import java.awt.*;
4 import java.awt.font.*;
5 import java.awt.geom.*;
6 import java.awt.print.*;
7
8 /**
9 * A banner that prints a text string on multiple pages.
10 */
11 public class Banner implements Printable
12 {
13 private String message;
14 private double scale;
```

(Continues)

**Listing 11.13 (Continued)**

```
15 /**
16 * Constructs a banner.
17 * @param m the message string
18 */
19 public Banner(String m)
20 {
21 message = m;
22 }
23
24 /**
25 * Gets the page count of this section.
26 * @param g2 the graphics context
27 * @param pf the page format
28 * @return the number of pages needed
29 */
30 public int getPageCount(Graphics2D g2, PageFormat pf)
31 {
32 if (message.equals("")) return 0;
33 FontRenderContext context = g2.getFontRenderContext();
34 Font f = new Font("Serif", Font.PLAIN, 72);
35 Rectangle2D bounds = f.getStringBounds(message, context);
36 scale = pf.getImageableHeight() / bounds.getHeight();
37 double width = scale * bounds.getWidth();
38 int pages = (int) Math.ceil(width / pf.getImageableWidth());
39 return pages;
40 }
41
42
43 public int print(Graphics g, PageFormat pf, int page) throws PrinterException
44 {
45 Graphics2D g2 = (Graphics2D) g;
46 if (page > getPageCount(g2, pf)) return Printable.NO_SUCH_PAGE;
47 g2.translate(pf.getImageableX(), pf.getImageableY());
48
49 drawPage(g2, pf, page);
50 return Printable.PAGE_EXISTS;
51 }
52
53 public void drawPage(Graphics2D g2, PageFormat pf, int page)
54 {
55 if (message.equals("")) return;
56 page--; // account for cover page
57
58 drawCropMarks(g2, pf);
59 g2.clip(new Rectangle2D.Double(0, 0, pf.getImageableWidth(), pf.getImageableHeight()));
60 g2.translate(-page * pf.getImageableWidth(), 0);
61 g2.scale(scale, scale);
62 FontRenderContext context = g2.getFontRenderContext();
```

```
63 Font f = new Font("Serif", Font.PLAIN, 72);
64 TextLayout layout = new TextLayout(message, f, context);
65 AffineTransform transform = AffineTransform.getTranslateInstance(0, layout.getAscent());
66 Shape outline = layout.getOutline(transform);
67 g2.draw(outline);
68 }
69
70 /**
71 * Draws 1/2" crop marks in the corners of the page.
72 * @param g2 the graphics context
73 * @param pf the page format
74 */
75 public void drawCropMarks(Graphics2D g2, PageFormat pf)
76 {
77 final double C = 36; // crop mark length = 1/2 inch
78 double w = pf.getImageableWidth();
79 double h = pf.getImageableHeight();
80 g2.draw(new Line2D.Double(0, 0, 0, C));
81 g2.draw(new Line2D.Double(0, 0, C, 0));
82 g2.draw(new Line2D.Double(w, 0, w, C));
83 g2.draw(new Line2D.Double(w, 0, w - C, 0));
84 g2.draw(new Line2D.Double(0, h, 0, h - C));
85 g2.draw(new Line2D.Double(0, h, C, h));
86 g2.draw(new Line2D.Double(w, h, w, h - C));
87 g2.draw(new Line2D.Double(w, h, w - C, h));
88 }
89 }
90
91 /**
92 * This class prints a cover page with a title.
93 */
94 class CoverPage implements Printable
95 {
96 private String title;
97
98 /**
99 * Constructs a cover page.
100 * @param t the title
101 */
102 public CoverPage(String t)
103 {
104 title = t;
105 }
106
107 public int print(Graphics g, PageFormat pf, int page) throws PrinterException
108 {
109 if (page >= 1) return Printable.NO_SUCH_PAGE;
110 Graphics2D g2 = (Graphics2D) g;
```

(Continues)

**Listing 11.13** *(Continued)*

```
111 g2.setPaint(Color.black);
112 g2.translate(pf.getImageableX(), pf.getImageableY());
113 FontRenderContext context = g2.getFontRenderContext();
114 Font f = g2.getFont();
115 TextLayout layout = new TextLayout(title, f, context);
116 float ascent = layout.getAscent();
117 g2.drawString(title, 0, ascent);
118 return Printable.PAGE_EXISTS;
119 }
120 }
```

---

**Listing 11.14** book/PrintPreviewDialog.java

```
1 package book;
2
3 import java.awt.*;
4 import java.awt.print.*;
5
6 import javax.swing.*;
7
8 /**
9 * This class implements a generic print preview dialog.
10 */
11 public class PrintPreviewDialog extends JDialog
12 {
13 private static final int DEFAULT_WIDTH = 300;
14 private static final int DEFAULT_HEIGHT = 300;
15
16 private PrintPreviewCanvas canvas;
17
18 /**
19 * Constructs a print preview dialog.
20 * @param p a Printable
21 * @param pf the page format
22 * @param pages the number of pages in p
23 */
24 public PrintPreviewDialog(Printable p, PageFormat pf, int pages)
25 {
26 Book book = new Book();
27 book.append(p, pf, pages);
28 layoutUI(book);
29 }
30
31 /**
32 * Constructs a print preview dialog.
33 * @param b a Book
34 */
```

```
35 public PrintPreviewDialog(Book b)
36 {
37 layoutUI(b);
38 }
39
40 /**
41 * Lays out the UI of the dialog.
42 * @param book the book to be previewed
43 */
44 public void layoutUI(Book book)
45 {
46 setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
47
48 canvas = new PrintPreviewCanvas(book);
49 add(canvas, BorderLayout.CENTER);
50
51 JPanel buttonPanel = new JPanel();
52
53 JButton nextButton = new JButton("Next");
54 buttonPanel.add(nextButton);
55 nextButton.addActionListener(event -> canvas.flipPage(1));
56
57 JButton previousButton = new JButton("Previous");
58 buttonPanel.add(previousButton);
59 previousButton.addActionListener(event -> canvas.flipPage(-1));
60
61 JButton closeButton = new JButton("Close");
62 buttonPanel.add(closeButton);
63 closeButton.addActionListener(event -> setVisible(false));
64
65 add(buttonPanel, BorderLayout.SOUTH);
66 }
67 }
```

---

**Listing 11.15** book/PrintPreviewCanvas.java

---

```
1 package book;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.awt.print.*;
6 import javax.swing.*;
7
8 /**
9 * The canvas for displaying the print preview.
10 */
11
```

(Continues)

**Listing 11.15 (Continued)**

```
11 class PrintPreviewCanvas extends JPanel
12 {
13 private Book book;
14 private int currentPage;
15
16 /**
17 * Constructs a print preview canvas.
18 * @param b the book to be previewed
19 */
20 public PrintPreviewCanvas(Book b)
21 {
22 book = b;
23 currentPage = 0;
24 }
25
26 public void paintComponent(Graphics g)
27 {
28 Graphics2D g2 = (Graphics2D) g;
29 PageFormat pageFormat = book.getPageFormat(currentPage);
30
31 double xoff; // x offset of page start in window
32 double yoff; // y offset of page start in window
33 double scale; // scale factor to fit page in window
34 double px = pageFormat.getWidth();
35 double py = pageFormat.getHeight();
36 double sx = getWidth() - 1;
37 double sy = getHeight() - 1;
38 if (px / py < sx / sy) // center horizontally
39 {
40 scale = sy / py;
41 xoff = 0.5 * (sx - scale * px);
42 yoff = 0;
43 }
44 else
45 // center vertically
46 {
47 scale = sx / px;
48 xoff = 0;
49 yoff = 0.5 * (sy - scale * py);
50 }
51 g2.translate((float) xoff, (float) yoff);
52 g2.scale((float) scale, (float) scale);
53
54 // draw page outline (ignoring margins)
55 Rectangle2D page = new Rectangle2D.Double(0, 0, px, py);
56 g2.setPaint(Color.white);
57 g2.fill(page);
```

```
58 g2.setPaint(Color.black);
59 g2.draw(page);
60
61 Printable printable = book.getPrintable(currentPage);
62 try
63 {
64 printable.print(g2, pageFormat, currentPage);
65 }
66 catch (PrinterException e)
67 {
68 g2.draw(new Line2D.Double(0, 0, px, py));
69 g2.draw(new Line2D.Double(px, 0, 0, py));
70 }
71 }
72
73 /**
74 * Flip the book by the given number of pages.
75 * @param by the number of pages to flip by. Negative values flip backward.
76 */
77 public void flipPage(int by)
78 {
79 int newPage = currentPage + by;
80 if (0 <= newPage && newPage < book.getNumberOfPages())
81 {
82 currentPage = newPage;
83 repaint();
84 }
85 }
86 }
```

**java.awt.print.PrinterJob 1.2**

- `void setPageable(Pageable p)`  
sets a `Pageable` (such as a `Book`) to be printed.

**java.awt.print.Book 1.2**

- `void append(Printable p, PageFormat format)`
- `void append(Printable p, PageFormat format, int pageCount)`  
appends a section to this book. If the page count is not specified, the first page is added.
- `Printable getPrintable(int page)`  
gets the `printable` for the specified page.

### 11.12.4 Print Services

So far, you have seen how to print 2D graphics. However, the printing API introduced in Java SE 1.4 affords far greater flexibility. The API defines a number of data types and lets you find print services that are able to print them. Among the data types are

- Images in GIF, JPEG, or PNG format
- Documents in text, HTML, PostScript, or PDF format
- Raw printer code data
- Objects of a class that implements `Printable`, `Pageable`, or `RenderableImage`

The data themselves can be stored in a source of bytes or characters such as an input stream, a URL, or an array. A *document flavor* describes the combination of a data source and a data type. The `DocFlavor` class defines a number of inner classes for the various data sources. Each of the inner classes defines constants to specify the flavors. For example, the constant

```
DocFlavor.INPUT_STREAM.GIF
```

describes a GIF image that is read from an input stream. Table 11.3 lists the combinations.

Suppose you want to print a GIF image located in a file. First, find out whether there is a *print service* that is capable of handling the task. The static `lookupPrintServices` method of the `PrintServiceLookup` class returns an array of `PrintService` objects that can handle the given document flavor.

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
PrintService[] services = PrintServiceLookup.lookupPrintServices(flavor, null);
```

The second parameter of the `lookupPrintServices` method is `null` to indicate that we don't want to constrain the search by specifying printer attributes. We'll cover attributes in the next section.

If the lookup yields an array with more than one element, select from the listed print services. You can call the `getName` method of the `PrintService` class to get the printer names and let the user choose.

Next, get a document print job from the service:

```
DocPrintJob job = services[i].createPrintJob();
```

For printing, you need an object that implements the `Doc` interface. The Java library supplies a class `SimpleDoc` for that purpose. The `SimpleDoc` constructor requires the data source object, the document flavor, and an optional attribute set. For example,

**Table 11.3** Document Flavors for Print Services

| Data Source       | Data Type           | MIME Type                                                         |
|-------------------|---------------------|-------------------------------------------------------------------|
| INPUT_STREAM      | GIF                 | image/gif                                                         |
| URL               | JPEG                | image/jpeg                                                        |
| BYTE_ARRAY        | PNG                 | image/png                                                         |
|                   | POSTSCRIPT          | application/postscript                                            |
|                   | PDF                 | application/pdf                                                   |
|                   | TEXT_HTML_HOST      | text/html (using host encoding)                                   |
|                   | TEXT_HTML_US_ASCII  | text/html; charset=us-ascii                                       |
|                   | TEXT_HTML_UTF_8     | text/html; charset=utf-8                                          |
|                   | TEXT_HTML_UTF_16    | text/html; charset=utf-16                                         |
|                   | TEXT_HTML_UTF_16LE  | text/html; charset=utf-16le (little-endian)                       |
|                   | TEXT_HTML_UTF_16BE  | text/html; charset=utf-16be (big-endian)                          |
|                   | TEXT_PLAIN_HOST     | text/plain (using host encoding)                                  |
|                   | TEXT_PLAIN_US_ASCII | text/plain; charset=us-ascii                                      |
|                   | TEXT_PLAIN_UTF_8    | text/plain; charset=utf-8                                         |
|                   | TEXT_PLAIN_UTF_16   | text/plain; charset=utf-16                                        |
|                   | TEXT_PLAIN_UTF_16LE | text/plain; charset=utf-16le (little-endian)                      |
|                   | TEXT_PLAIN_UTF_16BE | text/plain; charset=utf-16be (big-endian)                         |
|                   | PCL                 | application/vnd.hp-PCL (Hewlett Packard Printer Control Language) |
|                   | AUTONSENSE          | application/octet-stream (raw printer data)                       |
| READER            | TEXT_HTML           | text/html; charset=utf-16                                         |
| STRING            | TEXT_PLAIN          | text/plain; charset=utf-16                                        |
| CHAR_ARRAY        |                     |                                                                   |
| SERVICE_FORMATTED | PRINTABLE           | N/A                                                               |
|                   | PAGEABLE            | N/A                                                               |
|                   | RENDERABLE_IMAGE    | N/A                                                               |

```
InputStream in = new FileInputStream(fileName);
Doc doc = new SimpleDoc(in, flavor, null);
```

Finally, you are ready to print:

```
job.print(doc, null);
```

As before, the `null` parameter can be replaced by an attribute set.

Note that this printing process is quite different from that of the preceding section. There is no user interaction through print dialog boxes. For example, you can implement a server-side printing mechanism in which users submit print jobs through a web form.

The program in Listing 11.16 demonstrates how to use a print service to print an image file.

---

**Listing 11.16** printService/PrintServiceTest.java

---

```
1 package printService;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import javax.print.*;
6
7 /**
8 * This program demonstrates the use of print services. The program lets you print a GIF image to
9 * any of the print services that support the GIF document flavor.
10 * @version 1.10 2007-08-16
11 * @author Cay Horstmann
12 */
13 public class PrintServiceTest
14 {
15 public static void main(String[] args)
16 {
17 DocFlavor flavor = DocFlavor.URL.GIF;
18 PrintService[] services = PrintServiceLookup.lookupPrintServices(flavor, null);
19 if (args.length == 0)
20 {
21 if (services.length == 0) System.out.println("No printer for flavor " + flavor);
22 else
23 {
24 System.out.println("Specify a file of flavor " + flavor
25 + "\nand optionally the number of the desired printer.");
26 for (int i = 0; i < services.length; i++)
27 System.out.println((i + 1) + ": " + services[i].getName());
28 }
29 System.exit(0);
30 }
```

```
31 String fileName = args[0];
32 int p = 1;
33 if (args.length > 1) p = Integer.parseInt(args[1]);
34 if (fileName == null) return;
35 try (InputStream in = Files.newInputStream(Paths.get(fileName)))
36 {
37 Doc doc = new SimpleDoc(in, flavor, null);
38 DocPrintJob job = services[p - 1].createPrintJob();
39 job.print(doc, null);
40 }
41 catch (Exception ex)
42 {
43 ex.printStackTrace();
44 }
45 }
46 }
```

#### *javax.print.PrintServiceLookup* 1.4

- `PrintService[] lookupPrintServices(DocFlavor flavor, AttributeSet attributes)`  
looks up the print services that can handle the given document flavor and attributes.  
*Parameters:* flavor The document flavor  
                  attributes The required printing attributes, or `null` if attributes should not be considered

#### *javax.print.PrintService* 1.4

- `DocPrintJob createPrintJob()`  
creates a print job for printing an object of a class that implements the `Doc` interface, such as a `SimpleDoc`.

#### *javax.print.DocPrintJob* 1.4

- `void print(Doc doc, PrintRequestAttributeSet attributes)`  
prints the given document with the given attributes.  
*Parameters:* doc The `Doc` to be printed  
                  attributes The required printing attributes, or `null` if no printing attributes are required

**javax.print.SimpleDoc 1.4**

- `SimpleDoc(Object data, DocFlavor flavor, DocAttributeSet attributes)`

constructs a `SimpleDoc` object that can be printed with a `DocPrintJob`.

|                    |                         |                                                                                     |
|--------------------|-------------------------|-------------------------------------------------------------------------------------|
| <i>Parameters:</i> | <code>data</code>       | The object with the print data, such as an input stream or a <code>Printable</code> |
|                    | <code>flavor</code>     | The document flavor of the print data                                               |
|                    | <code>attributes</code> | Document attributes, or <code>null</code> if attributes are not required            |

### 11.12.5 Stream Print Services

A print service sends print data to a printer. A stream print service generates the same print data but instead sends them to a stream, perhaps for delayed printing or because the print data format can be interpreted by other programs. In particular, if the print data format is PostScript, it may be useful to save the print data to a file because many programs can process PostScript files. The Java platform includes a stream print service that can produce PostScript output from images and 2D graphics. You can use that service on all systems, even if there are no local printers.

Enumerating stream print services is a bit more tedious than locating regular print services. You need both the `DocFlavor` of the object to be printed and the MIME type of the stream output. You then get a `StreamPrintServiceFactory` array of factories.

```
DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
String mimeType = "application/postscript";
StreamPrintServiceFactory[] factories
 = StreamPrintServiceFactory.lookupStreamPrintServiceFactories(flavor, mimeType);
```

The `StreamPrintServiceFactory` class has no methods that would help us distinguish any one factory from another, so we just take `factories[0]`. We call the `getPrintService` method with an output stream parameter to get a `StreamPrintService` object.

```
OutputStream out = new FileOutputStream(fileName);
StreamPrintService service = factories[0].getPrintService(out);
```

The `StreamPrintService` class is a subclass of `PrintService`. To produce a printout, simply follow the steps of the preceding section.

**javax.print.StreamPrintServiceFactory 1.4**

- `StreamPrintServiceFactory[] lookupStreamPrintServiceFactories(DocFlavor flavor, String mimeType)`  
looks up the stream print service factories that can print the given document flavor and produce an output stream of the given MIME type.
- `StreamPrintService getPrintService(OutputStream out)`  
gets a print service that sends the printing output to the given output stream.

## 11.12.6 Printing Attributes

The print service API contains a complex set of interfaces and classes to specify various kinds of attributes. There are four important groups of attributes. The first two specify requests to the printer.

- *Print request attributes* request particular features for all `doc` objects in a print job, such as two-sided printing or the paper size.
- *Doc attributes* are request attributes that apply only to a single `doc` object.

The other two attributes contain information about the printer and job status.

- *Print service attributes* give information about the print service, such as the printer make and model or whether the printer is currently accepting jobs.
- *Print job attributes* give information about the status of a particular print job, such as whether the job is already completed.

To describe the various attributes, there is an interface `Attribute` with subinterfaces:

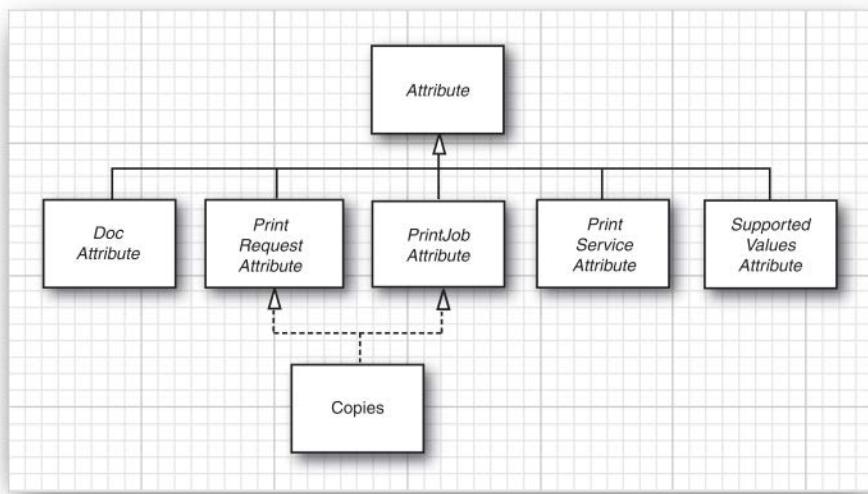
`PrintRequestAttribute`  
`DocAttribute`  
`PrintServiceAttribute`  
`PrintJobAttribute`  
`SupportedValuesAttribute`

Individual attribute classes implement one or more of these interfaces. For example, objects of the `Copies` class describe the number of copies of a printout. That class implements both the `PrintRequestAttribute` and the `PrintJobAttribute` interfaces. Clearly, a print request can contain a request for multiple copies. Conversely, an attribute of the print job might be how many of these copies were actually printed. That number might be lower, perhaps because of printer limitations or because the printer ran out of paper.

The `SupportedValuesAttribute` interface indicates that an attribute value does not reflect actual request or status data but rather the capability of a service. For example,

the `CopiesSupported` class implements the `SupportedValuesAttribute` interface. An object of that class might describe that a printer supports 1 through 99 copies of a printout.

Figure 11.38 shows a class diagram of the attribute hierarchy.



**Figure 11.38** The attribute hierarchy

In addition to the interfaces and classes for individual attributes, the print service API defines interfaces and classes for attribute sets. A superinterface, `AttributeSet`, has four subinterfaces:

```

PrintRequestAttributeSet
DocAttributeSet
PrintServiceAttributeSet
PrintJobAttributeSet

```

Each of these interfaces has an implementing class, yielding the five classes:

```

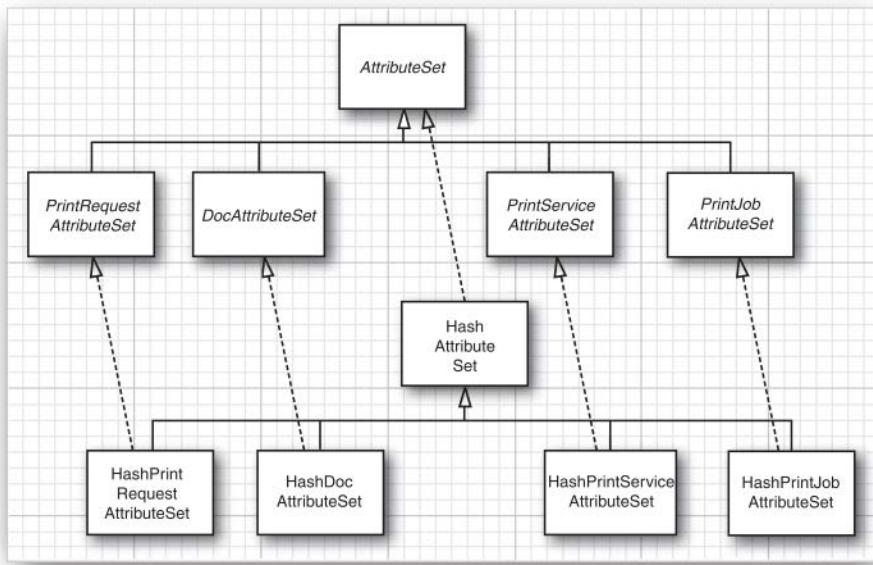
HashAttributeSet
HashPrintRequestAttributeSet
HashDocAttributeSet
HashPrintServiceAttributeSet
HashPrintJobAttributeSet

```

Figure 11.39 shows a class diagram of the attribute set hierarchy.

For example, you can construct a print request attribute set like this:

```
PrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
```



**Figure 11.39** The attribute set hierarchy

After constructing the set, you are freed from worrying about the `Hash` prefix.

Why have all these interfaces? They make it possible to check for correct attribute usage. For example, a `DocAttributeSet` accepts only objects that implement the `DocAttribute` interface. Any attempt to add another attribute results in a runtime error.

An attribute set is a specialized kind of map where the keys are of type `Class` and the values belong to a class that implements the `Attribute` interface. For example, if you insert an object

```
new Copies(10)
```

into an attribute set, then its key is the `Class` object `Copies.class`. That key is called the *category* of the attribute. The `Attribute` interface declares a method

```
Class getCategory()
```

that returns the category of an attribute. The `Copies` class defines the method to return the object `Copies.class`, but it isn't a requirement that the category be the same as the class of the attribute.

When an attribute is added to an attribute set, the category is extracted automatically. Just add the attribute value:

```
attributes.add(new Copies(10));
```

If you subsequently add another attribute with the same category, it overwrites the first one.

To retrieve an attribute, you need to use the category as the key, for example:

```
AttributeSet attributes = job.getAttributes();
Copies copies = (Copies) attribute.get(Copies.class);
```

Finally, attributes are organized by the values they can have. The `Copies` attribute can have any integer value. The `Copies` class extends the `IntegerSyntax` class that takes care of all integer-valued attributes. The `getValue` method returns the integer value of the attribute, for example:

```
int n = copies.getValue();
```

The classes

```
TextSyntax
DateTimeSyntax
URISyntax
```

encapsulate a string, a date and time value, or a URI.

Finally, many attributes can take a finite number of values. For example, the `PrintQuality` attribute has three settings: draft, normal, and high. They are represented by three constants:

```
PrintQuality.DRAFT
PrintQuality.NORMAL
PrintQuality.HIGH
```

Attribute classes with a finite number of values extend the `EnumSyntax` class, which provides a number of convenience methods to set up these enumerations in a typesafe manner. You need not worry about the mechanism when using such an attribute. Simply add the named values to attribute sets:

```
attributes.add(PrintQuality.HIGH);
```

Here is how you check the value of an attribute:

```
if (attributes.get(PrintQuality.class) == PrintQuality.HIGH)
 . . .
```

Table 11.4 lists the printing attributes. The second column lists the superclass of the attribute class (for example, `IntegerSyntax` for the `Copies` attribute) or the set of enumeration values for the attributes with a finite set of values. The last four columns indicate whether the attribute class implements the `DocAttribute` (DA), `PrintJobAttribute` (PJA), `PrintRequestAttribute` (PRA), and `PrintServiceAttribute` (PSA) interfaces.

**Table 11.4** Printing Attributes

| Attribute               | Superclass or Enumeration Constants                                                       | DA | PJA | PRA | PSA |
|-------------------------|-------------------------------------------------------------------------------------------|----|-----|-----|-----|
| Chromaticity            | MONOCHROME, COLOR                                                                         | ✓  | ✓   | ✓   |     |
| ColorSupported          | SUPPORTED, NOT_SUPPORTED                                                                  |    |     | ✓   |     |
| Compression             | COMPRESS, DEFLATE, GZIP, NONE                                                             |    | ✓   |     |     |
| Copies                  | IntegerSyntax                                                                             |    | ✓   | ✓   |     |
| DateTimeAtCompleted     | DateTimeSyntax                                                                            |    | ✓   |     |     |
| DateTimeAtCreation      | DateTimeSyntax                                                                            |    | ✓   |     |     |
| DateTimeAtProcessing    | DateTimeSyntax                                                                            |    | ✓   |     |     |
| Destination             | URISyntax                                                                                 |    | ✓   | ✓   |     |
| DocumentName            | TextSyntax                                                                                |    | ✓   |     |     |
| Fidelity                | FIDELITY_TRUE, FIDELITY_FALSE                                                             |    | ✓   | ✓   |     |
| Finishes                | NONE, STAPLE, EDGE_STITCH, BIND,<br>SADDLE_STITCH, COVER, ...                             | ✓  | ✓   | ✓   |     |
| JobHoldUntil            | DateTimeSyntax                                                                            |    | ✓   | ✓   |     |
| JobImpressions          | IntegerSyntax                                                                             |    | ✓   | ✓   |     |
| JobImpressionsCompleted | IntegerSyntax                                                                             |    | ✓   |     |     |
| JobKOctets              | IntegerSyntax                                                                             |    | ✓   | ✓   |     |
| JobKOctetsProcessed     | IntegerSyntax                                                                             |    | ✓   |     |     |
| JobMediaSheets          | IntegerSyntax                                                                             |    | ✓   | ✓   |     |
| JobMediaSheetsCompleted | IntegerSyntax                                                                             |    | ✓   |     |     |
| JobMessageFromOperator  | TextSyntax                                                                                |    | ✓   |     |     |
| JobName                 | TextSyntax                                                                                |    | ✓   | ✓   |     |
| JobOriginatingUserName  | TextSyntax                                                                                |    | ✓   |     |     |
| JobPriority             | IntegerSyntax                                                                             |    | ✓   | ✓   |     |
| JobSheets               | STANDARD, NONE                                                                            |    | ✓   | ✓   |     |
| JobState                | ABORTED, CANCELED, COMPLETED, PENDING,<br>PENDING_HELD, PROCESSING,<br>PROCESSING_STOPPED |    | ✓   |     |     |

(Continues)

**Table 11.4** (Continued)

| Attribute                | Superclass or Enumeration Constants                                                                                                     | DA | PJA | PRA | PSA |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|----|-----|-----|-----|
| JobStateReason           | ABORTED_BY_SYSTEM,<br>DOCUMENT_FORMAT_ERROR, many others                                                                                |    |     |     |     |
| JobStateReasons          | HashSet                                                                                                                                 | ✓  |     |     |     |
| MediaName                | ISO_A4_WHITE, ISO_A4_TRANSPARENT,<br>NA LETTER WHITE, NA LETTER TRANSPARENT                                                             | ✓  | ✓   | ✓   |     |
| MediaSize                | ISO_A0–ISO_A10, ISO_B0–ISO_B10,<br>ISO_C0–ISO_C10, NA LETTER, NA LEGAL,<br>various other paper and<br>envelope sizes                    |    |     |     |     |
| MediaSizeName            | ISO_A0–ISO_A10, ISO_B0–ISO_B10,<br>ISO_C0–ISO_C10, NA LETTER, NA LEGAL,<br>various other paper and<br>envelope size names               | ✓  | ✓   | ✓   |     |
| MediaTray                | TOP, MIDDLE, BOTTOM, SIDE, ENVELOPE,<br>LARGE_CAPACITY, MAIN, MANUAL                                                                    | ✓  | ✓   | ✓   |     |
| MultipleDocumentHandling | SINGLE_DOCUMENT,<br>SINGLE_DOCUMENT_NEW_SHEET,<br>SEPARATE_DOCUMENTS_COLLATED_COPIES,<br>SEPARATE_DOCUMENTS_UNCOLLATED_COPIES           |    | ✓   | ✓   |     |
| NumberOfDocuments        | IntegerSyntax                                                                                                                           | ✓  |     |     |     |
| NumberOfInterveningJobs  | IntegerSyntax                                                                                                                           | ✓  |     |     |     |
| NumberUp                 | IntegerSyntax                                                                                                                           | ✓  | ✓   | ✓   |     |
| OrientationRequested     | PORTRAIT, LANDSCAPE, REVERSE_PORTRAIT,<br>REVERSE_LANDSCAPE                                                                             | ✓  | ✓   | ✓   |     |
| OutputDeviceAssigned     | TextSyntax                                                                                                                              | ✓  |     |     |     |
| PageRanges               | SetOfInteger                                                                                                                            | ✓  | ✓   | ✓   |     |
| PagesPerMinute           | IntegerSyntax                                                                                                                           |    |     |     | ✓   |
| PagesPerMinuteColor      | IntegerSyntax                                                                                                                           |    |     |     | ✓   |
| PDLOverrideSupported     | ATTEMPTED, NOT_ATTEMPTED                                                                                                                |    |     |     | ✓   |
| PresentationDirection    | TORIGHT_TOBOTTOM, TORIGHT_TOTOP,<br>TOBOTTOM_TORIGHT, TOBOTTOM_TOLEFT,<br>TOLEFT_TOBOTTOM, TOLEFT_TOTOP,<br>TOTOP_TORIGHT, TOTOP_TOLEFT | ✓  | ✓   |     |     |

(Continues)

**Table 11.4 (Continued)**

| Attribute                    | Superclass or Enumeration Constants                                              | DA | PJA | PRA | PSA |
|------------------------------|----------------------------------------------------------------------------------|----|-----|-----|-----|
| PrinterInfo                  | TextSyntax                                                                       |    |     |     | ✓   |
| PrinterIsAcceptingJobs       | ACCEPTING_JOBS, NOT_ACCEPTING_JOBS                                               |    |     |     | ✓   |
| PrinterLocation              | TextSyntax                                                                       |    |     |     | ✓   |
| PrinterMakeAndModel          | TextSyntax                                                                       |    |     |     | ✓   |
| PrinterMessageFromOperator   | TextSyntax                                                                       |    |     |     | ✓   |
| PrinterMoreInfo              | URISyntax                                                                        |    |     |     | ✓   |
| PrinterMoreInfoManufacturer  | URISyntax                                                                        |    |     |     | ✓   |
| PrinterName                  | TextSyntax                                                                       |    |     |     | ✓   |
| PrinterResolution            | ResolutionSyntax                                                                 | ✓  | ✓   | ✓   |     |
| PrinterState                 | PROCESSING, IDLE, STOPPED, UNKNOWN                                               |    |     |     | ✓   |
| PrinterStateReason           | COVER_OPEN, FUSER_OVER_TEMP, MEDIA_JAM,<br>and many others                       |    |     |     |     |
| PrinterStateReasons          | HashMap                                                                          |    |     |     |     |
| PrinterURI                   | URISyntax                                                                        |    |     |     | ✓   |
| PrintQuality                 | DRAFT, NORMAL, HIGH                                                              | ✓  | ✓   | ✓   |     |
| QueuedJobCount               | IntegerSyntax                                                                    |    |     |     | ✓   |
| ReferenceUriSchemesSupported | FILE, FTP, GOPHER, HTTP, HTTPS, NEWS,<br>NNTP, WAIS                              |    |     |     |     |
| RequestingUserName           | TextSyntax                                                                       |    |     |     | ✓   |
| Severity                     | ERROR, REPORT, WARNING                                                           |    |     |     |     |
| SheetCollate                 | COLLATED, UNCOLLATED                                                             | ✓  | ✓   | ✓   |     |
| Sides                        | ONE_SIDED, DUPLEX (=<br>TWO_SIDED_LONG_EDGE), TUMBLE (=<br>TWO_SIDED_SHORT_EDGE) | ✓  | ✓   | ✓   |     |

**NOTE:** As you can see, there are lots of attributes, many of which are quite specialized. The source for most of the attributes is the Internet Printing Protocol 1.1 (RFC 2911).

---

**NOTE:** An earlier version of the printing API introduced the `JobAttributes` and `PageAttributes` classes, whose purpose was similar to the printing attributes covered in this section. These classes are now obsolete.

---

**`javax.print.attribute.Attribute` 1.4**

- `Class getCategory()`  
gets the category of this attribute.
- `String getName()`  
gets the name of this attribute.

**`javax.print.attribute.AttributeSet` 1.4**

- `boolean add(Attribute attr)`  
adds an attribute to this set. If the set has another attribute with the same category, that attribute is replaced by the given attribute. Returns `true` if the set changed as a result of this operation.
- `Attribute get(Class category)`  
retrieves the attribute with the given category key, or `null` if no such attribute exists.
- `boolean remove(Attribute attr)`
- `boolean remove(Class category)`  
removes the given attribute, or the attribute with the given category, from the set. Returns `true` if the set changed as a result of this operation.
- `Attribute[] toArray()`  
returns an array with all attributes in this set.

**`javax.print.PrintService` 1.4**

- `PrintServiceAttributeSet getAttributes()`  
gets the attributes of this print service.

**`javax.print.DocPrintJob` 1.4**

- `PrintJobAttributeSet getAttributes()`  
gets the attributes of this print job.

This concludes our discussion of printing. You now know how to print 2D graphics and other document types, how to enumerate printers and stream print services, and how to set and retrieve attributes. Next, we turn to two important user interface issues: the clipboard and the drag-and-drop mechanism.

## 11.13 The Clipboard

One of the most useful and convenient user interface mechanisms of GUI environments (such as Windows and the X Window System) is *cut and paste*. You select some data in one program and cut or copy them to the clipboard. Then, you switch to another program and paste the clipboard contents into that application. Using the clipboard, you can transfer text, images, or other data from one document to another or, of course, from one place in a document to another in the same document. Cut and paste is so natural that most computer users never think about it.

Even though the clipboard is conceptually simple, implementing clipboard services is actually harder than you might think. Suppose you copy text from a word processor to the clipboard. If you paste that text into another word processor, you expect the fonts and formatting to stay intact. That is, the text in the clipboard needs to retain the formatting information. However, if you paste the text into a plain text field, you expect that just the characters are pasted in, without additional formatting codes. To support this flexibility, the data provider must be able offer the clipboard data in multiple formats, so the data consumer can pick one of them.

The system clipboard implementations of Microsoft Windows and the Macintosh are similar, but, of course, there are slight differences. However, the X Window System clipboard mechanism is much more limited—cutting and pasting of anything but plain text is only sporadically supported. You should consider these limitations when trying out the programs in this section.

---

**NOTE:** Check out the file `jre/lib/flavormap.properties` on your platform to get an idea of what kinds of objects can be transferred between Java programs and the system clipboard.

---

Often, programs need to support cut and paste of data types that the system clipboard cannot handle. The data transfer API supports the transfer of arbitrary local object references in the same virtual machine. Between different virtual machines, you can transfer serialized objects and references to remote objects.

Table 11.5 summarizes the data transfer capabilities of the clipboard mechanism.

**Table 11.5** Capabilities of the Java Data Transfer Mechanism

| Transfer                                    | Format                                                           |
|---------------------------------------------|------------------------------------------------------------------|
| Between a Java program and a native program | Text, images, file lists, . . . (depending on the host platform) |
| Between two cooperating Java programs       | Serialized and remote objects                                    |
| Within one Java program                     | Any object                                                       |

### 11.13.1 Classes and Interfaces for Data Transfer

Data transfer in Java is implemented in a package called `java.awt.datatransfer`. Here is an overview of the most important classes and interfaces of that package:

- Objects that can be transferred via a clipboard must implement the `Transferable` interface.
- The `Clipboard` class describes a clipboard. Transferable objects are the only items that can be put on or taken off a clipboard. The system clipboard is a concrete example of a `Clipboard`.
- The `DataFlavor` class describes data flavors that can be placed on the clipboard.
- The `StringSelection` class is a concrete class that implements the `Transferable` interface. It transfers text strings.
- A class must implement the `ClipboardOwner` interface if it wants to be notified when the clipboard contents have been overwritten by someone else. Clipboard ownership enables “delayed formatting” of complex data. If a program transfers simple data (such as a string), it simply sets the clipboard contents and moves on. However, if a program places onto the clipboard complex data that can be formatted in multiple flavors, then it might not actually want to prepare all the flavors, because there is a good chance that most of them will never be needed. However, it then needs to hang on to the clipboard data so it can create the flavors later when they are requested. The clipboard owner is notified (by a call to its `lostOwnership` method) when the contents of the clipboard change. That tells it that the information is no longer needed. In our sample programs, we don’t worry about clipboard ownership.

### 11.13.2 Transferring Text

The best way to get comfortable with the data transfer classes is to start with the simplest situation: transferring text to and from the system clipboard. First, get a reference to the system clipboard:

```
Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
```

For strings to be transferred to the clipboard, they must be wrapped into `StringSelection` objects.

```
String text = . . .;
StringSelection selection = new StringSelection(text);
```

The actual transfer is done by a call to `setContents`, which takes a `StringSelection` object and a `ClipBoardOwner` as parameters. If you are not interested in designating a clipboard owner, set the second parameter to `null`.

```
clipboard.setContents(selection, null);
```

Here is the reverse operation—reading a string from the clipboard:

```
DataFlavor flavor = DataFlavor.stringFlavor;
if (clipboard.isDataFlavorAvailable(flavor))
 String text = (String) clipboard.getData(flavor);
```

Listing 11.17 is a program that demonstrates cutting and pasting between a Java application and the system clipboard. If you select some text in the text area and click Copy, the selection is copied to the system clipboard. You can then paste it into any text editor (see Figure 11.40). Conversely, when you copy text from the text editor, you can paste it into our sample program.

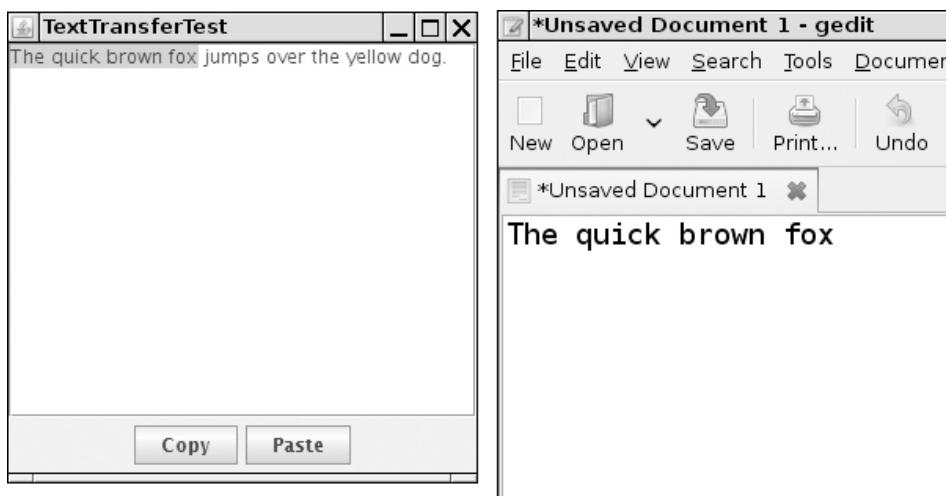


Figure 11.40 The TextTransferTest program

**Listing 11.17** transferText/TextTransferFrame.java

```
1 package transferText;
2
3 import java.awt.*;
4 import java.awt.datatransfer.*;
5 import java.awt.event.*;
6 import java.io.*;
7 import javax.swing.*;
8
9 /**
10 * This frame has a text area and buttons for copying and pasting text.
11 */
12 public class TextTransferFrame extends JFrame
13 {
14 private JTextArea textArea;
15 private static final int TEXT_ROWS = 20;
16 private static final int TEXT_COLUMNS = 60;
17
18 public TextTransferFrame()
19 {
20 textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
21 add(new JScrollPane(textArea), BorderLayout.CENTER);
22 JPanel panel = new JPanel();
23
24 JButton copyButton = new JButton("Copy");
25 panel.add(copyButton);
26 copyButton.addActionListener(event -> copy());
27
28 JButton pasteButton = new JButton("Paste");
29 panel.add(pasteButton);
30 pasteButton.addActionListener(event -> paste());
31
32 add(panel, BorderLayout.SOUTH);
33 pack();
34 }
35
36 /**
37 * Copies the selected text to the system clipboard.
38 */
39 private void copy()
40 {
41 Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
42 String text = textArea.getSelectedText();
43 if (text == null) text = textArea.getText();
44 StringSelection selection = new StringSelection(text);
45 clipboard.setContents(selection, null);
46 }
47
```

```
48 /**
49 * Pastes the text from the system clipboard into the text area.
50 */
51 private void paste()
52 {
53 Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
54 DataFlavor flavor = DataFlavor.stringFlavor;
55 if (clipboard.isDataFlavorAvailable(flavor))
56 {
57 try
58 {
59 String text = (String) clipboard.getData(flavor);
60 textArea.replaceSelection(text);
61 }
62 catch (UnsupportedFlavorException e | IOException ex)
63 {
64 JOptionPane.showMessageDialog(this, ex);
65 }
66 }
67 }
68 }
```

**java.awt.Toolkit 1.0**

- **Clipboard getSystemClipboard() 1.1**  
gets the system clipboard.

**java.awt.datatransfer.Clipboard 1.1**

- **Transferable getContents(Object requester)**  
gets the clipboard contents.  
*Parameters:* requester The object requesting the clipboard contents; this value is not actually used.
- **void setContents(Transferable contents, ClipboardOwner owner)**  
puts contents on the clipboard.  
*Parameters:* contents The Transferable encapsulating the contents  
owner The object to be notified (via its lostOwnership method) when new information is placed on the clipboard, or null if no notification is desired
- **boolean isDataFlavorAvailable(DataFlavor flavor) 5.0**  
returns true if the clipboard has data in the given flavor.

*(Continues)*

***java.awt.datatransfer.Clipboard 1.1 (Continued)***

- **Object getData(DataFlavor flavor) 5.0**

gets the data in the given flavor, or throws an `UnsupportedFlavorException` if no data are available in the given flavor.

***java.awt.datatransfer.ClipboardOwner 1.1***

- **void lostOwnership(Clipboard clipboard, Transferable contents)**

notifies this object that it is no longer the owner of the contents of the clipboard.

*Parameters:*      clipboard      The clipboard onto which the contents were placed  
                      contents      The item that this owner had placed onto the clipboard

***java.awt.datatransfer.Transferable 1.1***

- **boolean isDataFlavorSupported(DataFlavor flavor)**

returns true if the specified flavor is one of the supported data flavors, false otherwise.

- **Object getTransferData(DataFlavor flavor)**

returns the data, formatted in the requested flavor. Throws an `UnsupportedFlavorException` if the flavor requested is not supported.

### 11.13.3 The Transferable Interface and Data Flavors

A `DataFlavor` is defined by two characteristics:

- A MIME type name (such as "image/gif")
- A representation class for accessing the data (such as `java.awt.Image`)

In addition, every data flavor has a human-readable name (such as "GIF Image").

The representation class can be specified with a `class` parameter in the MIME type, for example:

```
image/gif;class=java.awt.Image
```

---

**NOTE:** This is just an example to show the syntax. There is no standard data flavor for transferring GIF image data.

---

If no class parameter is given, the representation class is `InputStream`.

Three MIME types are defined for transferring local, serialized, and remote Java objects:

```
application/x-java-jvm-local-objectref
application/x-java-serialized-object
application/x-java-remote-object
```

---

**NOTE:** The x- prefix indicates that this is an experimental name and not one that is sanctioned by IANA, the organization that assigns standard MIME type names.

---

For example, the standard `StringFlavor` data flavor is described by the MIME type

```
application/x-java-serialized-object;class=java.lang.String
```

You can ask the clipboard to list all available flavors:

```
DataFlavor[] flavors = clipboard.getAvailableDataFlavors();
```

You can also install a `FlavorListener` onto the clipboard. The listener is notified when the collection of data flavors on the clipboard changes. See the API notes for details.

#### java.awt.datatransfer.DataFlavor 1.1

- `DataFlavor(String mimeType, String humanPresentableName)`

creates a data flavor that describes stream data in a format described by a MIME type.

*Parameters:*      `mimeType`                          A MIME type string  
                      `humanPresentableName`              A more readable version of the name

- `DataFlavor(Class class, String humanPresentableName)`

creates a data flavor that describes a Java platform class. Its MIME type is `application/x-java-serialized-object;class=className`.

*Parameters:*      `class`                              The class that is retrieved from the Transferable  
                      `humanPresentableName`              A readable version of the name

- `String getMimeType()`

returns the MIME type string for this data flavor.

(Continues)

***java.awt.datatransfer.DataFlavor 1.1 (Continued)***

- `boolean isMimeTypeEqual(String mimeType)`  
tests whether this data flavor has the given MIME type.
- `String getHumanPresentableName()`  
returns the human-presentable name for the data format of this data flavor.
- `Class getRepresentationClass()`  
returns a `Class` object that represents the class of the object that a `Transferable` object will return when called with this data flavor. This is either the `class` parameter of the MIME type or `InputStream`.

***java.awt.datatransfer.Clipboard 1.1***

- `DataFlavor[] getAvailableDataFlavors() 5.0`  
returns an array of the available flavors.
- `void addFlavorListener(FlavorListener listener) 5.0`  
adds a listener that is notified when the set of available flavors changes.

***java.awt.datatransfer.Transferable 1.1***

- `DataFlavor[] getTransferDataFlavors()`  
returns an array of the supported flavors.

***java.awt.datatransfer.FlavorListener 5.0***

- `void flavorsChanged(FlavorEvent event)`  
is called when a clipboard's set of available flavors changes.

#### 11.13.4 Building an Image Transferable

Objects that you want to transfer via the clipboard must implement the `Transferable` interface. The `StringSelection` class is currently the only public class in the Java standard library that implements the `Transferable` interface. In this section, you will see how to transfer images into the clipboard. Since Java does not supply a class for image transfer, you must implement it yourself.

The class is completely trivial. It simply reports that the only available data format is `DataFlavor.imageFlavor`, and it holds an `image` object.

```
class ImageTransferable implements Transferable
{
 private Image theImage;

 public ImageTransferable(Image image)
 {
 theImage = image;
 }

 public DataFlavor[] getTransferDataFlavors()
 {
 return new DataFlavor[] { DataFlavor.imageFlavor };
 }

 public boolean isDataFlavorSupported(DataFlavor flavor)
 {
 return flavor.equals(DataFlavor.imageFlavor);
 }

 public Object getTransferData(DataFlavor flavor)
 throws UnsupportedFlavorException
 {
 if(flavor.equals(DataFlavor.imageFlavor))
 {
 return theImage;
 }
 else
 {
 throw new UnsupportedFlavorException(flavor);
 }
 }
}
```

---

**NOTE:** Java SE supplies the `DataFlavor.imageFlavor` constant and does all the heavy lifting to convert between Java images and native clipboard images. Curiously, however, it does not supply the wrapper class that is necessary to place images onto the clipboard.

---

The program in Listing 11.18 demonstrates the transfer of images between a Java application and the system clipboard. When the program starts, it generates an image containing a red circle. Click the Copy button to copy the image to the clipboard and then paste it into another application (see Figure 11.41). From another application, copy an image into the system clipboard. Then click the Paste button and see the image being pasted into the example program (see Figure 11.42).

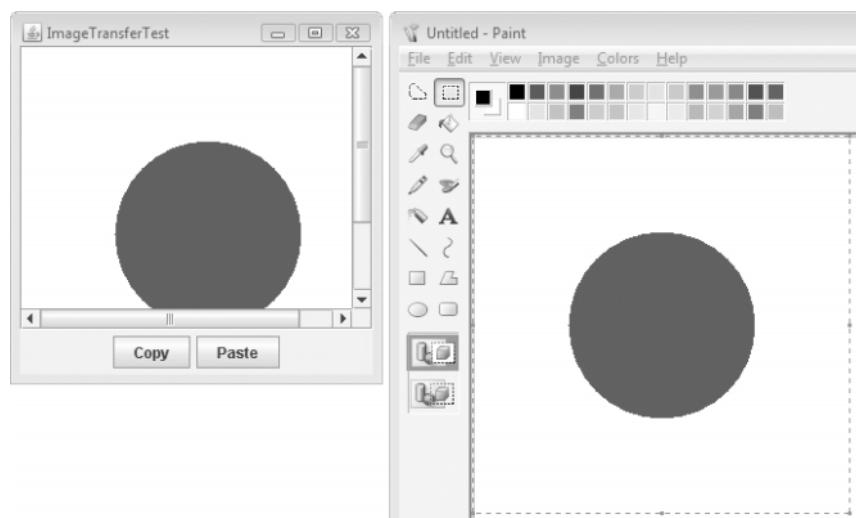


Figure 11.41 Copying from a Java program to a native program

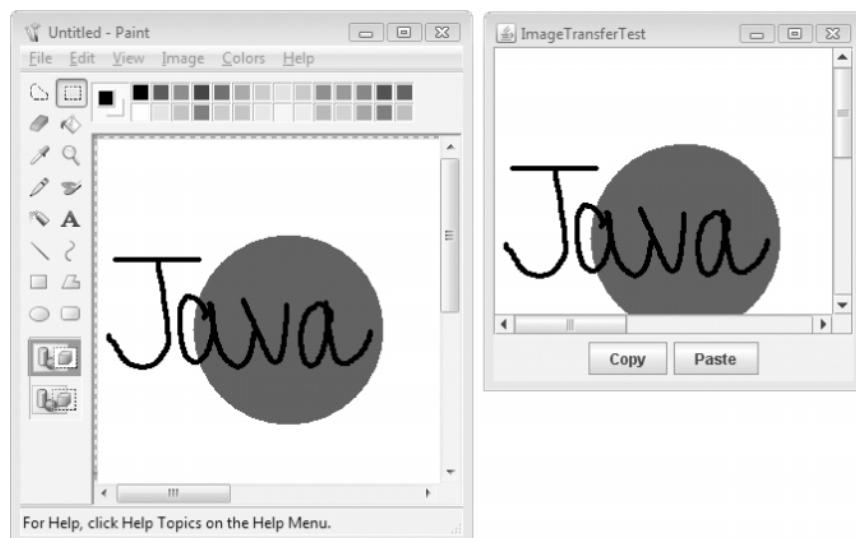


Figure 11.42 Copying from a native program to a Java program

The program is a straightforward modification of the text transfer program. The data flavor is now `DataFlavor.imageFlavor`, and we use the `ImageTransferable` class to transfer an image to the system clipboard.

**Listing 11.18** imageTransfer/ImageTransferFrame.java

```
1 package imageTransfer;
2
3 import java.awt.*;
4 import java.awt.datatransfer.*;
5 import java.awt.image.*;
6 import java.io.*;
7
8 import javax.swing.*;
9
10 /**
11 * This frame has an image label and buttons for copying and pasting an image.
12 */
13 class ImageTransferFrame extends JFrame
14 {
15 private JLabel label;
16 private Image image;
17 private static final int IMAGE_WIDTH = 300;
18 private static final int IMAGE_HEIGHT = 300;
19
20 public ImageTransferFrame()
21 {
22 label = new JLabel();
23 image = new BufferedImage(IMAGE_WIDTH, IMAGE_HEIGHT, BufferedImage.TYPE_INT_ARGB);
24 Graphics g = image.getGraphics();
25 g.setColor(Color.WHITE);
26 g.fillRect(0, 0, IMAGE_WIDTH, IMAGE_HEIGHT);
27 g.setColor(Color.RED);
28 g.fillOval(IMAGE_WIDTH / 4, IMAGE_WIDTH / 4, IMAGE_WIDTH / 2, IMAGE_HEIGHT / 2);
29
30 label.setIcon(new ImageIcon(image));
31 add(new JScrollPane(label), BorderLayout.CENTER);
32 JPanel panel = new JPanel();
33
34 JButton copyButton = new JButton("Copy");
35 panel.add(copyButton);
36 copyButton.addActionListener(event -> copy());
37
38 JButton pasteButton = new JButton("Paste");
39 panel.add(pasteButton);
40 pasteButton.addActionListener(event -> paste());
41
42 add(panel, BorderLayout.SOUTH);
43 pack();
44 }
45 }
```

*(Continues)*

**Listing 11.18 (Continued)**

```
46 /**
47 * Copies the current image to the system clipboard.
48 */
49 private void copy()
50 {
51 Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
52 ImageTransferable selection = new ImageTransferable(image);
53 clipboard.setContents(selection, null);
54 }
55 /**
56 * Pastes the image from the system clipboard into the image label.
57 */
58 private void paste()
59 {
60 Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
61 DataFlavor flavor = DataFlavor.imageFlavor;
62 if (clipboard.isDataFlavorAvailable(flavor))
63 {
64 try
65 {
66 image = (Image) clipboard.getData(flavor);
67 label.setIcon(new ImageIcon(image));
68 }
69 catch (UnsupportedFlavorException | IOException ex)
70 {
71 JOptionPane.showMessageDialog(this, ex);
72 }
73 }
74 }
75 }
76 }
```

---

### 11.13.5 Transferring Java Objects via the System Clipboard

Suppose you want to copy and paste objects from one Java application to another. You can accomplish this task by placing serialized Java objects onto the system clipboard.

The program in Listing 11.19 demonstrates this capability. The program shows a color chooser. The Copy button copies the current color to the system clipboard as a serialized `Color` object. The Paste button checks whether the system clipboard contains a serialized `Color` object. If so, it fetches the color and sets it as the current choice of the color chooser.

You can transfer the serialized object between two Java applications (see Figure 11.43). Run two copies of the `SerialTransferTest` program. Click Copy in the

first program, then click Paste in the second program. The `Color` object is transferred from one virtual machine to the other.

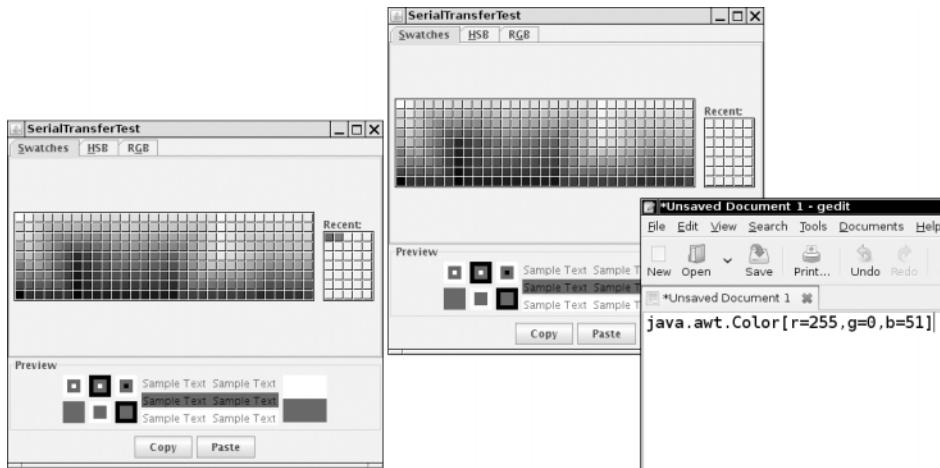


Figure 11.43 Data are copied between two instances of a Java application.

To enable data transfer, the Java platform places the binary data of the serialized object on the system clipboard. Another Java program—not necessarily of the same type as the one that generated the clipboard data—can retrieve the clipboard data and deserialize the object.

Of course, a non-Java application will not know what to do with the clipboard data. For that reason, the example program offers the clipboard data in a second flavor—as text. The text is simply the result of the `toString` method, applied to the transferred object. To see the second flavor, run the program, click on a color, and then select the Paste command in your text editor. A string such as

```
java.awt.Color[r=255,g=0,b=51]
```

will be inserted into your document.

Essentially no additional programming is required to transfer a serializable object. Use the MIME type

```
application/x-java-serialized-object;class=className
```

As before, you have to build your own transfer wrapper—see the example code for details.

**Listing 11.19** serialTransfer/SerialTransferFrame.java

```
1 package serialTransfer;
2
3 import java.awt.*;
4 import java.awt.datatransfer.*;
5 import java.awt.event.*;
6 import java.io.*;
7 import javax.swing.*;
8
9 /**
10 * This frame contains a color chooser, and copy and paste buttons.
11 */
12 class SerialTransferFrame extends JFrame
13 {
14 private JColorChooser chooser;
15
16 public SerialTransferFrame()
17 {
18 chooser = new JColorChooser();
19 add(chooser, BorderLayout.CENTER);
20 JPanel panel = new JPanel();
21
22 JButton copyButton = new JButton("Copy");
23 panel.add(copyButton);
24 copyButton.addActionListener(event -> copy());
25
26 JButton pasteButton = new JButton("Paste");
27 panel.add(pasteButton);
28 pasteButton.addActionListener(event -> paste());
29
30 add(panel, BorderLayout.SOUTH);
31 pack();
32 }
33
34 /**
35 * Copies the chooser's color into the system clipboard.
36 */
37 private void copy()
38 {
39 Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
40 Color color = chooser.getColor();
41 Serializable selection = new Serializable(color);
42 clipboard.setContents(selection, null);
43 }
44
45 /**
46 * Pastes the color from the system clipboard into the chooser.
47 */
```

```
48 private void paste()
49 {
50 Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
51 try
52 {
53 DataFlavor flavor = new DataFlavor(
54 "application/x-java-serialized-object;class=java.awt.Color");
55 if (clipboard.isDataFlavorAvailable(flavor))
56 {
57 Color color = (Color) clipboard.getData(flavor);
58 chooser.setColor(color);
59 }
60 }
61 catch (ClassNotFoundException | UnsupportedFlavorException | IOException ex)
62 {
63 JOptionPane.showMessageDialog(this, ex);
64 }
65 }
66 }
67 /**
68 * This class is a wrapper for the data transfer of serialized objects.
69 */
70 class SerialTransferable implements Transferable
71 {
72 private Serializable obj;
73
74 /**
75 * Constructs the selection.
76 * @param o any serializable object
77 */
78 SerialTransferable(Serializable o)
79 {
80 obj = o;
81 }
82
83 public DataFlavor[] getTransferDataFlavors()
84 {
85 DataFlavor[] flavors = new DataFlavor[2];
86 Class<?> type = obj.getClass();
87 String mimeType = "application/x-java-serialized-object;class=" + type.getName();
88 try
89 {
90 flavors[0] = new DataFlavor(mimeType);
91 flavors[1] = DataFlavor.stringFlavor;
92 return flavors;
93 }
94 }
```

(Continues)

**Listing 11.19 (Continued)**

```
95 catch (ClassNotFoundException e)
96 {
97 return new DataFlavor[0];
98 }
99 }
100
101 public boolean isDataFlavorSupported(DataFlavor flavor)
102 {
103 return DataFlavor.stringFlavor.equals(flavor)
104 || "application".equals(flavor.getPrimaryType())
105 && "x-java-serialized-object".equals(flavor.getSubType())
106 && flavor.getRepresentationClass().isAssignableFrom(obj.getClass());
107 }
108
109 public Object getTransferData(DataFlavor flavor) throws UnsupportedFlavorException
110 {
111 if (!isDataFlavorSupported(flavor)) throw new UnsupportedFlavorException(flavor);
112
113 if (DataFlavor.stringFlavor.equals(flavor)) return obj.toString();
114
115 return obj;
116 }
117 }
```

---

### 11.13.6 Using a Local Clipboard to Transfer Object References

Occasionally, you might need to copy and paste a data type that isn't one of the data types supported by the system clipboard and that isn't serializable. To transfer an arbitrary Java object reference within the same JVM, use the MIME type

*application/x-java-jvm-local-objectref;class=className*

You need to define a `Transferable` wrapper for this type. The process is entirely analogous to the `SerialTransferable` wrapper of the preceding example.

An object reference is only meaningful within a single virtual machine. For that reason, you cannot copy the shape object to the system clipboard. Instead, use a local clipboard:

```
Clipboard clipboard = new Clipboard("local");
```

The construction parameter is the clipboard name.

However, using a local clipboard has one major disadvantage. You need to synchronize the local and the system clipboard, so that users don't confuse the two. Currently, the Java platform doesn't do that synchronization for you.

**java.awt.datatransfer.Clipboard 1.1**

- `Clipboard(String name)`  
constructs a local clipboard with the given name.

## 11.14 Drag and Drop

When you use cut and paste to transmit information between two programs, the clipboard acts as an intermediary. The *drag and drop* metaphor cuts out the middleman and lets two programs communicate directly. The Java platform offers basic support for drag and drop. You can carry out drag and drop operations between Java applications and native applications. This section shows you how to write a Java application that is a drop target, and an application that is a drag source.

Before going deeper into the Java platform support for drag and drop, let us quickly look at the drag-and-drop user interface. We use the Windows Explorer and WordPad programs as examples—on another platform, you can experiment with locally available programs with drag-and-drop capabilities.

You initiate a *drag operation* with a *gesture* inside a *drag source*—by first selecting one or more elements and then dragging the selection away from its initial location. When you release the mouse button over a drop target that accepts the drop operation, the drop target queries the drag source for information about the dropped elements and carries out an appropriate operation. For example, if you drop a file icon from a file manager on top of a directory icon, the file is moved into that directory. However, if you drag it to a text editor, the text editor opens the file. (This requires, of course, that you use a file manager and text editor that are capable of drag and drop, such as Explorer/WordPad in Windows or Nautilus/gedit in Gnome.)

If you hold down the Ctrl key while dragging, the type of the drop action changes from a *move action* to a *copy action*, and a copy of the file is placed into the directory. If you hold down both Shift and Ctrl keys, then a *link* to the file is placed into the directory. (Other platforms might use other keyboard combinations for these operations.)

Thus, there are three types of drop actions with different gestures:

- Move
- Copy
- Link

The intention of the link action is to establish a reference to the dropped element. Such links typically require support from the host operating system (such as symbolic links for files, or object links for document components) and don't usually make a lot of sense in cross-platform programs. In this section, we focus on using drag and drop for copying and moving.

There is usually some visual feedback for the drag operation. Minimally, the cursor shape changes. As the cursor moves over possible *drop targets*, the cursor shape indicates whether the drop is possible or not. If a drop is possible, the cursor shape also indicates the type of the drop action. Table 11.6 shows several drop cursor shapes.

**Table 11.6** Drop Cursor Shapes

| Action           | Windows Icon | Gnome Icon |
|------------------|--------------|------------|
| Move             |              |            |
| Copy             |              |            |
| Link             |              |            |
| Drop not allowed |              |            |

You can also drag other elements besides file icons. For example, you can select text in WordPad or gedit and drag it. Try dropping text fragments into willing drop targets and see how they react.

---

**NOTE:** This experiment shows a disadvantage of drag and drop as a user interface mechanism. It can be difficult for users to anticipate what they can drag, where they can drop it, and what happens when they do. Because the default "move" action can remove the original, many users are understandably cautious about experimenting with drag and drop.

---

### 11.14.1 Data Transfer Support in Swing

Starting with Java SE 1.4, several Swing components have built-in support for drag and drop (see Table 11.7). You can drag selected text from a number of components, and you can drop text into text components. For backward

compatibility, you must call the `setDragEnabled` method to activate dragging. Dropping is always enabled.

**Table 11.7** Data Transfer Support in Swing Components

| Component                             | Drag Source                                       | Drop Target                 |
|---------------------------------------|---------------------------------------------------|-----------------------------|
| JFileChooser                          | Exports file list                                 | N/A                         |
| JColorChooser                         | Exports color object                              | Accepts color objects       |
| JTextField<br>JFormattedTextField     | Exports selected text                             | Accepts text                |
| JPasswordField                        | N/A (for security)                                | Accepts text                |
| JTextArea<br>JTextPane<br>JEditorPane | Exports selected text                             | Accepts text and file lists |
| JList<br>JTable<br>JTree              | Exports text description of selection (copy only) | N/A                         |

**NOTE:** The `java.awt.dnd` package provides a lower-level drag-and-drop API that forms the basis for the Swing drag and drop. We do not discuss that API in this book.

The program in Listing 11.20 demonstrates the behavior. As you run the program, note these points:

- You can select multiple items in the list, table, or tree (see Listing 11.21) and drag them.
- Dragging items from the table is a bit awkward. You first select with the mouse, then let go of the mouse button, then click it again, and then you drag.
- When you drop the items in the text area, you can see how the dragged information is formatted. Table cells are separated by tabs, and each selected row is on a separate line (see Figure 11.44).
- You can only copy, not move, items from the list, table, tree, file chooser, or color chooser. Removing items from a list, table, or tree is not possible with all data models. You will see in the next section how to implement this capability when the data model is editable.
- You cannot drag into the list, table, tree, or file chooser.

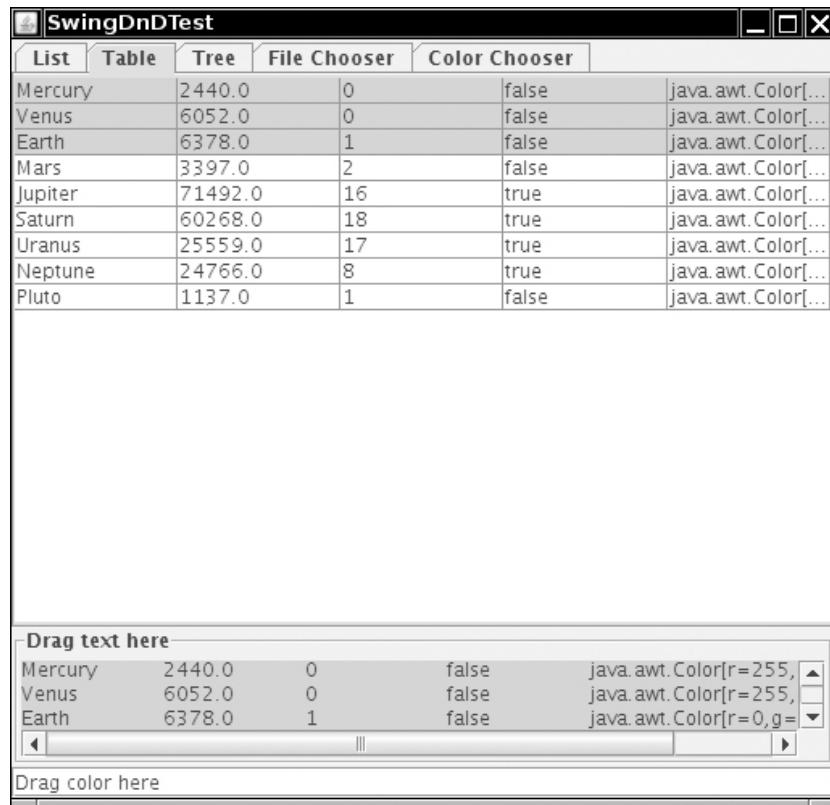


Figure 11.44 The Swing drag-and-drop test program

- If you run two copies of the program, you can drag a color from one color chooser to the other.
- You cannot drag text out of the text area because we didn't call `setDragEnabled` on it.

The Swing package provides a potentially useful mechanism to quickly turn a component into a drag source and drop target. You can install a *transfer handler* for a given property. For example, in our sample program, we call

```
textField.setTransferHandler(new TransferHandler("background"));
```

You can now drag a color into the text field, and its background color changes.

When a drop occurs, the transfer handler checks whether one of the data flavors has representation class `Color`. If so, it invokes the `setBackground` method.

By installing this transfer handler into the text field, you disable the standard transfer handler. You can no longer cut, copy, paste, drag, or drop text in the text field. However, you can now drag color out of this text field. You still need to select some text to initiate the drag gesture. When you drag the text, you'll find that you can drop it into the color chooser and change its color value to the text field's background color. However, you cannot drop the text into the text area.

---

**Listing 11.20** dnd/SwingDnDTest.java

---

```
1 package dnd;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7 * This program demonstrates the basic Swing support for drag and drop.
8 * @version 1.11 2016-05-10
9 * @author Cay Horstmann
10 */
11 public class SwingDnDTest
12 {
13 public static void main(String[] args)
14 {
15 EventQueue.invokeLater(() ->
16 {
17 JFrame frame = new SwingDnDFrame();
18 frame.setTitle("SwingDnDTest");
19 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20 frame.setVisible(true);
21 });
22 }
23 }
```

---

**Listing 11.21** dnd/SampleComponents.java

---

```
1 package dnd;
2
3 import java.awt.*;
4
5 import javax.swing.*;
6 import javax.swing.tree.*;
7
8 public class SampleComponents
9 {
10 public static JTree tree()
11 {
```

(Continues)

**Listing 11.21 (Continued)**

```
12 DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
13 DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
14 root.add(country);
15 DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
16 country.add(state);
17 DefaultMutableTreeNode city = new DefaultMutableTreeNode("San Jose");
18 state.add(city);
19 city = new DefaultMutableTreeNode("Cupertino");
20 state.add(city);
21 state = new DefaultMutableTreeNode("Michigan");
22 country.add(state);
23 city = new DefaultMutableTreeNode("Ann Arbor");
24 state.add(city);
25 country = new DefaultMutableTreeNode("Germany");
26 root.add(country);
27 state = new DefaultMutableTreeNode("Schleswig-Holstein");
28 country.add(state);
29 city = new DefaultMutableTreeNode("Kiel");
30 state.add(city);
31 return new JTree(root);
32 }
33
34 public static JList<String> list()
35 {
36 String[] words = { "quick", "brown", "hungry", "wild", "silent", "huge", "private",
37 "abstract", "static", "final" };
38
39 DefaultListModel<String> model = new DefaultListModel<>();
40 for (String word : words)
41 model.addElement(word);
42 return new JList<>(model);
43 }
44
45 public static JTable table()
46 {
47 Object[][] cells = { { "Mercury", 2440.0, 0, false, Color.YELLOW },
48 { "Venus", 6052.0, 0, false, Color.YELLOW },
49 { "Earth", 6378.0, 1, false, Color.BLUE }, { "Mars", 3397.0, 2, false, Color.RED },
50 { "Jupiter", 71492.0, 16, true, Color.ORANGE },
51 { "Saturn", 60268.0, 18, true, Color.ORANGE },
52 { "Uranus", 25559.0, 17, true, Color.BLUE },
53 { "Neptune", 24766.0, 8, true, Color.BLUE },
54 { "Pluto", 1137.0, 1, false, Color.BLACK } };
55
56 String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color" };
57 return new JTable(cells, columnNames);
58 }
59 }
```

---

**javax.swing.JComponent 1.2**

- void setTransferHandler(TransferHandler handler) 1.4

sets a transfer handler to handle data transfer operations (cut, copy, paste, drag, drop).

**javax.swing.TransferHandler 1.4**

- TransferHandler(String propertyName)

constructs a transfer handler that reads or writes the JavaBeans component property with the given name when a data transfer operation is executed.

**javax.swing.JFileChooser 1.2****javax.swing.JColorChooser 1.2****javax.swing.text.JTextComponent 1.2****javax.swing.JList 1.2****javax.swing.JTable 1.2****javax.swing.JTree 1.2**

- void setDragEnabled(boolean b) 1.4

enables or disables dragging of data out of this component.

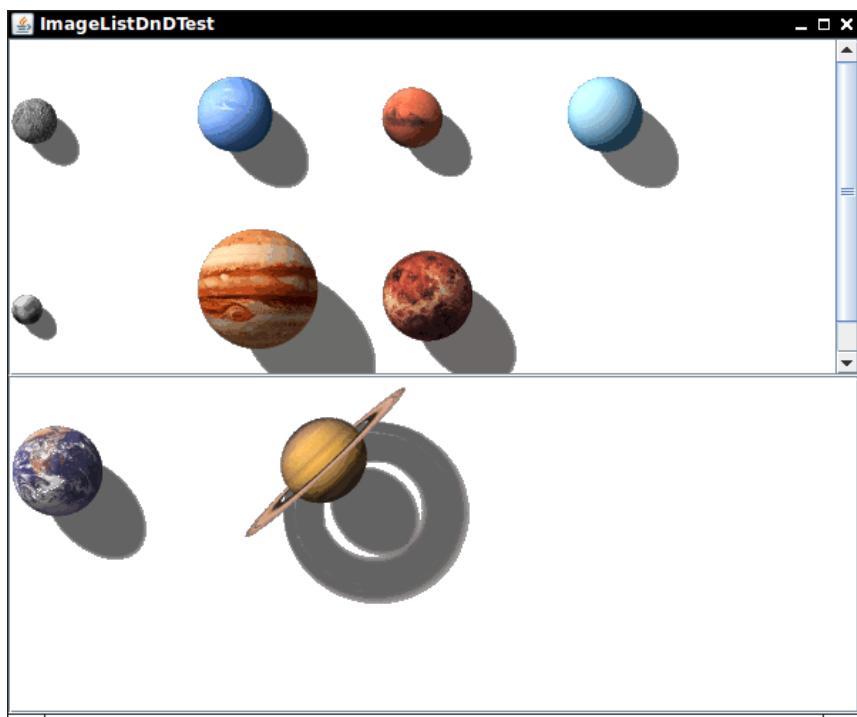
## 11.14.2 Drag Sources

In the previous section, you saw how to take advantage of the basic drag-and-drop support in Swing. In this section, we'll show you how to configure any component as a drag source. In the next section, we'll discuss drop targets and present a sample component that is both a source and a target for images.

To customize the drag-and-drop behavior of a Swing component, subclass the `TransferHandler` class. First, override the `getSourceActions` method to indicate which actions (copy, move, link) your component supports. Next, override the `createTransferable` method that produces a `Transferable` object, following the same process that you use for copying to the clipboard.

In our sample program, we drag images out of a `JList` that is filled with image icons (see Figure 11.45). Here is the implementation of the `createTransferable` method. The selected image is simply placed into an `ImageTransferable` wrapper.

```
protected Transferable createTransferable(JComponent source)
{
 JList list = (JList) source;
 int index = list.getSelectedIndex();
 if (index < 0) return null;
 ImageIcon icon = (ImageIcon) list.getModel().getElementAt(index);
 return new ImageTransferable(icon.getImage());
}
```



**Figure 11.45** The `ImageList` drag-and-drop application

In our example, we are fortunate that a `JList` is already wired for initiating a drag gesture. You simply activate that mechanism by calling the `setDragEnabled` method. If you add drag support to a component that does not recognize a drag gesture, you need to initiate the transfer yourself. For example, here is how you can initiate dragging on a `JLabel`:

```
label.addMouseListener(new MouseAdapter()
{
 public void mousePressed(MouseEvent evt)
 {
 int mode;
 if ((evt.getModifiers() & (InputEvent.CTRL_MASK | InputEvent.SHIFT_MASK)) != 0)
 mode = TransferHandler.COPY;
 else mode = TransferHandler.MOVE;
 JComponent comp = (JComponent) evt.getSource();
 TransferHandler th = comp.getTransferHandler();
 th.exportAsDrag(comp, evt, mode);
 }
});
```

Here, we simply start the transfer when the user clicks on the label. A more sophisticated implementation would watch for a mouse motion that drags the mouse by a small amount.

When the user completes the drop action, the `exportDone` method of the source transfer handler is invoked. In that method, you need to remove the transferred object if the user carried out a move action. Here is the implementation for the image list:

```
protected void exportDone(JComponent source, Transferable data, int action)
{
 if (action == MOVE)
 {
 JList list = (JList) source;
 int index = list.getSelectedIndex();
 if (index < 0) return;
 DefaultListModel model = (DefaultListModel) list.getModel();
 model.remove(index);
 }
}
```

To summarize, to turn a component into a drag source, you have to add a transfer handler that specifies the following:

- Which actions are supported
- Which data are transferred
- How the original data are removed after a move action

In addition, if your drag source is a component other than those listed in Table 11.7 on p. 905, you need to watch for a mouse gesture and initiate the transfer.

**javax.swing.TransferHandler 1.4**

- `int getSourceActions(JComponent c)`  
override to return the allowable source actions (bitwise or combination of COPY, MOVE, and LINK) when dragging from the given component.
- `protected Transferable createTransferable(JComponent source)`  
override to create the Transferable for the data that is to be dragged.
- `void exportAsDrag(JComponent comp, InputEvent e, int action)`  
starts a drag gesture from the given component. The action is COPY, MOVE, or LINK.
- `protected void exportDone(JComponent source, Transferable data, int action)`  
override to adjust the drag source after a successful transfer.

### 11.14.3 Drop Targets

In this section, we'll show you how to implement a drop target. Our example is again a `JList` with image icons. We'll add drop support so that users can drop images into the list.

To make a component into a drop target, set a `TransferHandler` and implement the `canImport` and `importData` methods.

---

**NOTE:** You can add a transfer handler to a `JFrame`. This is most commonly used for dropping files into an application. Valid drop locations include the frame decorations and the menu bar, but not components contained in the frame (which have their own transfer handlers).

---

The `canImport` method is called continuously as the user moves the mouse over the drop target component. Return `true` if a drop is allowed. This information affects the cursor icon that gives visual feedback whether the drop is allowed.

The `canImport` method has a parameter of type `TransferHandler.TransferSupport`. Through this parameter, you can obtain the drop action chosen by the user, the drop location, and the data to be transferred. (Before Java SE 6, a different `canImport` method was called that only supplies a list of data flavors.)

In the `canImport` method, you can also override the user drop action. For example, if a user chose the move action but it would be inappropriate to remove the original, you can force the transfer handler to use a copy action instead.

Here is a typical example. The image list component is willing to accept drops of file lists and images. However, if a file list is dragged into the component, a

user-selected MOVE action is changed into a COPY action, so that the image files do not get deleted.

```
public boolean canImport(TransferSupport support)
{
 if (support.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
 {
 if (support.getUserDropAction() == MOVE) support.setDropAction(COPY);
 return true;
 }
 else return support.isDataFlavorSupported(DataFlavor.imageFlavor);
}
```

A more sophisticated implementation could check that the files actually contain images.

The Swing components `JList`, `JTable`, `JTree`, and `JTextComponent` give visual feedback about insertion positions as the mouse is moved over the drop target. By default, the selection (for `JList`, `JTable`, and `JTree`) or the caret (for `JTextComponent`) is used to indicate the drop location. That approach is neither user-friendly nor flexible, and it is the default solely for backward compatibility. You should call the `setDropMode` method to choose a more appropriate visual feedback.

You can control whether the dropped data should overwrite existing items or be inserted between them. For example, in our sample program, we call

```
setDropMode(DropMode.ON_OR_INSERT);
```

to allow the user to drop onto an item (thereby replacing it), or to insert between two items (see Figure 11.46). Table 11.8 shows the drop modes supported by the Swing components.



**Figure 11.46** Visual indicators for dropping onto an item and between two items

Once the user completes the drop gesture, the `importData` method is invoked. You need to obtain the data from the drag source. Invoke the `getTransferable` method on the `TransferSupport` parameter to obtain a reference to a `Transferable` object. This is the same interface that is used for copy and paste.

**Table 11.8** Drop Modes

| Component      | Supported Drop Modes                                                                                    |
|----------------|---------------------------------------------------------------------------------------------------------|
| JList, JTree   | ON, INSERT, ON_OR_INSERT, USE_SELECTION                                                                 |
| JTable         | ON, INSERT, ON_OR_INSERT, INSERT_ROWS, INSERT_COLS, ON_OR_INSERT_ROWS, ON_OR_INSERT_COLS, USE_SELECTION |
| JTextComponent | INSERT, USE_SELECTION (actually moves the caret, not the selection)                                     |

One data type that is commonly used for drag and drop is the `DataFlavor.javaFileListFlavor`. A file list describes a set of files that are dropped onto the target. The transfer data is an object of type `List<File>`. Here is the code for retrieving the files:

```
DataFlavor[] flavors = transferable.getTransferDataFlavors();
if (Arrays.asList(flavors).contains(DataFlavor.javaFileListFlavor))
{
 List<File> fileList = (List<File>) transferable.getTransferData(DataFlavor.javaFileListFlavor);
 for (File f : fileList)
 {
 do something with f;
 }
}
```

When dropping into one of the components listed in Table 11.8, you need to know precisely where to drop the data. Invoke the `getDropLocation` method on the `TransferSupport` parameter to find where the drop occurred. This method returns an object of a subclass of `TransferHandler.DropLocation`. The `JList`, `JTable`, `JTree`, and `JTextComponent` classes define subclasses that specify location in the particular data model. For example, a location in a list is simply an integer index, but a location in a tree is a tree path. Here is how we obtain the drop location in our image list:

```
int index;
if (support.isDrop())
{
 JList.DropLocation location = (JList.DropLocation) support.getDropLocation();
 index = location.getIndex();
}
else index = model.size();
```

The `JList.DropLocation` subclass has a method `getIndex` that returns the index of the drop. (The `JTree.DropLocation` subclass has a method `getPath` instead.)

The `importData` method is also called when data are pasted into the component with the `Ctrl+V` keystroke. In that case, the `getDropLocation` method would throw an `IllegalStateException`. Therefore, if the `isDrop` method returns `false`, we simply append the pasted data to the end of the list.

When inserting into a list, table, or tree, you also need to check whether the data should be inserted between items or replace the item at the drop location. For a list, invoke the `isInsert` method of the `JList.DropLocation`. For the other components, see the API notes for their drop location classes at the end of this section.

To summarize, to turn a component into a drop target, add a transfer handler that specifies the following:

- When a dragged item can be accepted
- How the dropped data are imported

In addition, if you add drop support to a `JList`, `JTable`, `JTree`, or `JTextComponent`, you should set the drop mode.

Listing 11.22 shows the frame class of the program. Note that the `ImageList` class is both a drag source and a drop target. Try dragging images between the two lists. You can also drag image files from a file chooser of another program into the lists.

---

**Listing 11.22** dndImage/imageListDnDFrame.java

---

```
1 package dndImage;
2
3 import java.awt.*;
4 import java.awt.datatransfer.*;
5 import java.io.*;
6 import java.nio.file.*;
7 import java.util.*;
8 import java.util.List;
9 import javax.imageio.*;
10 import javax.swing.*;
11
12 public class ImageListDnDFrame extends JFrame
13 {
14 private static final int DEFAULT_WIDTH = 600;
15 private static final int DEFAULT_HEIGHT = 500;
16
17 private ImageList list1;
18 private ImageList list2;
19
20 public ImageListDnDFrame()
21 {
22 setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24 list1 = new ImageList(Paths.get(getClass().getPackage().getName(), "images1"));
25 list2 = new ImageList(Paths.get(getClass().getPackage().getName(), "images2"));
26 }
27 }
```

(Continues)

**Listing 11.22 (Continued)**

```
27 setLayout(new GridLayout(2, 1));
28 add(new JScrollPane(list1));
29 add(new JScrollPane(list2));
30 }
31 }
32
33 class ImageList extends JList<ImageIcon>
34 {
35 public ImageList(Path dir)
36 {
37 DefaultListModel<ImageIcon> model = new DefaultListModel<>();
38 try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir))
39 {
40 for (Path entry : entries)
41 model.addElement(new ImageIcon(entry.toString()));
42 }
43 catch (IOException ex)
44 {
45 ex.printStackTrace();
46 }
47
48 setModel(model);
49 setVisibleRowCount(0);
50 setLayoutOrientation(JList.HORIZONTAL_WRAP);
51 setDragEnabled(true);
52 setDropMode(DropMode.ON_OR_INSERT);
53 setTransferHandler(new ImageListTransferHandler());
54 }
55 }
56
57 class ImageListTransferHandler extends TransferHandler
58 {
59 // support for drag
60
61 public int getSourceActions(JComponent source)
62 {
63 return COPY_OR_MOVE;
64 }
65
66 protected Transferable createTransferable(JComponent source)
67 {
68 ImageList list = (ImageList) source;
69 int index = list.getSelectedIndex();
70 if (index < 0) return null;
71 ImageIcon icon = list.getModel().getElementAt(index);
72 return new ImageTransferable(icon.getImage());
73 }
74 }
```

```
75 protected void exportDone(JComponent source, Transferable data, int action)
76 {
77 if (action == MOVE)
78 {
79 ImageList list = (ImageList) source;
80 int index = list.getSelectedIndex();
81 if (index < 0) return;
82 DefaultListModel<?> model = (DefaultListModel<?>) list.getModel();
83 model.remove(index);
84 }
85 }
86
87 // support for drop
88
89 public boolean canImport(TransferSupport support)
90 {
91 if (support.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
92 {
93 if (support.getUserDropAction() == MOVE) support.setDropAction(COPY);
94 return true;
95 }
96 else return support.isDataFlavorSupported(DataFlavor.imageFlavor);
97 }
98
99 public boolean importData(TransferSupport support)
100 {
101 ImageList list = (ImageList) support.getComponent();
102 DefaultListModel<ImageIcon> model = (DefaultListModel<ImageIcon>) list.getModel();
103
104 Transferable transferable = support.getTransferable();
105 List<DataFlavor> flavors = Arrays.asList(transferable.getTransferDataFlavors());
106
107 List<Image> images = new ArrayList<>();
108
109 try
110 {
111 if (flavors.contains(DataFlavor.javaFileListFlavor))
112 {
113 @SuppressWarnings("unchecked") List<File> fileList
114 = (List<File>) transferable.getTransferData(DataFlavor.javaFileListFlavor);
115 for (File f : fileList)
116 {
117 try
118 {
119 images.add(ImageIO.read(f));
120 }
121 catch (IOException ex)
122 {
```

(Continues)

**Listing 11.22 (Continued)**

```
123 // couldn't read image--skip
124 }
125 }
126 }
127 else if (flavors.contains(DataFlavor.imageFlavor))
128 {
129 images.add((Image) transferable.getTransferData(DataFlavor.imageFlavor));
130 }
131
132 int index;
133 if (support.isDrop())
134 {
135 JList.DropLocation location = (JList.DropLocation) support.getDropLocation();
136 index = location.getIndex();
137 if (!location.isInsert()) model.remove(index); // replace location
138 }
139 else index = model.size();
140 for (Image image : images)
141 {
142 model.add(index, new ImageIcon(image));
143 index++;
144 }
145 return true;
146 }
147 catch (IOException | UnsupportedFlavorException ex)
148 {
149 return false;
150 }
151 }
152 }
```

---

**javax.swing.TransferHandler 1.4**

- **boolean canImport(TransferSupport support) 6**

Override to indicate whether the target component can accept the drag described by the TransferSupport parameter.

- **boolean importData(TransferSupport support) 6**

Override to carry out the drop or paste gesture described by the TransferSupport parameter, and return true if the import was successful.

**javax.swing.JFrame 1.2**

- void setTransferHandler(TransferHandler handler) 6

sets a transfer handler to handle drop and paste operations only

**javax.swing.JList 1.2****javax.swing.JTable 1.2****javax.swing.JTree 1.2****javax.swing.text.JTextComponent 1.2**

- void setDropMode(DropMode mode) 6

set the drop mode of this component to one of the values specified in Table 11.8 on p. 914.

**javax.swing.TransferHandler.TransferSupport 6**

- Component getComponent()

gets the target component of this transfer.

- DataFlavor[] getDataFlavors()

gets the data flavors of the data to be transferred.

- boolean isDrop()

true if this transfer is a drop, false if it is a paste.

- int getUserDropAction()

gets the drop action chosen by the user (MOVE, COPY, or LINK).

- getSourceDropActions()

gets the drop actions that are allowed by the drag source.

- getDropAction()

- setDropAction()

gets or sets the drop action of this transfer. Initially, this is the user drop action, but it can be overridden by the transfer handler.

- DropLocation getDropLocation()

gets the location of the drop, or throws an IllegalStateException if this transfer is not a drop.

**javax.swing.TransferHandler.DropLocation 6**

- `Point getDropPoint()`

gets the mouse location of the drop in the target component.

**javax.swing.JList.DropLocation 6**

- `boolean isInsert()`

returns true if the data are to be inserted before a given location, false if they are to replace existing data.

- `int getIndex()`

gets the model index for the insertion or replacement.

**javax.swing.JTable.DropLocation 6**

- `boolean isInsertRow()`

- `boolean isInsertColumn()`

returns true if data are to be inserted before a row or column.

- `int getRow()`

- `int getColumn()`

gets the model row or column index for the insertion or replacement, or -1 if the drop occurred in an empty area.

**javax.swing.JTree.DropLocation 6**

- `TreePath getPath()`

- `int getChildIndex()`

returns the tree path and child that, together with the drop mode of the target component, define the drop location, as described below.

**Drop Mode**

INSERT

ON or USE\_SELECTION

INSERT\_OR\_ON

**Tree Edit Action**

Insert as child of the path, before the child index.

Replace the data of the path (child index not used).

If the child index is -1, do as in ON, otherwise as in INSERT.

**javax.swing.text.JTextComponent.DropLocation 6**

- int getIndex()

the index at which to insert the data.

## 11.15 Platform Integration

We finish this chapter with several features for making Java applications feel more like native applications. The splash screen feature allows your application to display a splash screen as the virtual machine starts up. The `java.awt.Desktop` class lets you launch native applications such as the default browser and e-mail program. Finally, you now have access to the system tray and can clutter it up with icons, just like so many native applications do.

### 11.15.1 Splash Screens

A common complaint about Java applications is their long startup time. The Java virtual machine takes some time to load all required classes, particularly for a Swing application that needs to pull in large amounts of Swing and AWT library code. Users dislike applications that take a long time to bring up an initial screen, and they might even try launching the application multiple times if they suspect the first launch was unsuccessful. The remedy is a *splash screen*—a small window that appears quickly, telling the user that the application has been launched successfully.

Of course, you can put up a window as soon as your `main` method starts. However, the `main` method is only launched after the class loader has loaded all dependent classes, which might take a while.

Instead, you can ask the virtual machine to show an image immediately on launch. There are two mechanisms for specifying that image. You can use the `-splash` command-line option:

```
java -splash:myimage.png MyApp
```

Alternatively, you can specify it in the manifest of a JAR file:

```
Main-Class: MyApp
SplashScreen-Image: myimage.gif
```

The image is displayed immediately and automatically disappears when the first AWT window is made visible. You can supply any GIF, JPEG, or PNG image. Animation (in GIF) and transparency (GIF and PNG) are supported.

If your application is ready to go as soon as it reaches `main`, you can skip the remainder of this section. However, many applications use a plugin architecture in which a small core loads a set of plugins at startup. Eclipse and NetBeans are typical examples. In that case, you can indicate the loading progress on the splash screen.

There are two approaches. You can draw directly on the splash screen, or you can replace it with a borderless frame with identical contents and then draw inside the frame. Our sample program shows both techniques.

To draw directly on the splash screen, get a reference to the splash screen and get its graphics context and dimensions:

```
SplashScreen splash = SplashScreen.getSplashScreen();
Graphics2D g2 = splash.createGraphics();
Rectangle bounds = splash.getBounds();
```

You can now draw in the usual way. When you are done, call `update` to ensure that the drawing is refreshed. Our sample program draws a simple progress bar, as seen in the left image in Figure 11.47.

```
g.fillRect(x, y, width * percent / 100, height);
splash.update();
```



**Figure 11.47** The initial splash screen and a borderless follow-up window

---

**NOTE:** The splash screen is a singleton object. You cannot construct your own. If no splash screen was set on the command line or in the manifest, the `getSplashScreen` method returns null.

---

Drawing directly on the splash screen has a drawback. It is tedious to compute all pixel positions, and your progress indicator won't match the native progress bar. To avoid these problems, you can replace the initial splash screen with a follow-up window of the same size and content as soon as the `main` method starts. That window can contain arbitrary Swing components.

Our sample program in Listing 11.23 demonstrates this technique. The right image in Figure 11.47 shows a borderless frame with a panel that paints the splash screen and contains a `JProgressBar`. Now we have full access to the Swing API and can easily add message strings without having to fuss with pixel positions.

Note that we do not need to remove the initial splash screen. It is automatically removed as soon as the follow-up window is made visible.



**CAUTION:** Unfortunately, there is a noticeable flash when the splash screen is replaced by the follow-up window.

---

#### **Listing 11.23** `splashScreen/SplashScreenTest.java`

```
1 package SplashScreen;
2
3 import java.awt.*;
4 import java.util.List;
5 import javax.swing.*;
6
7 /**
8 * This program demonstrates the splash screen API.
9 * @version 1.01 2016-05-10
10 * @author Cay Horstmann
11 */
12 public class SplashScreenTest
13 {
14 private static final int DEFAULT_WIDTH = 300;
15 private static final int DEFAULT_HEIGHT = 300;
16
17 private static SplashScreen splash;
18 }
```

(Continues)

**Listing 11.23 (Continued)**

```
19 private static void drawOnSplash(int percent)
20 {
21 Rectangle bounds = splash.getBounds();
22 Graphics2D g = splash.createGraphics();
23 int height = 20;
24 int x = 2;
25 int y = bounds.height - height - 2;
26 int width = bounds.width - 4;
27 Color brightPurple = new Color(76, 36, 121);
28 g.setColor(brightPurple);
29 g.fillRect(x, y, width * percent / 100, height);
30 splash.update();
31 }
32
33 /**
34 * This method draws on the splash screen.
35 */
36 private static void init1()
37 {
38 splash = SplashScreen.getSplashScreen();
39 if (splash == null)
40 {
41 System.err.println("Did you specify a splash image with -splash or in the manifest?");
42 System.exit(1);
43 }
44
45 try
46 {
47 for (int i = 0; i <= 100; i++)
48 {
49 drawOnSplash(i);
50 Thread.sleep(100); // simulate startup work
51 }
52 }
53 catch (InterruptedException e)
54 {
55 }
56 }
57
58 /**
59 * This method displays a frame with the same image as the splash screen.
60 */
61 private static void init2()
62 {
63 final Image img = new ImageIcon(splash.getImageURL()).getImage();
64 }
```

```
65 final JFrame splashFrame = new JFrame();
66 splashFrame.setUndecorated(true);
67
68 final JPanel splashPanel = new JPanel()
69 {
70 public void paintComponent(Graphics g)
71 {
72 g.drawImage(img, 0, 0, null);
73 }
74 };
75
76 final JProgressBar progressBar = new JProgressBar();
77 progressBar.setStringPainted(true);
78 splashPanel.setLayout(new BorderLayout());
79 splashPanel.add(progressBar, BorderLayout.SOUTH);
80
81 splashFrame.add(splashPanel);
82 splashFrame.setBounds(splash.getBounds());
83 splashFrame.setVisible(true);
84
85 new SwingWorker<Void, Integer>()
86 {
87 protected Void doInBackground() throws Exception
88 {
89 try
90 {
91 for (int i = 0; i <= 100; i++)
92 {
93 publish(i);
94 Thread.sleep(100);
95 }
96 }
97 catch (InterruptedException e)
98 {
99 }
100 return null;
101 }
102
103 protected void process(List<Integer> chunks)
104 {
105 for (Integer chunk : chunks)
106 {
107 progressBar.setString("Loading module " + chunk);
108 progressBar.setValue(chunk);
109 splashPanel.repaint(); // because img is loaded asynchronously
110 }
111 }
112 }
```

(Continues)

**Listing 11.23 (Continued)**

```
113 protected void done()
114 {
115 splashFrame.setVisible(false);
116
117 JFrame frame = new JFrame();
118 frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
119 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
120 frame.setTitle("SplashScreenTest");
121 frame.setVisible(true);
122 }
123 }.execute();
124 }
125
126 public static void main(String args[])
127 {
128 init1();
129 EventQueue.invokeLater(() -> init2());
130 }
131 }
```

---

**java.awt.SplashScreen 6**

- **static SplashScreen get SplashScreen()**  
gets a reference to the splash screen, or `null` if no splash screen is present.
- **URL getImageURL()**
- **void setImageURL(URL imageURL)**  
gets or sets the URL of the splash screen image. Setting the image updates the splash screen.
- **Rectangle getBounds()**  
gets the bounds of the splash screen.
- **Graphics2D createGraphics()**  
gets a graphics context for drawing on the splash screen.
- **void update()**  
updates the display of the splash screen.
- **void close()**  
closes the splash screen. The splash screen is automatically closed when the first AWT window is made visible.

## 11.15.2 Launching Desktop Applications

The `java.awt.Desktop` class lets you launch the default browser and e-mail program. You can also open, edit, and print files, using the applications that are registered for the file type.

The API is very straightforward. First, call the static `isDesktopSupported` method. If it returns `true`, the current platform supports the launching of desktop applications. Then call the static `getDesktop` method to obtain a `Desktop` instance.

Not all desktop environments support all API operations. For example, in the Gnome desktop on Linux, it is possible to open files, but you cannot print them. (There is no support for “verbs” in file associations.) To find out what is supported on your platform, call the `isSupported` method, passing a value in the `Desktop.Action` enumeration. Our sample program contains tests such as the following:

```
if (desktop.isSupported(Desktop.Action.PRINT)) printButton.setEnabled(true);
```

To open, edit, or print a file, first check that the action is supported, and then call the `open`, `edit`, or `print` method. To launch the browser, pass a `URI`. (See Chapter 4 for more information on URIs.) You can simply call the `URI` constructor with a string containing an `http` or `https` URL.

To launch the default e-mail program, you need to construct a `URI` of a particular format, namely

`mailto:recipient?query`

Here `recipient` is the e-mail address of the recipient, such as `president@whitehouse.gov`, and `query` contains `&`-separated `name=value` pairs, with percent-encoded values. (Percent encoding is essentially the same as the URL encoding algorithm described in Chapter 4, but a space is encoded as `%20`, not `+`). An example is `subject=dinner%20RSVP&bcc=putin%40kremvax.ru`. The format is documented in RFC 2368 ([www.ietf.org/rfc/rfc2368.txt](http://www.ietf.org/rfc/rfc2368.txt)). Unfortunately, the `URI` class does not know anything about `mailto` URIs, so you have to assemble and encode your own.

Our sample program in Listing 11.24 lets you open, edit, or print a file of your choice, browse a URL, or launch your e-mail program (see Figure 11.48).

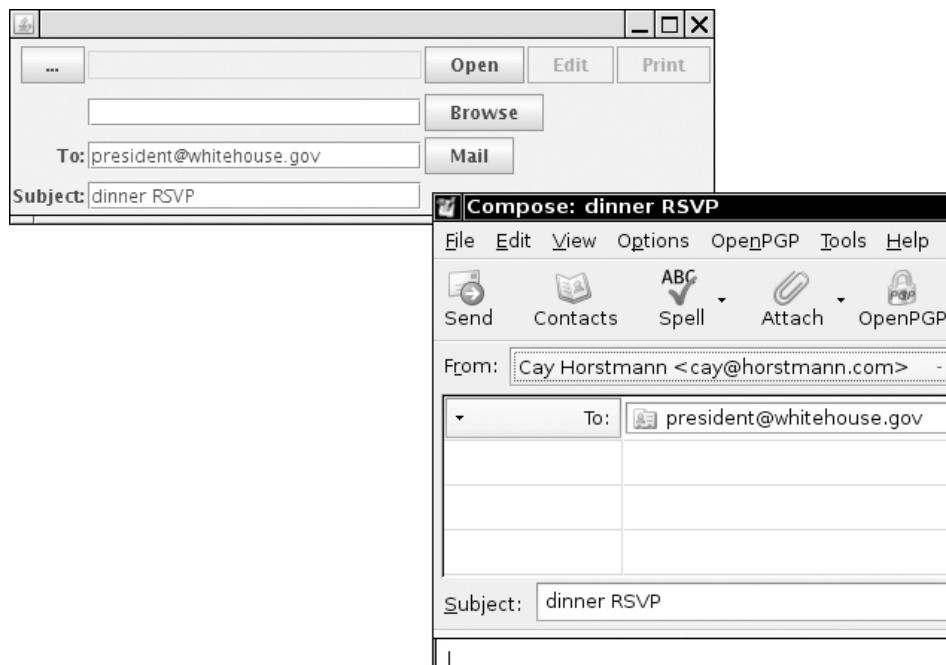


Figure 11.48 Launching a desktop application

---

**Listing 11.24** desktopApp/DesktopAppFrame.java

---

```
1 package desktopApp;
2
3 import java.awt.*;
4 import java.io.*;
5 import java.net.*;
6
7 import javax.swing.*;
8
9 class DesktopAppFrame extends JFrame
10 {
11 public DesktopAppFrame()
12 {
13 setLayout(new GridBagLayout());
14 final JFileChooser chooser = new JFileChooser();
15 JButton fileChooserButton = new JButton("...");
16 final JTextField fileField = new JTextField(20);
17 fileField.setEditable(false);
18 JButton openButton = new JButton("Open");
```

```
19 JButton editButton = new JButton("Edit");
20 JButton printButton = new JButton("Print");
21 final JTextField browseField = new JTextField();
22 JButton browseButton = new JButton("Browse");
23 final JTextField toField = new JTextField();
24 final JTextField subjectField = new JTextField();
25 JButton mailButton = new JButton("Mail");
26
27 openButton.setEnabled(false);
28 editButton.setEnabled(false);
29 printButton.setEnabled(false);
30 browseButton.setEnabled(false);
31 mailButton.setEnabled(false);
32
33 if (Desktop.isDesktopSupported())
34 {
35 Desktop desktop = Desktop.getDesktop();
36 if (desktop.isSupported(Desktop.Action.OPEN)) openButton.setEnabled(true);
37 if (desktop.isSupported(Desktop.Action.EDIT)) editButton.setEnabled(true);
38 if (desktop.isSupported(Desktop.Action.PRINT)) printButton.setEnabled(true);
39 if (desktop.isSupported(Desktop.Action.BROWSE)) browseButton.setEnabled(true);
40 if (desktop.isSupported(Desktop.Action.MAIL)) mailButton.setEnabled(true);
41 }
42
43 fileChooserButton.addActionListener(event ->
44 {
45 if (chooser.showOpenDialog(DesktopAppFrame.this) == JFileChooser.APPROVE_OPTION)
46 fileField.setText(chooser.getSelectedFile().getAbsolutePath());
47 });
48
49 openButton.addActionListener(event ->
50 {
51 try
52 {
53 Desktop.getDesktop().open(chooser.getSelectedFile());
54 }
55 catch (IOException ex)
56 {
57 ex.printStackTrace();
58 }
59 });
60
61 editButton.addActionListener(event ->
62 {
63 try
64 {
65 Desktop.getDesktop().edit(chooser.getSelectedFile());
66 }
```

(Continues)

**Listing 11.24 (Continued)**

```
67 catch (IOException ex)
68 {
69 ex.printStackTrace();
70 }
71 });
72
73 printButton.addActionListener(event ->
74 {
75 try
76 {
77 Desktop.getDesktop().print(chooser.getSelectedFile());
78 }
79 catch (IOException ex)
80 {
81 ex.printStackTrace();
82 }
83 });
84
85 browseButton.addActionListener(event ->
86 {
87 try
88 {
89 Desktop.getDesktop().browse(new URI/browseField.getText());
90 }
91 catch (URISyntaxException | IOException ex)
92 {
93 ex.printStackTrace();
94 }
95 });
96
97 mailButton.addActionListener(event ->
98 {
99 try
100 {
101 String subject = percentEncode(subjectField.getText());
102 URI uri = new URI("mailto:" + toField.getText() + "?subject=" + subject);
103
104 System.out.println(uri);
105 Desktop.getDesktop().mail(uri);
106 }
107 catch (URISyntaxException | IOException ex)
108 {
109 ex.printStackTrace();
110 }
111 });
112
```

```
113 JPanel buttonPanel = new JPanel();
114 ((FlowLayout) buttonPanel.getLayout()).setHgap(2);
115 buttonPanel.add(openButton);
116 buttonPanel.add(editButton);
117 buttonPanel.add(printButton);
118
119 add(fileChooserButton, new GBC(0, 0).setAnchor(GBC.EAST).setInsets(2));
120 add(fileField, new GBC(1, 0).setFill(GBC.HORIZONTAL));
121 add(buttonPanel, new GBC(2, 0).setAnchor(GBC.WEST).setInsets(0));
122 add(browseField, new GBC(1, 1).setFill(GBC.HORIZONTAL));
123 add(browseButton, new GBC(2, 1).setAnchor(GBC.WEST).setInsets(2));
124 add(new JLabel("To:"), new GBC(0, 2).setAnchor(GBC.EAST).setInsets(5, 2, 5, 2));
125 add(toField, new GBC(1, 2).setFill(GBC.HORIZONTAL));
126 add(mailButton, new GBC(2, 2).setAnchor(GBC.WEST).setInsets(2));
127 add(new JLabel("Subject:"), new GBC(0, 3).setAnchor(GBC.EAST).setInsets(5, 2, 5, 2));
128 add(subjectField, new GBC(1, 3).setFill(GBC.HORIZONTAL));
129
130 pack();
131 }
132
133 private static String percentEncode(String s)
134 {
135 try
136 {
137 return URLEncoder.encode(s, "UTF-8").replaceAll("[+]", "%20");
138 }
139 catch (UnsupportedEncodingException ex)
140 {
141 return null; // UTF-8 is always supported
142 }
143 }
144 }
```

**java.awt.Desktop 6**

- **static boolean isDesktopSupported()**  
returns true if launching desktop applications is supported on this platform.
- **static Desktop getDesktop()**  
returns the Desktop object for launching desktop operations. Throws an UnsupportedOperationException if this platform does not support launching desktop operations.
- **boolean isSupported(Desktop.Action action)**  
returns true if the given action is supported. action is one of OPEN, EDIT, PRINT, BROWSE, or MAIL.

*(Continues)*

**java.awt.Desktop 6 (Continued)**

- `void open(File file)`  
launches the application that is registered for viewing the given file.
- `void edit(File file)`  
launches the application that is registered for editing the given file.
- `void print(File file)`  
prints the given file.
- `void browse(URI uri)`  
launches the default browser with the given URI.
- `void mail()`
- `void mail(URI uri)`  
launches the default mailer. The second version can be used to fill in parts of the e-mail message.

### 11.15.3 The System Tray

Many desktop environments have an area for icons of programs that run in the background and occasionally notify users of events. In Windows, this area is called the *system tray*, and the icons are called *tray icons*. The Java API adopts the same terminology. A typical example of such a program is a monitor that checks for software updates. If new updates are available, the monitor program can change the appearance of the icon or display a message near the icon.

Frankly, the system tray is somewhat overused, and computer users are not usually filled with joy when they discover yet another tray icon. Our sample system tray application—a program that dispenses virtual fortune cookies—is no exception to that rule.

The `java.awt.SystemTray` class is the cross-platform conduit to the system tray. As in the `Desktop` class discussed in the preceding section, you first call the static `isSupported` method to check that the local Java platform supports the system tray. If so, you get a `SystemTray` singleton by calling the static `getSystemTray` method.

The most important method of the `SystemTray` class is the `add` method that lets you add a `TrayIcon` instance. A tray icon has three key properties:

- The icon image
- The tooltip that is visible when the mouse hovers over the icon

- The pop-up menu that is displayed when the user clicks on the icon with the right mouse button

The pop-up menu is an instance of the `PopupMenu` class of the AWT library, representing a native pop-up menu, not a Swing menu. Fill it out with AWT `MenuItem` instances, each having an action listener just like the Swing counterpart.

Finally, a tray icon can display notifications to the user (see Figure 11.49). Call the `displayMessage` method of the `TrayIcon` class and specify the caption, message, and message type.

```
trayIcon.displayMessage("Your Fortune", fortunes.get(index), TrayIcon.MessageType.INFO);
```



Figure 11.49 A notification from a tray icon

Listing 11.25 shows the application that places a fortune cookie icon into the system tray. The program reads a fortune cookie file (from the venerable UNIX `fortune` program) in which each fortune is terminated by a line containing a % character. It displays a message every ten seconds. Mercifully, there is a pop-up menu with a command to exit the application. If only all tray icons were so considerate!

**Listing 11.25** systemTray/SystemTrayTest.java

```
1 package systemTray;
2
3 import java.awt.*;
4 import java.io.*;
5 import java.util.*;
6 import java.util.List;
7
8 import javax.swing.*;
9 import javax.swing.Timer;
10
11 /**
12 * This program demonstrates the system tray API.
13 * @version 1.02 2016-05-10
14 * @author Cay Horstmann
15 */
16 public class SystemTrayTest
17 {
18 public static void main(String[] args)
19 {
20 SystemTrayApp app = new SystemTrayApp();
21 app.init();
22 }
23 }
24
25 class SystemTrayApp
26 {
27 public void init()
28 {
29 final TrayIcon trayIcon;
30
31 if (!SystemTray.isSupported())
32 {
33 System.err.println("System tray is not supported.");
34 return;
35 }
36
37 SystemTray tray = SystemTray.getSystemTray();
38 Image image = new ImageIcon(getClass().getResource("cookie.png")).getImage();
39
40 PopupMenu popup = new PopupMenu();
41 MenuItem exitItem = new MenuItem("Exit");
42 exitItem.addActionListener(event -> System.exit(0));
43 popup.add(exitItem);
44
45 trayIcon = new TrayIcon(image, "Your Fortune", popup);
46
47 trayIcon.setImageAutoSize(true);
```

```
48 trayIcon.addActionListener(event ->
49 {
50 trayIcon.displayMessage("How do I turn this off?",
51 "Right-click on the fortune cookie and select Exit.",
52 TrayIcon.MessageType.INFO);
53 });
54
55 try
56 {
57 tray.add(trayIcon);
58 }
59 catch (AWTException e)
60 {
61 System.err.println("TrayIcon could not be added.");
62 return;
63 }
64
65 final List<String> fortunes = readFortunes();
66 Timer timer = new Timer(10000, event ->
67 {
68 int index = (int) (fortunes.size() * Math.random());
69 trayIcon.displayMessage("Your Fortune", fortunes.get(index),
70 TrayIcon.MessageType.INFO);
71 });
72 timer.start();
73 }
74
75 private List<String> readFortunes()
76 {
77 List<String> fortunes = new ArrayList<>();
78 try (InputStream inStream = getClass().getResourceAsStream("fortunes"))
79 {
80 Scanner in = new Scanner(inStream, "UTF-8");
81 StringBuilder fortune = new StringBuilder();
82 while (in.hasNextLine())
83 {
84 String line = in.nextLine();
85 if (line.equals("%"))
86 {
87 fortunes.add(fortune.toString());
88 fortune = new StringBuilder();
89 }
90 else
91 {
92 fortune.append(line);
93 fortune.append(' ');
94 }
95 }
96 }
}
```

(Continues)

**Listing 11.25 (Continued)**

```
97 catch (IOException ex)
98 {
99 ex.printStackTrace();
100 }
101 return fortunes;
102 }
103 }
```

---

**java.awt.SystemTray 6**

- `static boolean isSupported()`  
returns true if system tray access is supported on this platform.
- `static SystemTray getSystemTray()`  
returns the SystemTray object for accessing the system tray. Throws an `UnsupportedOperationException` if this platform does not support system tray access.
- `Dimension getTrayIconSize()`  
gets the dimensions for an icon in the system tray.
- `void add(TrayIcon trayIcon)`
- `void remove(TrayIcon trayIcon)`  
adds or removes a system tray icon.

**java.awt.TrayIcon 6**

- `TrayIcon(Image image)`
- `TrayIcon(Image image, String tooltip)`
- `TrayIcon(Image image, String tooltip, PopupMenu popupMenu)`  
constructs a tray icon with the given image, tooltip, and pop-up menu.
- `Image getImage()`
- `void setImage(Image image)`
- `String getTooltip()`
- `void setTooltip(String tooltip)`
- `PopupMenu getPopupMenu()`
- `void setPopupMenu(PopupMenu popupMenu)`  
gets or sets the image, tooltip, or pop-up menu of this tooltip.

(Continues)

**java.awt.TrayIcon 6 (Continued)**

- `boolean isImageAutoSize()`
- `void setImageAutoSize(boolean autosize)`  
gets or sets the `imageAutoSize` property. If set, the image is scaled to fit the tooltip icon area; if not (the default), it is cropped (if too large) or centered (if too small).
- `void displayMessage(String caption, String text, TrayIcon.MessageType messageType)`  
displays a message near the tray icon. The message type is one of `INFO`, `WARNING`, `ERROR`, or `NONE`.
- `public void addActionListener(ActionListener listener)`
- `public void removeActionListener(ActionListener listener)`  
adds or removes an action listener when the listener called is platform-dependent.  
Typical cases are clicking on a notification or double-clicking on the tray icon.

You have now reached the end of this long chapter covering advanced AWT features. In the final chapter, we will turn to a different aspect of Java programming: interacting, on the same machine, with “native” code in a different programming language.

*This page intentionally left blank*

# Native Methods

## In this chapter

- 12.1 Calling a C Function from a Java Program, page 940
- 12.2 Numeric Parameters and Return Values, page 947
- 12.3 String Parameters, page 949
- 12.4 Accessing Fields, page 956
- 12.5 Encoding Signatures, page 961
- 12.6 Calling Java Methods, page 963
- 12.7 Accessing Array Elements, page 970
- 12.8 Handling Errors, page 974
- 12.9 Using the Invocation API, page 980
- 12.10 A Complete Example: Accessing the Windows Registry, page 985

While a “100% Pure Java” solution is nice in principle, there are situations in which you will want to write (or use) code written in another language. (Such code is usually called *native* code.)

Particularly in the early days of Java, many people assumed that it would be a good idea to use C or C++ to speed up critical parts of a Java application. However, in practice, this was rarely useful. A presentation at the 1996 JavaOne conference showed this clearly. The implementors of the cryptography library at Sun Microsystems reported that a pure Java platform implementation of their cryptographic functions was more than adequate. It was true that the code was not as

fast as a C implementation would have been, but it turned out not to matter. The Java platform implementation was far faster than the network I/O. This turned out to be the real bottleneck.

Of course, there are drawbacks to going native. If a part of your application is written in another language, you must supply a separate native library for every platform you want to support. Code written in C or C++ offers no protection against overwriting memory through invalid pointer usage. It is easy to write native methods that corrupt your program or infect the operating system.

Thus, we suggest using native code only when you need to. In particular, there are three reasons why native code might be the right choice:

- Your application requires access to system features or devices that are not accessible through the Java platform.
- You have substantial amounts of tested and debugged code in another language, and you know how to port it to all desired target platforms.
- You have found, through benchmarking, that the Java code is much slower than the equivalent code in another language.

The Java platform has an API for interoperating with native C code called the Java Native Interface (JNI). We'll discuss JNI programming in this chapter.



**C++ NOTE:** You can also use C++ instead of C to write native methods. There are a few advantages—type checking is slightly stricter, and accessing the JNI functions is a bit more convenient. However, JNI does not support any mapping between Java and C++ classes.

## 12.1 Calling a C Function from a Java Program

Suppose you have a C function that does something you like and, for one reason or another, you don't want to bother reimplementing it in Java. For the sake of illustration, we'll start with a simple C function that prints a greeting.

The Java programming language uses the keyword `native` for a native method, and you will obviously need to place a method in a class. The result is shown in Listing 12.1.

The `native` keyword alerts the compiler that the method will be defined externally. Of course, native methods will contain no code in the Java programming language, and the method header is followed immediately by a terminating semicolon. Therefore, native method declarations look similar to abstract method declarations.

**Listing 12.1** helloNative/HelloNative.java

```
1 /**
2 * @version 1.11 2007-10-26
3 * @author Cay Horstmann
4 */
5 class HelloNative
6 {
7 public static native void greeting();
8 }
```

**NOTE:** As in the previous chapter, we do not use packages here to keep examples simple.

In this particular example, the native method is also declared as `static`. Native methods can be both `static` and `nonstatic`. We'll start with a `static` method because we do not yet want to deal with parameter passing.

You can actually compile this class, but if you try to use it in a program, the virtual machine will tell you it doesn't know how to find `greeting`—reporting an `UnsatisfiedLinkError`. To implement the native code, write a corresponding C function. You must name that function *exactly* the way the Java virtual machine expects. Here are the rules:

1. Use the full Java method name, such as `HelloNative.greeting`. If the class is in a package, prepend the package name, such as `com.horstmann.HelloNative.greeting`.
2. Replace every period with an underscore, and append the prefix `Java_`. For example, `Java_HelloNative_greeting` or `Java_com_horstmann_HelloNative_greeting`.
3. If the class name contains characters that are not ASCII letters or digits—that is, `'_'`, `'$'`, or Unicode characters with codes greater than '`\u007F`'—replace them with `_xxxx`, where `xxxx` is the sequence of four hexadecimal digits of the character's Unicode value.

**NOTE:** If you *overload* native methods—that is, if you provide multiple native methods with the same name—you must append a double underscore followed by the encoded argument types. (We'll describe the encoding of the argument types later in this chapter.) For example, if you have a native method `greeting` and another native method `greeting(int repeat)`, then the first one is called `Java_HelloNative_greeting__`, and the second, `Java_HelloNative_greeting_I`.

Actually, nobody does this by hand; instead, run the `javah` utility which automatically generates the function names. To use `javah`, first compile the source file in Listing 12.1:

```
javac HelloNative.java
```

Next, call the `javah` utility, which produces a C header file from the class file. The `javah` executable can be found in the `jdk/bin` directory. Invoke it with the name of the class, just as you would start a Java program:

```
javah HelloNative
```

This command creates a header file, `HelloNative.h`, shown in Listing 12.2.

---

### Listing 12.2 `helloNative/HelloNative.h`

---

```
1 /* DO NOT EDIT THIS FILE - it is machine generated */
2 #include <jni.h>
3 /* Header for class HelloNative */
4
5 #ifndef _Included_HelloNative
6 #define _Included_HelloNative
7 #ifdef __cplusplus
8 extern "C" {
9 #endif
10 /*
11 * Class: HelloNative
12 * Method: greeting
13 * Signature: ()V
14 */
15 JNIEXPORT void JNICALL Java_HelloNative_greeting
16 (JNIEnv *, jclass);
17
18 #ifdef __cplusplus
19 }
20 #endif
21 #endif
```

---

As you can see, this file contains the declaration of a function `Java_HelloNative_greeting`. (The macros `JNIEXPORT` and `JNICALL` are defined in the header file `jni.h`. They denote compiler-dependent specifiers for exported functions that come from a dynamically loaded library.)

Now, simply copy the function prototype from the header file into a source file and give the implementation code for the function, as shown in Listing 12.3.

---

**Listing 12.3** helloNative/HelloNative.c

---

```
1 /*
2 * @version 1.10 1997-07-01
3 * @author Cay Horstmann
4 */
5
6 #include "HelloNative.h"
7 #include <stdio.h>
8
9 JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
10 {
11 printf("Hello Native World!\n");
12 }
```

---

In this simple function, ignore the `env` and `cl` arguments. You'll see their use later.



**C++ NOTE:** You can use C++ to implement native methods. However, you must then declare the functions that implement the native methods as `extern "C"`. (This stops the C++ compiler from “mangling” the method name.) For example,

```
extern "C"
JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
{
 cout << "Hello, Native World!" << endl;
```

---

Compile the native C code into a dynamically loaded library. The details depend on your compiler.

For example, with the GNU C compiler on Linux, use these commands:

```
gcc -fPIC -I jdk/include -I jdk/include/linux -shared -o libHelloNative.so HelloNative.c
```

With the Sun compiler under the Solaris operating system, the command is

```
cc -G -I jdk/include -I jdk/include/solaris -o libHelloNative.so HelloNative.c
```

With the Microsoft compiler under Windows, the command is

```
cl -I jdk\include -I jdk\include\win32 -LD HelloNative.c -FeHelloNative.dll
```

Here, *jdk* is the directory that contains the JDK.



**TIP:** If you use the Microsoft compiler from a command shell, first run a batch file such as `vsvars32.bat` or `vcvarsall.bat`. That batch file sets up the path and the environment variables needed by the compiler. You can find it in the directory `c:\Program Files\Microsoft Visual Studio 14.0\Common7\Tools`, or a similar monstrosity. Check the Visual Studio documentation for details.

---

You can also use the freely available Cygwin programming environment from [www.cygwin.com](http://www.cygwin.com). It contains the GNU C compiler and libraries for UNIX-style programming on Windows. With Cygwin, use the command

```
gcc -mno-cygwin -D __int64="long long" -I jdk/include/ -I jdk/include/win32
-shared -Wl,--add-stdcall-alias -o HelloNative.dll HelloNative.c
```

Type the entire command on a single line.

---

**NOTE:** The Windows version of the header file `jni_md.h` contains the type declaration

```
typedef __int64 jlong;
```

which is specific to the Microsoft compiler. If you use the GNU compiler, you might want to edit that file, for example,

```
#ifdef __GNUC__
 typedef long long jlong;
#else
 typedef __int64 jlong;
#endif
```

Alternatively, compile with `-D __int64="long long"`, as shown in the sample compiler invocation.

---

Finally, add a call to the `System.loadLibrary` method in your program. To ensure that the virtual machine will load the library before the first use of the class, use a static initialization block, as in Listing 12.4.

Figure 12.1 gives a summary of the native code processing.

After you compile and run this program, the message “Hello, Native World!” is displayed in a terminal window.

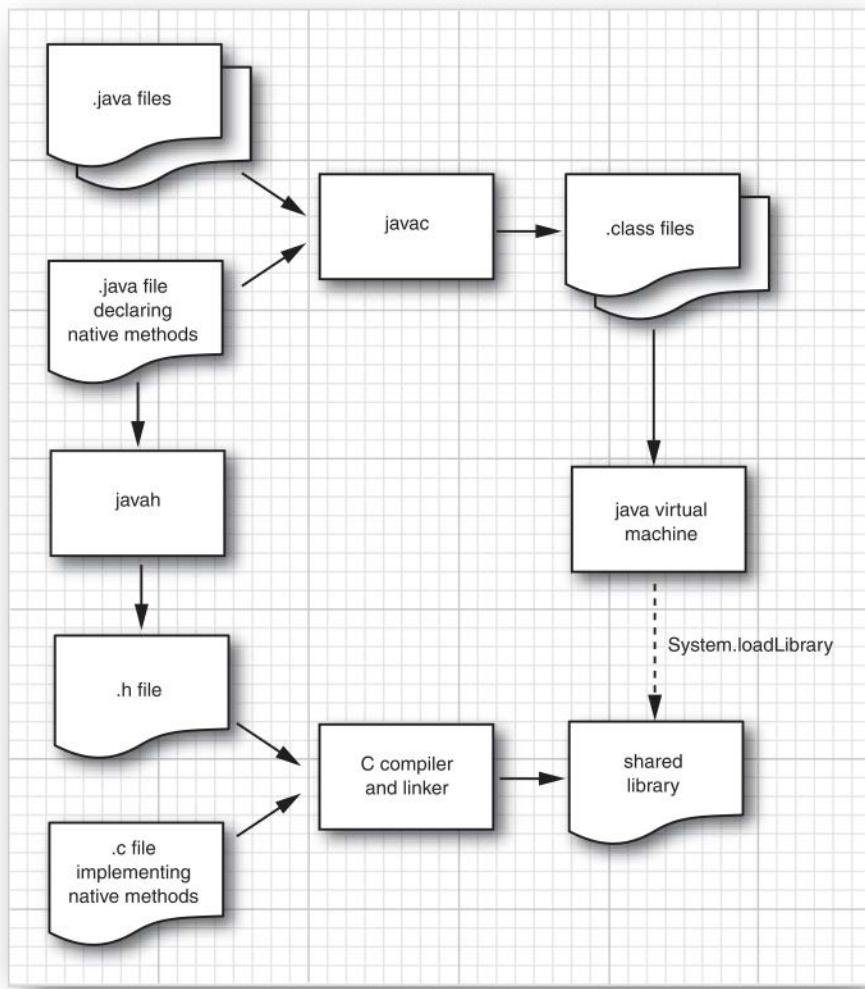


Figure 12.1 Processing native code

**Listing 12.4** helloNative/HelloNativeTest.java

```
1 /**
2 * @version 1.11 2007-10-26
3 * @author Cay Horstmann
4 */
```

(Continues)

**Listing 12.4 (Continued)**

```
5 class HelloNativeTest
6 {
7 public static void main(String[] args)
8 {
9 HelloNative.greeting();
10 }
11
12 static
13 {
14 System.loadLibrary("HelloNative");
15 }
16 }
```

---

**NOTE:** If you run Linux, you must add the current directory to the library path.  
Either set the `LD_LIBRARY_PATH` environment variable:

`export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH`

or set the `java.library.path` system property:

`java -Djava.library.path=. HelloNativeTest`

---

Of course, this is not particularly impressive by itself. However, keep in mind that this message is generated by the C `printf` command and not by any Java programming language code. We have taken the first step toward bridging the gap between the two languages!

In summary, follow these steps to link a native method to a Java program:

1. Declare a native method in a Java class.
2. Run `javah` to get a header file with a C declaration for the method.
3. Implement the native method in C.
4. Place the code in a shared library.
5. Load that library in your Java program.

**java.lang.System 1.0**

- `void loadLibrary(String libname)`

loads the library with the given name. The library is located in the library search path. The exact method for locating the library depends on the operating system.

---

**NOTE:** Some shared libraries for native code must execute certain initializations.

You can place any initialization code into a `JNI_OnLoad` method. Similarly, when the virtual machine (VM) shuts down, it will call the `JNI_OnUnload` method if you provide it. The prototypes are

```
jint JNI_OnLoad(JavaVM* vm, void* reserved);
void JNI_OnUnload(JavaVM* vm, void* reserved);
```

The `JNI_OnLoad` method needs to return the minimum version of the VM it requires, such as `JNI_VERSION_1_2`.

---

## 12.2 Numeric Parameters and Return Values

When passing numbers between C and Java, you should understand which types correspond to each other. For example, although C does have data types called `int` and `long`, their implementation is platform-dependent. On some platforms, an `int` is a 16-bit quantity, on others it is a 32-bit quantity. In the Java platform, of course, an `int` is *always* a 32-bit integer. For that reason, JNI defines types `jint`, `jlong`, and so on.

Table 12.1 shows the correspondence between Java types and C types.

**Table 12.1** Java Types and C Types

| Java Programming Language | C Programming Language | Bytes |
|---------------------------|------------------------|-------|
| <code>boolean</code>      | <code>jboolean</code>  | 1     |
| <code>byte</code>         | <code>jbyte</code>     | 1     |
| <code>char</code>         | <code>jchar</code>     | 2     |
| <code>short</code>        | <code>jshort</code>    | 2     |
| <code>int</code>          | <code>jint</code>      | 4     |
| <code>long</code>         | <code>jlong</code>     | 8     |
| <code>float</code>        | <code>jfloat</code>    | 4     |
| <code>double</code>       | <code>jdouble</code>   | 8     |

In the header file `jni.h`, these types are declared with `typedef` statements as the equivalent types on the target platform. That header file also defines the constants `JNI_FALSE = 0` and `JNI_TRUE = 1`.

Until Java SE 5.0, Java had no direct analog of the C `printf` function. In the following examples, we will suppose you are stuck with an ancient JDK release and decide

to implement the same functionality by calling the C `printf` function in a native method.

Listing 12.5 shows a class called `Printf1` that uses a native method to print a floating-point number with a given field width and precision.

---

**Listing 12.5** printf1/Printf1.java

```
1 /**
2 * @version 1.10 1997-07-01
3 * @author Cay Horstmann
4 */
5 class Printf1
6 {
7 public static native int print(int width, int precision, double x);
8
9 static
10 {
11 System.loadLibrary("Printf1");
12 }
13 }
```

---

Notice that when the method is implemented in C, all `int` and `double` parameters are changed to `jint` and `jdouble`, as shown in Listing 12.6.

---

**Listing 12.6** printf1/Printf1.c

```
1 /**
2 * @version 1.10 1997-07-01
3 * @author Cay Horstmann
4 */
5
6 #include "Printf1.h"
7 #include <stdio.h>
8
9 JNIEXPORT jint JNICALL Java_Printf1_print(JNIEnv* env, jclass cl,
10 jint width, jint precision, jdouble x)
11 {
12 char fmt[30];
13 jint ret;
14 sprintf(fmt, "%%%d.%df", width, precision);
15 ret = printf(fmt, x);
16 fflush(stdout);
17 return ret;
18 }
```

---

The function simply assembles a format string "%w.pf" in the variable `fmt`, then calls `printf`. It returns the number of characters printed.

Listing 12.7 shows the test program that demonstrates the `Printf1` class.

**Listing 12.7** `printf1/Printf1Test.java`

```
1 /**
2 * @version 1.10 1997-07-01
3 * @author Cay Horstmann
4 */
5 class Printf1Test
6 {
7 public static void main(String[] args)
8 {
9 int count = Printf1.print(8, 4, 3.14);
10 count += Printf1.print(8, 4, count);
11 System.out.println();
12 for (int i = 0; i < count; i++)
13 System.out.print("-");
14 System.out.println();
15 }
16 }
```

## 12.3 String Parameters

Next, we want to consider how to transfer strings to and from native methods. Strings are quite different in the two languages: In Java they are sequences of UTF-16 code points whereas C strings are null-terminated sequences of bytes. JNI has two sets of functions for manipulating strings: One converts Java strings to “modified UTF-8” byte sequences and another converts them to arrays of UTF-16 values—that is, to `jchar` arrays. (The UTF-8, “modified UTF-8,” and UTF-16 formats were discussed in Chapter 2. Recall that the UTF-8 and “modified UTF-8” encodings leave ASCII characters unchanged, but all other Unicode characters are encoded as multibyte sequences.)

---

**NOTE:** The standard UTF-8 encoding and the “modified UTF-8” encoding differ only for “supplementary” characters with codes higher than 0xFFFF. In the standard UTF-8 encoding, these characters are encoded as 4-byte sequences. In the “modified” encoding, each such character is first encoded as a pair of “surrogates” in the UTF-16 encoding, and then each surrogate is encoded with UTF-8, yielding a total of 6 bytes. This is clumsy, but it is a historical accident—the JVM specification was written when Unicode was still limited to 16 bits.

---

If your C code already uses Unicode, you'll want to use the second set of conversion functions. On the other hand, if all your strings are restricted to ASCII characters, you can use the "modified UTF-8" conversion functions.

A native method with a `String` parameter actually receives a value of an opaque type called `jstring`. A native method with a return value of type `String` must return a value of type `jstring`. JNI functions read and construct these `jstring` objects. For example, the `NewStringUTF` function makes a new `jstring` object out of a `char` array that contains ASCII characters or, more generally, "modified UTF-8"-encoded byte sequences.

JNI functions have a somewhat odd calling convention. Here is a call to the `NewStringUTF` function:

```
JNIEXPORT jstring JNICALL Java_HelloNative_getGreeting(JNIEnv* env, jclass c)
{
 jstring jstr;
 char greeting[] = "Hello, Native World\n";
 jstr = (*env)->NewStringUTF(env, greeting);
 return jstr;
}
```

---

**NOTE:** Unless explicitly mentioned otherwise, all code in this chapter is C code.

---

All calls to JNI functions use the `env` pointer that is the first argument of every native method. The `env` pointer is a pointer to a table of function pointers (see Figure 12.2). Therefore, you must prefix every JNI call with `(*env)->` to actually dereference the function pointer. Furthermore, `env` is the first parameter of every JNI function.



**C++ NOTE:** It is simpler to access JNI functions in C++. The C++ version of the `JNIEnv` class has inline member functions that take care of the function pointer lookup for you. For example, you can call the `NewStringUTF` function as

```
jstr = env->NewStringUTF(greeting);
```

Note that you omit the `JNIEnv` pointer from the parameter list of the call.

---

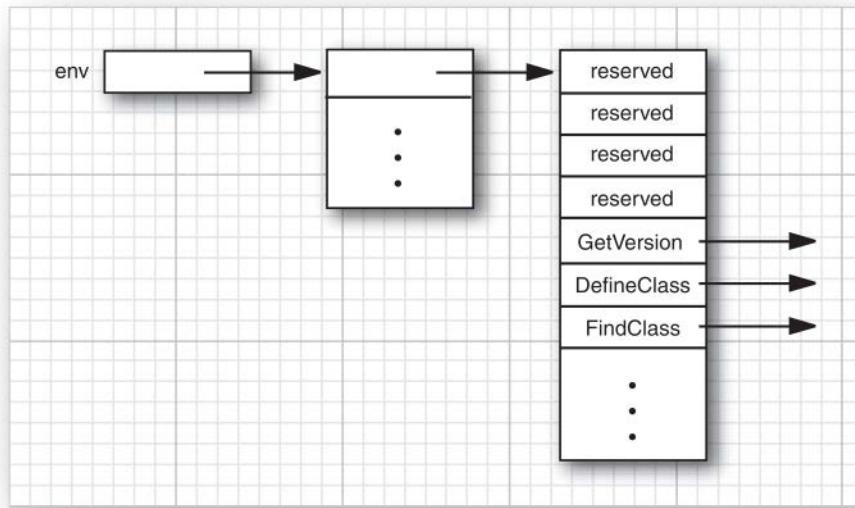


Figure 12.2 The `env` pointer

The `NewStringUTF` function lets you construct a new `jstring`. To read the contents of an existing `jstring` object, use the `GetStringUTFChars` function. This function returns a `const jbyte*` pointer to the “modified UTF-8” characters that describe the character string. Note that a specific virtual machine is free to choose this character encoding for its internal string representation, so you might get a character pointer into the actual Java string. Since Java strings are meant to be immutable, it is *very* important that you treat the `const` seriously and do not try to write into this character array. On the other hand, if the virtual machine uses UTF-16 or UTF-32 characters for its internal string representation, this function call allocates a new memory block that will be filled with the “modified UTF-8” equivalents.

The virtual machine must know when you are finished using the string so that it can garbage-collect it. (The garbage collector runs in a separate thread, and it can interrupt the execution of native methods.) For that reason, you must call the `ReleaseStringUTFChars` function.

Alternatively, you can supply your own buffer to hold the string characters by calling the `GetStringRegion` or `GetStringUTFRegion` methods.

Finally, the `GetStringUTFLength` function returns the number of characters needed for the “modified UTF-8” encoding of the string.

---

**NOTE:** You can find the JNI API at <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/>.

---

#### Accessing Java Strings from C Code

- `jstring NewStringUTF(JNIEnv* env, const char bytes[])`  
returns a new Java string object from a zero byte-terminated “modified UTF-8” byte sequence, or `NULL` if the string cannot be constructed.
- `jsize GetStringUTFLength(JNIEnv* env, jstring string)`  
returns the number of bytes required for the “modified UTF-8” encoding (not counting the zero byte terminator).
- `const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean* isCopy)`  
returns a pointer to the “modified UTF-8” encoding of a string, or `NULL` if the character array cannot be constructed. The pointer is valid until `ReleaseStringUTFChars` is called. `isCopy` points to a `jboolean` filled with `JNI_TRUE` if a copy is made, with `JNI_FALSE` otherwise.
- `void ReleaseStringUTFChars(JNIEnv* env, jstring string, const jbyte bytes[])`  
informs the virtual machine that the native code no longer needs access to the Java string through `bytes` (a pointer returned by `GetStringUTFChars`).
- `void GetStringRegion(JNIEnv *env, jstring string, jsize start, jsize length, jchar *buffer)`  
copies a sequence of UTF-16 double bytes from a string to a user-supplied buffer of size at least  $2 \times \text{length}$ .
- `void GetStringUTFRegion(JNIEnv *env, jstring string, jsize start, jsize length, jbyte *buffer)`  
copies a sequence of “modified UTF-8” bytes from a string to a user-supplied buffer. The buffer must be long enough to hold the bytes. In the worst case,  $3 \times \text{length}$  bytes are copied.

(Continues)

**Accessing Java Strings from C Code (Continued)**

- `jstring NewString(JNIEnv* env, const jchar chars[], jsize length)`  
returns a new Java string object from a Unicode string, or `NULL` if the string cannot be constructed.

*Parameters:*    `env`              The JNI interface pointer  
                  `chars`          The null-terminated UTF-16 string  
                  `length`        The number of characters in the string
- `jsize GetStringLength(JNIEnv* env, jstring string)`  
returns the number of characters in the string.
- `const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean* isCopy)`  
returns a pointer to the Unicode encoding of a string, or `NULL` if the character array cannot be constructed. The pointer is valid until `ReleaseStringChars` is called. `isCopy` is either `NULL` or points to a `jboolean` filled with `JNI_TRUE` if a copy is made, with `JNI_FALSE` otherwise.
- `void ReleaseStringChars(JNIEnv* env, jstring string, const jchar chars[])`  
informs the virtual machine that the native code no longer needs access to the Java string through `chars` (a pointer returned by `GetStringChars`).

Let us put these functions to work and write a class that calls the C function `sprintf`. We would like to call the function as shown in Listing 12.8.

**Listing 12.8 printf2/Printf2Test.java**

```
1 /**
2 * @version 1.10 1997-07-01
3 * @author Cay Horstmann
4 */
5 class Printf2Test
6 {
7 public static void main(String[] args)
8 {
9 double price = 44.95;
10 double tax = 7.75;
11 double amountDue = price * (1 + tax / 100);
12
13 String s = Printf2.printf("Amount due = %8.2f", amountDue);
14 System.out.println(s);
15 }
16 }
```

Listing 12.9 shows the class with the native `sprint` method.

---

**Listing 12.9** printf2/Printf2.java

---

```
1 /**
2 * @version 1.10 1997-07-01
3 * @author Cay Horstmann
4 */
5 class Printf2
6 {
7 public static native String sprint(String format, double x);
8
9 static
10 {
11 System.loadLibrary("Printf2");
12 }
13 }
```

---

Therefore, the C function that formats a floating-point number has the prototype

```
JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* env, jclass cl, jstring format, jdouble x)
```

Listing 12.10 shows the code for the C implementation. Note the calls to `GetStringUTFChars` to read the format argument, `NewStringUTF` to generate the return value, and `ReleaseStringUTFChars` to inform the virtual machine that access to the string is no longer required.

---

**Listing 12.10** printf2/Printf2.c

---

```
1 /**
2 * @version 1.10 1997-07-01
3 * @author Cay Horstmann
4 */
5
6 #include "Printf2.h"
7 #include <string.h>
8 #include <stdlib.h>
9 #include <float.h>
10
11 /**
12 * @param format a string containing a printf format specifier
13 * (such as "%8.2f"). Substrings "%%" are skipped.
14 * @return a pointer to the format specifier (skipping the '%')
15 * or NULL if there wasn't a unique format specifier
16 */
17 char* find_format(const char format[])
18 {
19 char* p;
```

```
20 char* q;
21
22 p = strchr(format, '%');
23 while (p != NULL && *(p + 1) == '%') /* skip %% */
24 p = strchr(p + 2, '%');
25 if (p == NULL) return NULL;
26 /* now check that % is unique */
27 p++;
28 q = strchr(p, '%');
29 while (q != NULL && *(q + 1) == '%') /* skip %% */
30 q = strchr(q + 2, '%');
31 if (q != NULL) return NULL; /* % not unique */
32 q = p + strspn(p, " -0#"); /* skip past flags */
33 q += strspn(q, "0123456789"); /* skip past field width */
34 if (*q == '.') { q++; q += strspn(q, "0123456789"); }
35 /* skip past precision */
36 if (strchr("eEfFgG", *q) == NULL) return NULL;
37 /* not a floating-point format */
38 return p;
39 }
40
41 JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* env, jclass cl,
42 jstring format, jdouble x)
43 {
44 const char* cformat;
45 char* fmt;
46 jstring ret;
47
48 cformat = (*env)->GetStringUTFChars(env, format, NULL);
49 fmt = find_format(cformat);
50 if (fmt == NULL)
51 ret = format;
52 else
53 {
54 char* cret;
55 int width = atoi(fmt);
56 if (width == 0) width = DBL_DIG + 10;
57 cret = (char*) malloc(strlen(cformat) + width);
58 sprintf(cret, cformat, x);
59 ret = (*env)->NewStringUTF(env, cret);
60 free(cret);
61 }
62 (*env)->ReleaseStringUTFChars(env, format, cformat);
63 return ret;
64 }
```

In this function, we chose to keep error handling simple. If the format code to print a floating-point number is not of the form `%w.pc`, where `c` is one of the

characters e, E, f, g, or G, then we simply do not format the number. We'll show you later how to make a native method throw an exception.

## 12.4 Accessing Fields

All the native methods you saw so far were static methods with number and string parameters. We'll now consider native methods that operate on objects. As an exercise, we will reimplement as native a method of the Employee class that was introduced in Volume I, Chapter 4. Again, this is not something you would normally want to do, but it does illustrate how to access fields from a native method when you need to do so.

### 12.4.1 Accessing Instance Fields

To see how to access instance fields from a native method, we will reimplement the `raiseSalary` method. Here is the code in Java:

```
public void raiseSalary(double byPercent)
{
 salary *= 1 + byPercent / 100;
}
```

Let us rewrite this as a native method. Unlike the previous examples of native methods, this is not a static method. Running `javah` gives the following prototype:

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv *, jobject, jdouble);
```

Note the second argument. It is no longer of type `jclass` but of type `jobject`. In fact, it is an equivalent of the `this` reference. Static methods obtain a reference to the class, whereas nonstatic methods obtain a reference to the implicit `this` argument object.

Now we access the `salary` field of the implicit argument. In the “raw” Java-to-C binding of Java 1.0, this was easy—a programmer could directly access object data fields. However, direct access requires all virtual machines to expose their internal data layout. For that reason, the JNI requires programmers to get and set the values of data fields by calling special JNI functions.

In our case, we need to use the `GetDoubleField` and `SetDoubleField` functions because the type of `salary` is `double`. There are other functions—`GetIntField/SetIntField`, `GetObjectField/SetObjectField`, and so on for other field types. The general syntax is:

```
x = (*env)->GetXxxField(env, this_obj, fieldID);
(*env)->SetXxxField(env, this_obj, fieldID, x);
```

Here, `fieldID` is a value of a special type, `jfieldID`, that identifies a field in a structure, and `Xxx` represents a Java data type (`Object`, `Boolean`, `Byte`, and so on). To obtain the `fieldID`, you must first get a value representing the class, which you can do in one of two ways. The `GetObjectClass` function returns the class of any object. For example:

```
jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
```

The `FindClass` function lets you specify the class name as a string (curiously, with `/` characters instead of periods as package name separators).

```
jclass class_String = (*env)->FindClass(env, "java/lang/String");
```

Use the `GetFieldID` function to obtain the `fieldID`. You must supply the name of the field and its *signature*, an encoding of its type. For example, here is the code to obtain the field ID of the `salary` field:

```
jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");
```

The string `"D"` denotes the type `double`. You'll learn the complete rules for encoding signatures in the next section.

You might be thinking that accessing a data field is quite convoluted. The designers of the JNI did not want to expose the data fields directly, so they had to supply functions for getting and setting field values. To minimize the cost of these functions, computing the field ID from the field name—which is the most expensive step—is factored out into a separate step. That is, if you repeatedly get and set the value of a particular field, you can incur the cost of computing the field identifier only once.

Let us put all the pieces together. The following code reimplements the `raiseSalary` method as a native method:

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env, jobject this_obj, jdouble byPercent)
{
 /* get the class */
 jclass class_Employee = (*env)->GetObjectClass(env, this_obj);

 /* get the field ID */
 jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");

 /* get the field value */
 jdouble salary = (*env)->GetDoubleField(env, this_obj, id_salary);

 salary *= 1 + byPercent / 100;

 /* set the field value */
 (*env)->SetDoubleField(env, this_obj, id_salary, salary);
}
```



**CAUTION:** Class references are only valid until the native method returns. You cannot cache the return values of `GetObjectClass` in your code. Do *not* store away a class reference for reuse in a later method call. You must call `GetObjectClass` every time the native method executes. If this is intolerable, you can lock the reference with a call to `NewGlobalRef`:

```
static jclass class_X = 0;
static jfieldID id_a;

. . .

if (class_X == 0)
{
 jclass cx = (*env)->GetObjectClass(env, obj);
 class_X = (*env)->NewGlobalRef(env, cx);
 id_a = (*env)->GetFieldID(env, cls, "a", ". . .");
}
```

Now you can use the class reference and field IDs in subsequent calls. When you are done using the class, make sure to call

```
(*env)->DeleteGlobalRef(env, class_X);
```

---

Listings 12.11 and 12.12 show the Java code for a test program and the `Employee` class. Listing 12.13 contains the C code for the native `raiseSalary` method.

#### **Listing 12.11** employee/EmployeeTest.java

```
1 /**
2 * @version 1.10 1999-11-13
3 * @author Cay Horstmann
4 */
5
6 public class EmployeeTest
7 {
8 public static void main(String[] args)
9 {
10 Employee[] staff = new Employee[3];
11
12 staff[0] = new Employee("Harry Hacker", 35000);
13 staff[1] = new Employee("Carl Cracker", 75000);
14 staff[2] = new Employee("Tony Tester", 38000);
15
16 for (Employee e : staff)
17 e.raiseSalary(5);
18 for (Employee e : staff)
19 e.print();
20 }
21 }
```

---

**Listing 12.12** employee/Employee.java

```
1 /**
2 * @version 1.10 1999-11-13
3 * @author Cay Horstmann
4 */
5
6 public class Employee
7 {
8 private String name;
9 private double salary;
10
11 public native void raiseSalary(double byPercent);
12
13 public Employee(String n, double s)
14 {
15 name = n;
16 salary = s;
17 }
18
19 public void print()
20 {
21 System.out.println(name + " " + salary);
22 }
23
24 static
25 {
26 System.loadLibrary("Employee");
27 }
28 }
```

**Listing 12.13** employee/Employee.c

```
1 /**
2 * @version 1.10 1999-11-13
3 * @author Cay Horstmann
4 */
5
6 #include "Employee.h"
7
8 #include <stdio.h>
9
10 JNIREPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env, jobject this_obj, jdouble byPercent)
11 {
12 /* get the class */
13 jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
14 }
```

*(Continues)*

**Listing 12.13 (Continued)**

```

15 /* get the field ID */
16 jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");
17
18 /* get the field value */
19 jdouble salary = (*env)->GetDoubleField(env, this_obj, id_salary);
20
21 salary *= 1 + byPercent / 100;
22
23 /* set the field value */
24 (*env)->SetDoubleField(env, this_obj, id_salary, salary);
25 }
```

### 12.4.2 Accessing Static Fields

Accessing static fields is similar to accessing nonstatic fields. Use the `GetStaticFieldID` and `GetStaticXxxField`/`SetStaticXxxField` functions that work almost identically to their nonstatic counterparts, with two differences:

- As you have no object, you must use `FindClass` instead of `GetObjectClass` to obtain the class reference.
- You have to supply the class, not the instance object, when accessing the field.

For example, here is how you can get a reference to `System.out`:

```

/* get the class */
jclass class_System = (*env)->FindClass(env, "java/lang/System");

/* get the field ID */
jfieldID id_out = (*env)->GetStaticFieldID(env, class_System, "out",
 "Ljava/io/PrintStream;");

/* get the field value */
jobject obj_out = (*env)->GetStaticObjectField(env, class_System, id_out);
```

**Accessing Fields**

- `jfieldID GetFieldID(JNIEnv *env, jclass cl, const char name[], const char fieldSignature[])`  
returns the identifier of a field in a class.
- `Xxx GetXxxField(JNIEnv *env, jobject obj, jfieldID id)`  
returns the value of a field. The field type `Xxx` is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.

(Continues)

**Accessing Fields (Continued)**

- `void SetXxxField(JNIEnv *env, jobject obj, jfieldID id, Xxx value)`  
sets a field to a new value. The field type `Xxx` is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.
- `jfieldID GetStaticFieldID(JNIEnv *env, jclass cl, const char name[], const char fieldSignature[])`  
returns the identifier of a static field in a class.
- `Xxx GetStaticXxxField(JNIEnv *env, jclass cl, jfieldID id)`  
returns the value of a static field. The field type `Xxx` is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.
- `void SetStaticXxxField(JNIEnv *env, jclass cl, jfieldID id, Xxx value)`  
sets a static field to a new value. The field type `Xxx` is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`.

## 12.5 Encoding Signatures

To access instance fields and call methods defined in the Java programming language, you need to learn the rules for “mangling” the names of data types and method signatures. (A method signature describes the parameters and return type of the method.) Here is the encoding scheme:

|             |              |
|-------------|--------------|
| B           | byte         |
| C           | char         |
| D           | double       |
| F           | float        |
| I           | int          |
| J           | long         |
| Lclassname; | a class type |
| S           | short        |
| V           | void         |
| Z           | boolean      |

To describe an array type, use a `[`. For example, an array of strings is

`[Ljava/lang/String;`

A `float[][]` is mangled into

`[[F`

For the complete signature of a method, list the parameter types inside a pair of parentheses and then list the return type. For example, a method receiving two integers and returning an integer is encoded as

(II)I

The `sprint` method in Section 12.3, “String Parameters,” on p. 949 has a mangled signature of

`(Ljava/lang/String;D)Ljava/lang/String;`

That is, the method receives a `String` and a `double` and returns a `String`.

Note that the semicolon at the end of the `L` expression is the terminator of the type expression, not a separator between parameters. For example, the constructor

`Employee(java.lang.String, double, java.util.Date)`

has a signature

`"(Ljava/lang/String;DLjava/util/Date;)V"`

Note that there is no separator between the `D` and `Ljava/util/Date;.` Also note that in this encoding scheme, you must use `/` instead of `.` to separate the package and class names. The `V` at the end denotes a return type of `void`. Even though you don’t specify a return type for constructors in Java, you need to add a `V` to the virtual machine signature.



**TIP:** You can use the `javap` command with option `-s` to generate the method signatures from class files. For example, run

```
javap -s -private Employee
```

You will get the following output, displaying the signatures of all fields and methods:

```
Compiled from "Employee.java"
public class Employee extends java.lang.Object{
 private java.lang.String name;
 Signature: Ljava/lang/String;
 private double salary;
 Signature: D
 public Employee(java.lang.String, double);
 Signature: (Ljava/lang/String;D)V
 public native void raiseSalary(double);
 Signature: (D)V
 public void print();
 Signature: ()V
```

```
static {};
 Signature: ()V
}
```

---

**NOTE:** There is no rationale whatsoever for forcing programmers to use this mangling scheme for signatures. The designers of the native calling mechanism could have just as easily written a function that reads signatures in the Java programming language style, such as `void(int,java.lang.String)`, and encodes them into whatever internal representation they prefer. Then again, using the mangled signatures lets you partake in the mystique of programming close to the virtual machine.

---

## 12.6 Calling Java Methods

Of course, Java programming language functions can call C functions—that is what native methods are for. Can we go the other way? Why would we want to do this anyway? It often happens that a native method needs to request a service from an object that was passed to it. We'll first show you how to do it for instance methods, then for static methods.

### 12.6.1 Instance Methods

As an example of calling a Java method from native code, let's enhance the `Printf` class and add a method that works similarly to the C function `fprintf`. That is, it should be able to print a string on an arbitrary `PrintWriter` object. Here is the definition of the method in Java:

```
class Printf3
{
 public native static void fprintf(PrintWriter out, String s, double x);
 ...
}
```

We'll first assemble the string to be printed into a `String` object `str`, as in the `sprint` method that we already implemented. Then, from the C function that implements the native method, we'll call the `print` method of the `PrintWriter` class.

You can call any Java method from C by using the function call

```
(*env)->CallXxxMethod(env, implicit parameter, methodID, explicit parameters)
```

Replace `Xxx` with `Void`, `Int`, `Object`, and so on, depending on the return type of the method. Just as you need a `fieldID` to access a field of an object, you need a method

ID to call a method. To obtain a method ID, call the JNI function `GetMethodID` and supply the class, the name of the method, and the method signature.

In our example, we want to obtain the ID of the `print` method of the `PrintWriter` class. The `PrintWriter` class has several overloaded methods called `print`. For that reason, you must also supply a string describing the parameters and the return value of the specific function that you want to use. For example, we want to use `void print(java.lang.String)`. As described in the preceding section, we must now “mangle” the signature into the string `"(Ljava/lang/String;)V"`.

Here is the complete code to make the method call, by

1. Obtaining the class of the implicit parameter
2. Obtaining the method ID
3. Making the call

```
/* get the class */
class_PrintWriter = (*env)->GetObjectClass(env, out);

/* get the method ID */
id_print = (*env)->GetMethodID(env, class_PrintWriter, "print", "(Ljava/lang/String;)V");

/* call the method */
(*env)->CallVoidMethod(env, out, id_print, str);
```

Listings 12.14 and 12.15 show the Java code for a test program and the `Printf3` class. Listing 12.16 contains the C code for the native `fprint` method.

---

**NOTE:** The numerical method IDs and field IDs are conceptually similar to `Method` and `Field` objects in the reflection API. You can convert between them with the following functions:

```
jobject ToReflectedMethod(JNIEnv* env, jclass class, jmethodID methodID);
 // returns Method object
methodID FromReflectedMethod(JNIEnv* env, jobject method);
jobject ToReflectedField(JNIEnv* env, jclass class, jfieldID fieldID);
 // returns Field object
fieldID FromReflectedField(JNIEnv* env, jobject field);
```

---

## 12.6.2 Static Methods

Calling static methods from native methods is similar to calling instance methods. There are two differences:

- Use the `GetStaticMethodID` and `CallStaticXxxMethod` functions

- Supply a class object, not an implicit parameter object, when invoking the method

As an example of this, let's make the call to the static method

```
System.getProperty("java.class.path")
```

from a native method. The return value of this call is a string that gives the current class path.

First, we have to find the class to use. As we have no object of the class `System` readily available, we use `FindClass` rather than `GetObjectClass`.

```
jclass class_System = (*env)->FindClass(env, "java/lang/System");
```

Next, we need the ID of the static `getProperty` method. The encoded signature of that method is

```
"(Ljava/lang/String;)Ljava/lang/String;"
```

because both the parameter and the return value are strings. Hence, we obtain the method ID as follows:

```
jmethodID id_getProperty = (*env)->GetStaticMethodID(env, class_System, "getProperty",
"(Ljava/lang/String;)Ljava/lang/String;");
```

Finally, we can make the call. Note that the class object is passed to the `CallStaticObjectMethod` function.

```
jobject obj_ret = (*env)->CallStaticObjectMethod(env, class_System, id_getProperty,
(*env)->NewStringUTF(env, "java.class.path"));
```

The return value of this method is of type `jobject`. If we want to manipulate it as a string, we must cast it to `jstring`:

```
jstring str_ret = (jstring) obj_ret;
```



**C++ NOTE:** In C, the types `jstring` and `jclass`, as well as the array types to be introduced later, are all type-equivalent to `jobject`. The cast of the preceding example is therefore not strictly necessary in C. But in C++, these types are defined as pointers to “dummy classes” that have the correct inheritance hierarchy. For example, assigning a `jstring` to a `jobject` is legal without a cast in C++, but an assignment from a `jobject` to a `jstring` requires a cast.

### 12.6.3 Constructors

A native method can create a new Java object by invoking its constructor. Invoke the constructor by calling the `NewObject` function.

```
 jobject obj_new = (*env)->NewObject(env, class, methodID, construction parameters);
```

You can obtain the method ID needed for this call from the `GetMethodID` function by specifying the method name as "`<init>`" and the encoded signature of the constructor (with return type `void`). For example, here is how a native method can create a `FileOutputStream` object:

```
const char[] fileName = ". . .";
jstring str_fileName = (*env)->NewStringUTF(env, fileName);
 jclass class_FileOutputStream = (*env)->FindClass(env, "java/io/FileOutputStream");
jmethodID id_FileOutputStream
 = (*env)->GetMethodID(env, class_FileOutputStream, "<init>", "(Ljava/lang/String;)V");
 jobject obj_stream
 = (*env)->NewObject(env, class_FileOutputStream, id_FileOutputStream, str_fileName);
```

Note that the signature of the constructor takes a parameter of type `java.lang.String` and has a return type of `void`.

## 12.6.4 Alternative Method Invocations

Several variants of the JNI functions can be used to call a Java method from native code. These are not as important as the functions that we already discussed, but they are occasionally useful.

The `CallNonvirtualXxxMethod` functions receive an implicit argument, a method ID, a class object (which must correspond to a superclass of the implicit argument), and explicit arguments. The function calls the version of the method in the specified class, bypassing the normal dynamic dispatch mechanism.

All call functions have versions with suffixes "A" and "V" that receive the explicit parameters in an array or a `va_list` (as defined in the C header `stdarg.h`).

---

**Listing 12.14** printf3/Printf3Test.java

---

```
1 import java.io.*;
2
3 /**
4 * @version 1.10 1997-07-01
5 * @author Cay Horstmann
6 */
7 class Printf3Test
8 {
9 public static void main(String[] args)
10 {
11 double price = 44.95;
12 double tax = 7.75;
13 double amountDue = price * (1 + tax / 100);
```

```
14 PrintWriter out = new PrintWriter(System.out);
15 Printf3.fprintf(out, "Amount due = %8.2f\n", amountDue);
16 out.flush();
17 }
18 }
```

**Listing 12.15** printf3/Printf3.java

```
1 import java.io.*;
2
3 /**
4 * @version 1.10 1997-07-01
5 * @author Cay Horstmann
6 */
7 class Printf3
8 {
9 public static native void fprintf(PrintWriter out, String format, double x);
10
11 static
12 {
13 System.loadLibrary("Printf3");
14 }
15 }
```

**Listing 12.16** printf3/Printf3.c

```
1 /**
2 * @version 1.10 1997-07-01
3 * @author Cay Horstmann
4 */
5
6 #include "Printf3.h"
7 #include <string.h>
8 #include <stdlib.h>
9 #include <float.h>
10
11 /**
12 * @param format a string containing a printf format specifier
13 * (such as "%8.2f"). Substrings "%" are skipped.
14 * @return a pointer to the format specifier (skipping the '%')
15 * or NULL if there wasn't a unique format specifier
16 */
17 char* find_format(const char format[])
18 {
19 char* p;
20 char* q;
```

(Continues)

**Listing 12.16 (Continued)**

```
22 p = strchr(format, '%');
23 while (p != NULL && *(p + 1) == '%') /* skip %% */
24 p = strchr(p + 2, '%');
25 if (p == NULL) return NULL;
26 /* now check that % is unique */
27 p++;
28 q = strchr(p, '%');
29 while (q != NULL && *(q + 1) == '%') /* skip %% */
30 q = strchr(q + 2, '%');
31 if (q != NULL) return NULL; /* % not unique */
32 q = p + strspn(p, " -0+#"); /* skip past flags */
33 q += strspn(q, "0123456789"); /* skip past field width */
34 if (*q == '.') { q++; q += strspn(q, "0123456789"); }
35 /* skip past precision */
36 if (strchr("eEfFgG", *q) == NULL) return NULL;
37 /* not a floating-point format */
38 return p;
39 }
40
41 JNIEXPORT void JNICALL Java_Printf3_fprint(JNIEnv* env, jclass cl,
42 jobject out, jstring format, jdouble x)
43 {
44 const char* cformat;
45 char* fmt;
46 jstring str;
47 jclass class_PrintWriter;
48 jmethodID id_print;
49
50 cformat = (*env)->GetStringUTFChars(env, format, NULL);
51 fmt = find_format(cformat);
52 if (fmt == NULL)
53 str = format;
54 else
55 {
56 char* cstr;
57 int width = atoi(fmt);
58 if (width == 0) width = DBL_DIG + 10;
59 cstr = (char*) malloc(strlen(cformat) + width);
60 sprintf(cstr, cformat, x);
61 str = (*env)->NewStringUTF(env, cstr);
62 free(cstr);
63 }
64 (*env)->ReleaseStringUTFChars(env, format, cformat);
65
66 /* now call ps.print(str) */
67
68 /* get the class */
69 class_PrintWriter = (*env)->GetObjectClass(env, out);
```

```
70
71 /* get the method ID */
72 id_print = (*env)->GetMethodID(env, class_PrintWriter, "print", "(Ljava/lang/String;)V");
73
74 /* call the method */
75 (*env)->CallVoidMethod(env, out, id_print, str);
76 }
```

### Executing Java Methods

- `jmethodID GetMethodID(JNIEnv *env, jclass cl, const char name[], const char methodSignature[])` returns the identifier of a method in a class.
- `Xxx CallXxxMethod(JNIEnv *env, jobject obj, jmethodID id, args)`
- `Xxx CallXxxMethodA(JNIEnv *env, jobject obj, jmethodID id, jvalue args[])`
- `Xxx CallXxxMethodV(JNIEnv *env, jobject obj, jmethodID id, va_list args)` calls a method. The return type `Xxx` is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of `jvalue`, where `jvalue` is a union defined as

```
typedef union jvalue
{
 jboolean z;
 jbyte b;
 jchar c;
 jshort s;
 jint i;
 jlong j;
 jfloat f;
 jdouble d;
 jobject l;
} jvalue;
```

The third function receives the method parameters in a `va_list`, as defined in the C header `stdarg.h`.

- `Xxx CallNonvirtualXxxMethod(JNIEnv *env, jobject obj, jclass cl, jmethodID id, args)`
- `Xxx CallNonvirtualXxxMethodA(JNIEnv *env, jobject obj, jclass cl, jmethodID id, jvalue args[])`
- `Xxx CallNonvirtualXxxMethodV(JNIEnv *env, jobject obj, jclass cl, jmethodID id, va_list args)`

calls a method, bypassing dynamic dispatch. The return type `Xxx` is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, or `Double`. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of `jvalue`. The third function receives the method parameters in a `va_list`, as defined in the C header `stdarg.h`.

(Continues)

**Executing Java Methods (Continued)**

- `jmethodID GetStaticMethodID(JNIEnv *env, jclass cl, const char name[], const char methodSignature[])`  
returns the identifier of a static method in a class.
- `Xxx CallStaticXxxMethod(JNIEnv *env, jclass cl, jmethodID id, args)`
- `Xxx CallStaticXxxMethodA(JNIEnv *env, jclass cl, jmethodID id, jvalue args[])`
- `Xxx CallStaticXxxMethodV(JNIEnv *env, jclass cl, jmethodID id, va_list args)`  
calls a static method. The return type Xxx is one of Object, Boolean, Byte, Char, Short, Int, Long, Float, or Double. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of jvalue. The third function receives the method parameters in a va\_list, as defined in the C header stdarg.h.
- `jobject NewObject(JNIEnv *env, jclass cl, jmethodID id, args)`
- `jobject NewObjectA(JNIEnv *env, jclass cl, jmethodID id, jvalue args[])`
- `jobject NewObjectV(JNIEnv *env, jclass cl, jmethodID id, va_list args)`  
calls a constructor. The method ID is obtained from GetMethodID with a method name of "<init>" and a return type of void. The first function has a variable number of arguments—simply append the method parameters after the method ID. The second function receives the method arguments in an array of jvalue. The third function receives the method parameters in a va\_list, as defined in the C header stdarg.h.

## 12.7 Accessing Array Elements

All array types of the Java programming language have corresponding C types, as shown in Table 12.2.

**Table 12.2** Correspondence between Java Array Types and C Types

| Java Type              | C Type                     | Java Type             | C Type                    |
|------------------------|----------------------------|-----------------------|---------------------------|
| <code>boolean[]</code> | <code>jbooleanArray</code> | <code>long[]</code>   | <code>jlongArray</code>   |
| <code>byte[]</code>    | <code>jbyteArray</code>    | <code>float[]</code>  | <code>jfloatArray</code>  |
| <code>char[]</code>    | <code>jcharArray</code>    | <code>double[]</code> | <code>jdoubleArray</code> |
| <code>int[]</code>     | <code>jintArray</code>     | <code>Object[]</code> | <code>jobjectArray</code> |
| <code>short[]</code>   | <code>jshortArray</code>   |                       |                           |



**C++ NOTE:** In C, all these array types are actually type synonyms of `jobject`. In C++, however, they are arranged in the inheritance hierarchy shown in Figure 12.3. The type `jarray` denotes a generic array.

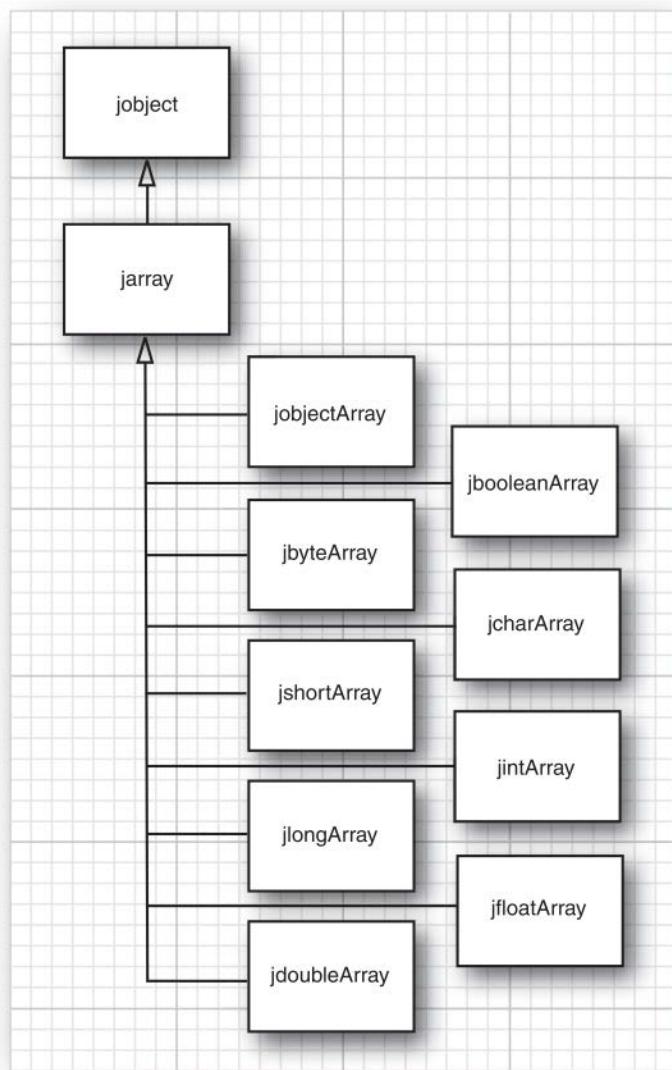


Figure 12.3 Inheritance hierarchy of array types

The `GetArrayLength` function returns the length of an array.

```
jarray array = . . .;
jsize length = (*env)->GetArrayLength(env, array);
```

How you access elements in an array depends on whether the array stores objects or values of a primitive type (`bool`, `char`, or a numeric type). To access elements in an object array, use the `GetObjectArrayElement` and `SetObjectArrayElement` methods.

```
jobjectArray array = . . .;
int i, j;
jobject x = (*env)->GetObjectArrayElement(env, array, i);
(*env)->SetObjectArrayElement(env, array, j, x);
```

Although simple, this approach is also clearly inefficient; you want to be able to access array elements directly, especially when doing vector and matrix computations.

The `GetXxxArrayElements` function returns a C pointer to the starting element of an array. As with ordinary strings, you must remember to call the corresponding `ReleaseXxxArrayElements` function to tell the virtual machine when you no longer need that pointer. Here, the type `Xxx` must be a primitive type—that is, not `Object`. You can then read and write the array elements directly. However, since the pointer *might point to a copy*, any changes that you make are guaranteed to be reflected in the original array only after you call the corresponding `ReleaseXxxArrayElements` function!

---

**NOTE:** You can find out if an array is a copy by passing a pointer to a `jboolean` variable as the third parameter to a `GetXxxArrayElements` method. The variable is filled with `JNI_TRUE` if the array is a copy. If you aren't interested in that information, just pass a `NULL` pointer.

---

Here is an example that multiplies all elements in an array of `double` values by a constant. We obtain a C pointer `a` into the Java array and then access individual elements as `a[i]`.

```
jdoubleArray array_a = . . .;
double scaleFactor = . . .;
double* a = (*env)->GetDoubleArrayElements(env, array_a, NULL);
for (i = 0; i < (*env)->GetArrayLength(env, array_a); i++)
 a[i] = a[i] * scaleFactor;
(*env)->ReleaseDoubleArrayElements(env, array_a, a, 0);
```

Whether the virtual machine actually copies the array depends on how it allocates arrays and does its garbage collection. Some “copying” garbage collectors routinely move objects around and update object references. That strategy is not

compatible with “pinning” an array to a particular location, because the collector cannot update the pointer values in native code.

---

**NOTE:** In the Oracle JVM implementation, boolean arrays are represented as packed arrays of 32-bit words. The `GetBooleanArrayElements` method copies them into unpacked arrays of `jboolean` values.

---

To access just a few elements of a large array, use the `GetXxxArrayRegion` and `SetXxxArrayRegion` methods that copy a range of elements from the Java array into a C array and back.

You can create new Java arrays in native methods with the `NewXxxArray` function. To create a new array of objects, specify the length, the type of the array elements, and an initial element for all entries (typically, `NULL`). Here is an example:

```
jclass class_Employee = (*env)->FindClass(env, "Employee");
jobjectArray array_e = (*env)->NewObjectArray(env, 100, class_Employee, NULL);
```

Arrays of primitive types are simpler: just supply the length of the array.

```
jdoubleArray array_d = (*env)->NewDoubleArray(env, 100);
```

The array is then filled with zeroes.

---

**NOTE:** The following methods are used for working with “direct buffers”:

```
jobject NewDirectByteBuffer(JNIEnv* env, void* address, jlong capacity)
void* GetDirectBufferAddress(JNIEnv* env, jobject buf)
jlong GetDirectBufferCapacity(JNIEnv* env, jobject buf)
```

Direct buffers are used in the `java.nio` package to support more efficient input/output operations and to minimize the copying of data between native and Java arrays.

---

#### Manipulating Java Arrays

- `jsize GetArrayLength(JNIEnv *env, jarray array)`  
returns the number of elements in the array.
- `jobject GetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index)`  
returns the value of an array element.

(Continues)

**Manipulating Java Arrays (Continued)**

- `void SetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index, jobject value)`  
sets an array element to a new value.
- `Xxx* GetXxxArrayElements(JNIEnv *env, jarray array, jboolean* isCopy)`  
yields a C pointer to the elements of a Java array. The field type Xxx is one of Boolean, Byte, Char, Short, Int, Long, Float, or Double. The pointer must be passed to `ReleaseXxxArrayElements` when it is no longer needed. `isCopy` is either NULL or points to a `jboolean` that is filled with `JNI_TRUE` if a copy is made, with `JNI_FALSE` otherwise.
- `void ReleaseXxxArrayElements(JNIEnv *env, jarray array, Xxx elems[], jint mode)`  
notifies the virtual machine that a pointer obtained by `GetXxxArrayElements` is no longer needed. `mode` is one of 0 (free the `elems` buffer after updating the array elements), `JNI_COMMIT` (do not free the `elems` buffer after updating the array elements), or `JNI_ABORT` (free the `elems` buffer without updating the array elements).
- `void GetXxxArrayRegion(JNIEnv *env, jarray array, jint start, jint length, Xxx elems[])`  
copies elements from a Java array to a C array. The field type Xxx is one of Boolean, Byte, Char, Short, Int, Long, Float, or Double.
- `void SetXxxArrayRegion(JNIEnv *env, jarray array, jint start, jint length, Xxx elems[])`  
copies elements from a C array to a Java array. The field type Xxx is one of Boolean, Byte, Char, Short, Int, Long, Float, or Double.

## 12.8 Handling Errors

Native methods are a significant security risk to Java programs. The C runtime system has no protection against array bounds errors, indirection through bad pointers, and so on. It is particularly important that programmers of native methods handle all error conditions to preserve the integrity of the Java platform. In particular, when your native method diagnoses a problem it cannot handle, it should report this problem to the Java virtual machine.

Normally, you would throw an exception in this situation. However, C has no exceptions. Instead, you must call the `Throw` or `ThrowNew` function to create a new exception object. When the native method exits, the Java virtual machine throws that exception.

To use the `Throw` function, call `NewObject` to create an object of a subtype of `Throwable`. For example, here we allocate an `EOFException` object and throw it:

```
jclass class_EOFException = (*env)->FindClass(env, "java/io/EOFException");
jmethodID id_EOFException = (*env)->GetMethodID(env, class_EOFException, "<init>", "()V");
/* ID of no-argument constructor */
```

```
jthrowable obj_exc = (*env)->NewObject(env, class_EOFException, id_EOFException);
(*env)->Throw(env, obj_exc);
```

It is usually more convenient to call `ThrowNew`, which constructs an exception object, given a class and a “modified UTF-8” byte sequence.

```
(*env)->ThrowNew(env, (*env)->FindClass(env, "java/io/EOFException"), "Unexpected end of file");
```

Both `Throw` and `ThrowNew` merely *post* the exception; they do not interrupt the control flow of the native method. Only when the method returns does the Java virtual machine throw the exception. Therefore, every call to `Throw` and `ThrowNew` should always be immediately followed by a `return` statement.



**C++ NOTE:** If you implement native methods in C++, you cannot throw a Java exception object in your C++ code. In a C++ binding, it would be possible to implement a translation between exceptions in C++ and Java; however, this is not currently done. Use `Throw` or `ThrowNew` to throw a Java exception in a native C++ method, and make sure your native methods throw no C++ exceptions.

Normally, native code need not be concerned with catching Java exceptions. However, when a native method calls a Java method, that method might throw an exception. Moreover, a number of the JNI functions throw exceptions as well. For example, `SetObjectArrayElement` throws an `ArrayIndexOutOfBoundsException` if the index is out of bounds, and an `ArrayStoreException` if the class of the stored object is not a subclass of the element class of the array. In situations like these, a native method should call the `ExceptionOccurred` method to determine whether an exception has been thrown. The call

```
jthrowable obj_exc = (*env)->ExceptionOccurred(env);
```

returns `NULL` if no exception is pending, or a reference to the current exception object. If you just want to check whether an exception has been thrown, without obtaining a reference to the exception object, use

```
jboolean occurred = (*env)->ExceptionCheck(env);
```

Normally, a native method should simply return when an exception has occurred so that the virtual machine can propagate it to the Java code. However, a native method *may* analyze the exception object to determine if it can handle the exception. If it can, then the function

```
(*env)->ExceptionClear(env);
```

must be called to turn off the exception.

In our next example, we implement the `fprint` native method with all the paranoia appropriate for a native method. Here are the exceptions that we throw:

- A `NullPointerException` if the format string is `NULL`
- An `IllegalArgumentException` if the format string doesn't contain a `%` specifier that is appropriate for printing a `double`
- An `OutOfMemoryError` if the call to `malloc` fails

Finally, to demonstrate how to check for an exception when calling a Java method from a native method, we send the string to the stream, a character at a time, and call `ExceptionOccurred` after each call. Listing 12.17 shows the code for the native method, and Listing 12.18 shows the definition of the class containing the native method. Notice that the native method does not immediately terminate when an exception occurs in the call to `PrintWriter.print`—it first frees the `cstr` buffer. When the native method returns, the virtual machine again raises the exception. The test program in Listing 12.19 demonstrates how the native method throws an exception when the formatting string is not valid.

---

**Listing 12.17** printf4/Printf4.c

---

```
1 /**
2 * @version 1.10 1997-07-01
3 * @author Cay Horstmann
4 */
5
6 #include "Printf4.h"
7 #include <string.h>
8 #include <stdlib.h>
9 #include <float.h>
10
11 /**
12 * @param format a string containing a printf format specifier
13 * (such as "%8.2f"). Substrings "%" are skipped.
14 * @return a pointer to the format specifier (skipping the '%')
15 * or NULL if there wasn't a unique format specifier
16 */
17 char* find_format(const char format[])
18 {
19 char* p;
20 char* q;
21
22 p = strchr(format, '%');
23 while (p != NULL && *(p + 1) == '%') /* skip %% */
24 p = strchr(p + 2, '%');
25 if (p == NULL) return NULL;
26 /* now check that % is unique */
27 p++;
```

```
28 q = strchr(p, '%');
29 while (q != NULL && *(q + 1) == '%') /* skip %% */
30 {
31 q = strchr(q + 2, '%');
32 if (q != NULL) return NULL; /* % not unique */
33 q = p + strspn(p, " -0+#"); /* skip past flags */
34 q += strspn(q, "0123456789"); /* skip past field width */
35 if (*q == '.') { q++; q += strspn(q, "0123456789"); }
36 /* skip past precision */
37 if (strchr("eEfFgG", *q) == NULL) return NULL;
38 /* not a floating-point format */
39 }
40
41 JNIEXPORT void JNICALL Java_Printf4_fprint(JNIEnv* env, jclass cl,
42 jobject out, jstring format, jdouble x)
43 {
44 const char* cformat;
45 char* fmt;
46 jclass class_PrintWriter;
47 jmethodID id_print;
48 char* cstr;
49 int width;
50 int i;
51
52 if (format == NULL)
53 {
54 (*env)->ThrowNew(env,
55 (*env)->FindClass(env,
56 "java/lang/NullPointerException"),
57 "Printf4.fprint: format is null");
58 return;
59 }
60
61 cformat = (*env)->GetStringUTFChars(env, format, NULL);
62 fmt = find_format(cformat);
63
64 if (fmt == NULL)
65 {
66 (*env)->ThrowNew(env,
67 (*env)->FindClass(env,
68 "java/lang/IllegalArgumentException"),
69 "Printf4.fprint: format is invalid");
70 return;
71 }
72
73 width = atoi(fmt);
74 if (width == 0) width = DBL_DIG + 10;
75 cstr = (char*)malloc(strlen(cformat) + width);
76
```

(Continues)

**Listing 12.17 (Continued)**

```
77 if (cstr == NULL)
78 {
79 (*env)->ThrowNew(env,
80 (*env)->FindClass(env, "java/lang/OutOfMemoryError"),
81 "Printf4.fprintf: malloc failed");
82 return;
83 }
84
85 sprintf(cstr, cformat, x);
86
87 (*env)->ReleaseStringUTFChars(env, format, cformat);
88
89 /* now call ps.print(str) */
90
91 /* get the class */
92 class_PrintWriter = (*env)->GetObjectClass(env, out);
93
94 /* get the method ID */
95 id_print = (*env)->GetMethodID(env, class_PrintWriter, "print", "(C)V");
96
97 /* call the method */
98 for (i = 0; cstr[i] != 0 && !(*env)->ExceptionOccurred(env); i++)
99 (*env)->CallVoidMethod(env, out, id_print, cstr[i]);
100
101 free(cstr);
102 }
```

---

**Listing 12.18 printf4/Printf4.java**

```
1 import java.io.*;
2
3 /**
4 * @version 1.10 1997-07-01
5 * @author Cay Horstmann
6 */
7 class Printf4
8 {
9 public static native void fprintf(PrintWriter ps, String format, double x);
10
11 static
12 {
13 System.loadLibrary("Printf4");
14 }
15 }
```

---

**Listing 12.19** printf4/Printf4Test.java

```
1 import java.io.*;
2 /**
3 * @version 1.10 1997-07-01
4 * @author Cay Horstmann
5 */
6 class Printf4Test
7 {
8 public static void main(String[] args)
9 {
10 double price = 44.95;
11 double tax = 7.75;
12 double amountDue = price * (1 + tax / 100);
13 PrintWriter out = new PrintWriter(System.out);
14 /* This call will throw an exception--note the %% */
15 Printf4.fprintf(out, "Amount due = %%8.2f\n", amountDue);
16 out.flush();
17 }
18 }
19 }
```

**Handling Java Exceptions**

- **jint Throw(JNIEnv \*env, jthrowable obj)**  
prepares an exception to be thrown upon exiting from the native code. Returns 0 on success, a negative value on failure.
- **jint ThrowNew(JNIEnv \*env, jclass cl, const char msg[])**  
prepares an exception of type *cl* to be thrown upon exiting from the native code. Returns 0 on success, a negative value on failure. *msg* is a “modified UTF-8” byte sequence denoting the String construction argument of the exception object.
- **jthrowable ExceptionOccurred(JNIEnv \*env)**  
returns the exception object if an exception is pending, or NULL otherwise.
- **jboolean ExceptionCheck(JNIEnv \*env)**  
returns true if an exception is pending.
- **void ExceptionClear(JNIEnv \*env)**  
clears any pending exceptions.

## 12.9 Using the Invocation API

Up to now, we have considered programs in the Java programming language that made a few C calls, presumably because C was faster or allowed access to functionality inaccessible from the Java platform. Suppose you are in the opposite situation. You have a C or C++ program and would like to make calls to Java code. The *invocation API* enables you to embed the Java virtual machine into a C or C++ program. Here is the minimal code that you need to initialize a virtual machine:

```
JavaVMOption options[1];
JavaVMInitArgs vm_args;
JavaVM *jvm;
JNIEnv *env;

options[0].optionString = "-Djava.class.path=.";

memset(&vm_args, 0, sizeof(vm_args));
vm_args.version = JNI_VERSION_1_2;
vm_args.nOptions = 1;
vm_args.options = options;

JNI_CreateJavaVM(&jvm, (void**) &env, &vm_args);
```

The call to `JNI_CreateJavaVM` creates the virtual machine and fills in a pointer `jvm` to the virtual machine and a pointer `env` to the execution environment.

You can supply any number of options to the virtual machine. Simply increase the size of the `options` array and the value of `vm_args.nOptions`. For example,

```
options[i].optionString = "-Djava.compiler=None";
deactivates the just-in-time compiler.
```



**TIP:** When you run into trouble and your program crashes, refuses to initialize the JVM, or can't load your classes, turn on the JNI debugging mode. Set an option to

```
options[i].optionString = "-verbose:jni";
```

You will see a flurry of messages that indicate the progress in initializing the JVM. If you don't see your classes loaded, check both your path and class path settings.

---

Once you have set up the virtual machine, you can call Java methods as described in the preceding sections. Simply use the `env` pointer in the usual way.

You'll need the `jvm` pointer only to call other functions in the invocation API. Currently, there are only four such functions. The most important one is the function to terminate the virtual machine:

```
(*jvm)->DestroyJavaVM(jvm);
```

Unfortunately, under Windows, it has become difficult to dynamically link to the `JNI_CreateJavaVM` function in the `jre/bin/client/jvm.dll` library, due to the changed linking rules in Vista and Oracle's reliance on an older C runtime library. Our sample program overcomes this problem by loading the library manually. This is the same approach used by the `java` program—see the file `launcher/java_md.c` in the `src.jar` file that is a part of the JDK.

The C program in Listing 12.20 sets up a virtual machine and calls the `main` method of the `Welcome` class, which was discussed in Volume I, Chapter 2. (Make sure to compile the `Welcome.java` file before starting the invocation test program.)

#### **Listing 12.20** invocation/InvocationTest.c

```
1 /**
2 * @version 1.20 2007-10-26
3 * @author Cay Horstmann
4 */
5
6 #include <jni.h>
7 #include <stdlib.h>
8
9 #ifdef _WINDOWS
10
11 #include <windows.h>
12 static HINSTANCE LoadJVMLibrary(void);
13 typedef jint (JNICALL *CreateJavaVM_t)(JavaVM **, void **, JavaVMInitArgs *);
14
15 #endif
16
17 int main()
18 {
19 JavaVMOption options[2];
20 JavaVMInitArgs vm_args;
21 JavaVM *jvm;
22 JNIEnv *env;
23 long status;
24
25 jclass class_Welcome;
26 jclass class_String;
27 jobjectArray args;
```

(Continues)

**Listing 12.20 (Continued)**

```
28 jmethodID id_main;
29
30 #ifdef _WINDOWS
31 HINSTANCE hJvmLib;
32 CreateJavaVM_t createJavaVM;
33 #endif
34
35 options[0].optionString = "-Djava.class.path=.";
36
37 memset(&vm_args, 0, sizeof(vm_args));
38 vm_args.version = JNI_VERSION_1_2;
39 vm_args.nOptions = 1;
40 vm_args.options = options;
41
42
43 #ifdef _WINDOWS
44 hJvmLib = loadJVMLibrary();
45 createJavaVM = (CreateJavaVM_t) GetProcAddress(hJvmLib, "JNI_CreateJavaVM");
46 status = (*createJavaVM)(&jVm, (void **) &env, &vm_args);
47 #else
48 status = JNI_CreateJavaVM(&jVm, (void **) &env, &vm_args);
49 #endif
50
51 if (status == JNI_ERR)
52 {
53 fprintf(stderr, "Error creating VM\n");
54 return 1;
55 }
56
57 class_Welcome = (*env)->FindClass(env, "Welcome");
58 id_main = (*env)->GetStaticMethodID(env, class_Welcome, "main", "([Ljava/lang/String;)V");
59
60 class_String = (*env)->FindClass(env, "java/lang/String");
61 args = (*env)->NewObjectArray(env, 0, class_String, NULL);
62 (*env)->CallStaticVoidMethod(env, class_Welcome, id_main, args);
63
64 (*jVm)->DestroyJavaVM(jVm);
65
66 return 0;
67 }
68
69 #ifdef _WINDOWS
70
71 static int GetStringFromRegistry(HKEY key, const char *name, char *buf, jint bufsize)
72 {
73 DWORD type, size;
```

```
74 return RegQueryValueEx(key, name, 0, &type, 0, &size) == 0
75 && type == REG_SZ
76 && size < (unsigned int) bufsize
77 && RegQueryValueEx(key, name, 0, 0, buf, &size) == 0;
78 }
79 }

80 static void GetPublicJREHome(char *buf, jint bufsize)
81 {
82 HKEY key, subkey;
83 char version[MAX_PATH];
84
85 /* Find the current version of the JRE */
86 char *JRE_KEY = "Software\\JavaSoft\\Java Runtime Environment";
87 if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, JRE_KEY, 0, KEY_READ, &key) != 0)
88 {
89 fprintf(stderr, "Error opening registry key '%s'\n", JRE_KEY);
90 exit(1);
91 }
92
93 if (!GetStringFromRegistry(key, "CurrentVersion", version, sizeof(version)))
94 {
95 fprintf(stderr, "Failed reading value of registry key:\n\t%s\\CurrentVersion\n", JRE_KEY);
96 RegCloseKey(key);
97 exit(1);
98 }
99
100 /* Find directory where the current version is installed. */
101 if (RegOpenKeyEx(key, version, 0, KEY_READ, &subkey) != 0)
102 {
103 fprintf(stderr, "Error opening registry key '%s\\%s'\n", JRE_KEY, version);
104 RegCloseKey(key);
105 exit(1);
106 }
107
108 if (!GetStringFromRegistry(subkey, "JavaHome", buf, bufsize))
109 {
110 fprintf(stderr, "Failed reading value of registry key:\n\t%s\\%s\\JavaHome\n",
111 JRE_KEY, version);
112 RegCloseKey(key);
113 RegCloseKey(subkey);
114 exit(1);
115 }
116
117 RegCloseKey(key);
118 RegCloseKey(subkey);
119 }
120 }
```

(Continues)

**Listing 12.20 (Continued)**

```
122 static HINSTANCE loadJVMLibrary(void)
123 {
124 HINSTANCE h1, h2;
125 char msrvcdll[MAX_PATH];
126 char javadll[MAX_PATH];
127 GetPublicJREHome(msrvcdll, MAX_PATH);
128 strcpy(javadll, msrvcdll);
129 strncat(msrvcdll, "\\bin\\msvcr71.dll", MAX_PATH - strlen(msrvcdll));
130 msrvcdll[MAX_PATH - 1] = '\0';
131 strncat(javadll, "\\bin\\client\\jvm.dll", MAX_PATH - strlen(javadll));
132 javadll[MAX_PATH - 1] = '\0';
133
134 h1 = LoadLibrary(msrvcdll);
135 if (h1 == NULL)
136 {
137 fprintf(stderr, "Can't load library msvcr71.dll\n");
138 exit(1);
139 }
140
141 h2 = LoadLibrary(javadll);
142 if (h2 == NULL)
143 {
144 fprintf(stderr, "Can't load library jvm.dll\n");
145 exit(1);
146 }
147 return h2;
148 }
149
150 #endif
```

---

To compile this program under Linux, use

```
gcc -I jdk/include -I jdk/include/linux -o InvocationTest
 -L jdk/jre/lib/i386/client -ljvm InvocationTest.c
```

Under Solaris, use

```
cc -I jdk/include -I jdk/include/solaris -o InvocationTest
 -L jdk/jre/lib/sparc -ljvm InvocationTest.c
```

When compiling in Windows with the Microsoft compiler, use the command line

```
c1 -D_WINDOWS -I jdk\include -I jdk\include\win32 InvocationTest.c jdk\lib\jvm.lib advapi32.lib
```

You will need to make sure that the INCLUDE and LIB environment variables include the paths to the Windows API header and library files.

Using Cygwin, compile with

```
gcc -D_WINDOWS -mno-cygwin -I jdk\include -I jdk\include\win32 -D_int64="long long"
-I c:\cygwin\usr\include\w32api -o InvocationTest
```

Before you run the program under Linux/UNIX, make sure that the `LD_LIBRARY_PATH` contains the directories for the shared libraries. For example, if you use the `bash` shell on Linux, issue the following command:

```
export LD_LIBRARY_PATH=jdk/jre/lib/i386/client:$LD_LIBRARY_PATH
```

#### Invocation API Functions

- `jint JNI_CreateJavaVM(JavaVM** p_jvm, void** p_env, JavaVMInitArgs* vm_args)`  
initializes the Java virtual machine. The function returns 0 if successful, `JNI_ERR` on failure.  
*Parameters:* `p_jvm` Filled with a pointer to the invocation API function table  
`p_env` Filled with a pointer to the JNI function table  
`vm_args` The virtual machine arguments
- `jint DestroyJavaVM(JavaVM* jvm)`  
destroys the virtual machine. Returns 0 on success, a negative number on failure. This function must be called through a virtual machine pointer, that is, `(*jvm)->DestroyJavaVM(jvm)`.

## 12.10 A Complete Example: Accessing the Windows Registry

In this section, we describe a full, working example that covers everything we discussed in this chapter: using native methods with strings, arrays, objects, constructor calls, and error handling. We'll show you how to put a Java platform wrapper around a subset of the ordinary C-based APIs used to work with the Windows registry. Of course, the Windows registry being a Windows-specific feature, such a program is inherently nonportable. For that reason, the standard Java library has no support for the registry, and it makes sense to use native methods to gain access to it.

### 12.10.1 Overview of the Windows Registry

The Windows registry is a data depository that holds configuration information for the Windows operating system and application programs. It provides a single point for administration and backup of system and application preferences. On the downside, the registry is also a single point of failure—if you mess up the registry, your computer could malfunction or even fail to boot!

We don't suggest that you use the registry to store configuration parameters for your Java programs. The Java preferences API is a better solution (see Volume I, Chapter 13 for more information). We'll simply use the registry to demonstrate how to wrap a nontrivial native API into a Java class.

The principal tool for inspecting the registry is the *registry editor*. Because of the potential for error by naive but enthusiastic users, there is no icon for launching the registry editor. Instead, start a DOS shell (or open the Start → Run dialog box) and type `regedit`. Figure 12.4 shows the registry editor in action.

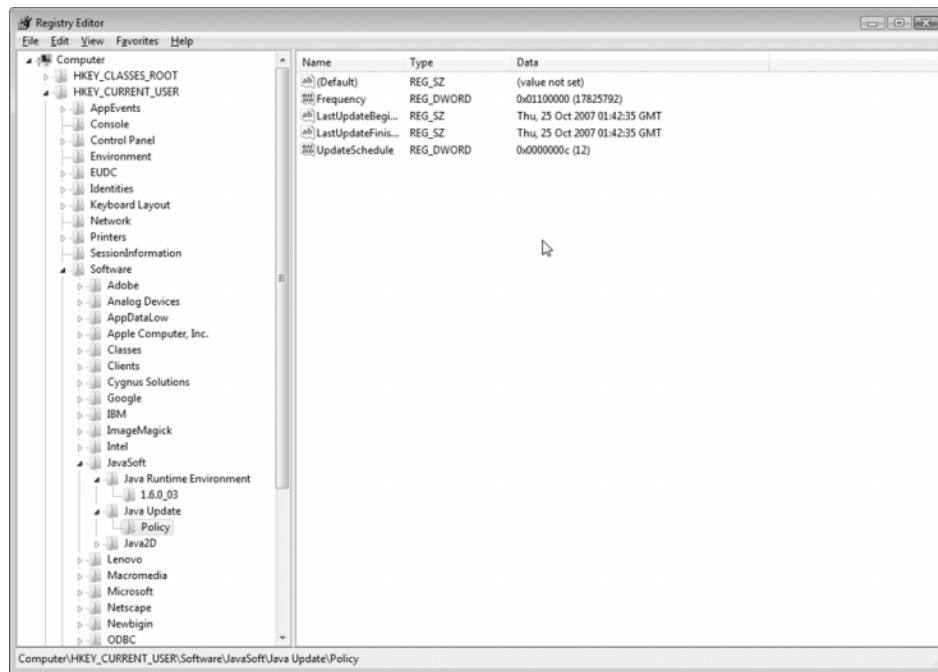


Figure 12.4 The registry editor

The left side shows the keys, which are arranged in a tree structure. Note that each key starts with one of the HKEY nodes like

```
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
...
...
```

The right side shows the name/value pairs associated with a particular key. For example, if you installed Java SE 7, the key

HKEY\_LOCAL\_MACHINE\Software\JavaSoft\Java Runtime Environment

contains a name/value pair such as

CurrentVersion="1.7.0\_10"

In this case, the value is a string. The values can also be integers or arrays of bytes.

### 12.10.2 A Java Platform Interface for Accessing the Registry

We create a simple interface to access the registry from Java code, and then implement this interface with native code. Our interface allows only a few registry operations; to keep the code size down, we omitted some important operations such as adding, deleting, and enumerating keys. (It should be easy to add the remaining registry API functions.)

Even with the limited subset that we supply, you can

- Enumerate all names stored in a key
- Read the value stored with a name
- Set the value stored with a name

Here is the Java class that encapsulates a registry key:

```
public class Win32RegKey
{
 public Win32RegKey(int theRoot, String thePath) { . . . }
 public Enumeration names() { . . . }
 public native Object getValue(String name);
 public native void setValue(String name, Object value);

 public static final int HKEY_CLASSES_ROOT = 0x80000000;
 public static final int HKEY_CURRENT_USER = 0x80000001;
 public static final int HKEY_LOCAL_MACHINE = 0x80000002;
 . .
}
```

The `names` method returns an enumeration that holds all the names stored with the key. You can get at them with the familiar `hasMoreElements/nextElement` methods. The `getValue` method returns an object that is either a string, an `Integer` object, or a byte array. The `value` parameter of the `setValue` method must also be of one of these three types.

### 12.10.3 Implementation of Registry Access Functions as Native Methods

We need to implement three actions:

- Get the value of a name
- Set the value of a name
- Iterate through the names of a key

Fortunately, you have seen essentially all the tools that are required, such as the conversion between Java strings and arrays and those of C. You also saw how to raise a Java exception in case something goes wrong.

Two issues make these native methods more complex than the preceding examples. The `getValue` and `setValue` methods deal with the type `Object`, which can be one of `String`, `Integer`, or `byte[]`. The enumeration object stores the state between successive calls to `hasMoreElements` and `nextElement`.

Let us first look at the `getValue` method. The method (shown in Listing 12.22) goes through the following steps:

1. Opens the registry key. To read their values, the registry API requires that keys be open.
2. Queries the type and size of the value associated with the name.
3. Reads the data into a buffer.
4. Calls `NewStringUTF` to create a new string with the value data if the type is `REG_SZ` (a string).
5. Invokes the `Integer` constructor if the type is `REG_DWORD` (a 32-bit integer).
6. Calls `NewByteArray` to create a new byte array, then `SetByteArrayRegion` to copy the value data into the byte array, if the type is `REG_BINARY`.
7. If the type is none of these or if an error occurred when an API function was called, throws an exception and releases all resources that had been acquired up to that point.
8. Closes the key and returns the object (`String`, `Integer`, or `byte[]`) that had been created.

As you can see, this example illustrates quite nicely how to generate Java objects of different types.

In this native method, coping with the generic return type is not difficult. The `jstring`, `jobject`, or `jarray` reference is simply returned as a `jobject`. However, the `setValue` method receives a reference to an `Object` and must determine the `Object`'s exact type to save the `Object` as a `String`, `integer`, or `byte array`. We can make this determination by querying the class of the `value` object, finding the class references for

`java.lang.String`, `java.lang.Integer`, and `byte[]`, and comparing them with the `IsAssignableFrom` function.

If `class1` and `class2` are two class references, then the call

```
(*env)->IsAssignableFrom(env, class1, class2)
```

returns `JNI_TRUE` when `class1` and `class2` are the same class or when `class1` is a subclass of `class2`. In either case, references to objects of `class1` can be cast to `class2`. For example, when

```
(*env)->IsAssignableFrom(env, (*env)->GetObjectClass(env, value), (*env)->FindClass(env, "[B"))
```

is true, we know that `value` is a byte array.

Here is an overview of the steps in the `setValue` method:

1. Open the registry key for writing.
2. Find the type of the value to write.
3. Call `GetStringUTFChars` to get a pointer to the characters if the type is `String`.
4. Call the `intValue` method to get the integer stored in the wrapper object if the type is `Integer`.
5. Call `GetByteArrayElements` to get a pointer to the bytes if the type is `byte[]`.
6. Pass the data and length to the registry.
7. Close the key.
8. Release the pointer to the data if the type is `String` or `byte[]`.

Finally, let us turn to the native methods that enumerate keys. These are methods of the `Win32RegKeyNameEnumeration` class (see Listing 12.21). When the enumeration process starts, we must open the key. For the duration of the enumeration, we must retain the key handle—that is, the key handle must be stored with the enumeration object. The key handle is of type `DWORD` (a 32-bit quantity), so it can be stored in a Java integer. We store it in the `hkey` field of the enumeration class. When the enumeration starts, the field is initialized with `SetIntField`. Subsequent calls read the value with `GetIntField`.

In this example, we store three other data items with the enumeration object. When the enumeration first starts, we can query the registry for the count of name/value pairs and the length of the longest name, which we need so we can allocate C character arrays to hold the names. These values are stored in the `count` and `maxsize` fields of the enumeration object. Finally, the `index` field, initialized with -1 to indicate the start of the enumeration, is set to 0 once the other instance fields are initialized, and is incremented after every enumeration step.

Let's walk through the native methods that support the enumeration. The `hasMoreElements` method is simple:

1. Retrieve the `index` and `count` fields.
2. If the `index` is `-1`, call the `startNameEnumeration` function, which opens the key, queries the `count` and maximum length, and initializes the `hkey`, `count`, `maxsize`, and `index` fields.
3. Return `JNI_TRUE` if `index` is less than `count`, and `JNI_FALSE` otherwise.

The `nextElement` method needs to work a little harder:

1. Retrieve the `index` and `count` fields.
2. If the `index` is `-1`, call the `startNameEnumeration` function, which opens the key, queries the `count` and maximum length, and initializes the `hkey`, `count`, `maxsize`, and `index` fields.
3. If `index` equals `count`, throw a `NoSuchElementException`.
4. Read the next name from the registry.
5. Increment `index`.
6. If `index` equals `count`, close the key.

Before compiling, remember to run `javah` on both `Win32RegKey` and `Win32RegKeyNameEnumeration`. The complete command line for the Microsoft compiler is

```
c1 -I jdk\include -I jdk\include\win32 -LD Win32RegKey.c advapi32.lib -FeWin32RegKey.dll
```

With Cygwin, use

```
gcc -mno-cygwin -D __int64="long long" -I jdk\include -I jdk\include\win32
-I c:/cygwin/usr/include/w32api -shared -Wl,--add-stdcall-alias -o Win32RegKey.dll
Win32RegKey.c
```

As the registry API is specific to Windows, this program will not work on other operating systems.

Listing 12.23 shows a program to test our new registry functions. We add three name/value pairs, a string, an integer, and a byte array to the key

```
HKEY_CURRENT_USER\Software\JavaSoft\Java Runtime Environment
```

We then enumerate all names of that key and retrieve their values. The program will print

```
Default user=Harry Hacker
Lucky number=13
Small primes=2 3 5 7 11 13
```

Although adding these name/value pairs to that key probably does no harm, you might want to use the registry editor to remove them after running this program.

**Listing 12.21** win32reg/Win32RegKey.java

```
1 import java.util.*;
2
3 /**
4 * A Win32RegKey object can be used to get and set values of a registry key in the Windows
5 * registry.
6 * @version 1.00 1997-07-01
7 * @author Cay Horstmann
8 */
9 public class Win32RegKey
10 {
11 public static final int HKEY_CLASSES_ROOT = 0x80000000;
12 public static final int HKEY_CURRENT_USER = 0x80000001;
13 public static final int HKEY_LOCAL_MACHINE = 0x80000002;
14 public static final int HKEY_USERS = 0x80000003;
15 public static final int HKEY_CURRENT_CONFIG = 0x80000005;
16 public static final int HKEY_DYN_DATA = 0x80000006;
17
18 private int root;
19 private String path;
20
21 /**
22 * Gets the value of a registry entry.
23 * @param name the entry name
24 * @return the associated value
25 */
26 public native Object getValue(String name);
27
28 /**
29 * Sets the value of a registry entry.
30 * @param name the entry name
31 * @param value the new value
32 */
33 public native void setValue(String name, Object value);
34
35 /**
36 * Construct a registry key object.
37 * @param theRoot one of HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS,
38 * HKEY_CURRENT_CONFIG, HKEY_DYN_DATA
39 * @param thePath the registry key path
40 */
41 public Win32RegKey(int theRoot, String thePath)
42 {
43 root = theRoot;
44 path = thePath;
45 }
```

(Continues)

**Listing 12.21 (Continued)**

```
46
47 /**
48 * Enumerates all names of registry entries under the path that this object describes.
49 * @return an enumeration listing all entry names
50 */
51 public Enumeration<String> names()
52 {
53 return new Win32RegKeyNameEnumeration(root, path);
54 }
55
56 static
57 {
58 System.loadLibrary("Win32RegKey");
59 }
60 }
61
62 class Win32RegKeyNameEnumeration implements Enumeration<String>
63 {
64 public native String nextElement();
65 public native boolean hasMoreElements();
66 private int root;
67 private String path;
68 private int index = -1;
69 private int hkey = 0;
70 private int maxsize;
71 private int count;
72
73 Win32RegKeyNameEnumeration(int theRoot, String thePath)
74 {
75 root = theRoot;
76 path = thePath;
77 }
78 }
79
80 class Win32RegKeyException extends RuntimeException
81 {
82 public Win32RegKeyException()
83 {
84 }
85
86 public Win32RegKeyException(String why)
87 {
88 super(why);
89 }
90 }
```

---

**Listing 12.22** win32reg/Win32RegKey.c

```
1 /**
2 * @version 1.00 1997-07-01
3 * @author Cay Horstmann
4 */
5
6 #include "Win32RegKey.h"
7 #include "Win32RegKeyNameEnumeration.h"
8 #include <string.h>
9 #include <stdlib.h>
10 #include <windows.h>
11
12 JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue(JNIEnv* env, jobject this_obj, jobject name)
13 {
14 const char* cname;
15 jstring path;
16 const char* cpath;
17 HKEY hkey;
18 DWORD type;
19 DWORD size;
20 jclass this_class;
21 jfieldID id_root;
22 jfieldID id_path;
23 HKEY root;
24 jobject ret;
25 char* cret;
26
27 /* get the class */
28 this_class = (*env)->GetObjectClass(env, this_obj);
29
30 /* get the field IDs */
31 id_root = (*env)->GetFieldID(env, this_class, "root", "I");
32 id_path = (*env)->GetFieldID(env, this_class, "path", "Ljava/lang/String;");
33
34 /* get the fields */
35 root = (HKEY) (*env)->GetIntField(env, this_obj, id_root);
36 path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
37 cpath = (*env)->GetStringUTFChars(env, path, NULL);
38
39 /* open the registry key */
40 if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) != ERROR_SUCCESS)
41 {
42 (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
43 "Open key failed");
44 (*env)->ReleaseStringUTFChars(env, path, cpath);
45 return NULL;
46 }
```

*(Continues)*

**Listing 12.22 (Continued)**

```
47
48 (*env)->ReleaseStringUTFChars(env, path, cpath);
49 cname = (*env)->GetStringUTFChars(env, name, NULL);
50
51 /* find the type and size of the value */
52 if (RegQueryValueEx(hkey, cname, NULL, &type, NULL, &size) != ERROR_SUCCESS)
53 {
54 (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
55 "Query value key failed");
56 RegCloseKey(hkey);
57 (*env)->ReleaseStringUTFChars(env, name, cname);
58 return NULL;
59 }
60
61 /* get memory to hold the value */
62 cret = (char*)malloc(size);
63
64 /* read the value */
65 if (RegQueryValueEx(hkey, cname, NULL, &type, cret, &size) != ERROR_SUCCESS)
66 {
67 (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
68 "Query value key failed");
69 free(cret);
70 RegCloseKey(hkey);
71 (*env)->ReleaseStringUTFChars(env, name, cname);
72 return NULL;
73 }
74
75 /* depending on the type, store the value in a string,
76 integer or byte array */
77 if (type == REG_SZ)
78 {
79 ret = (*env)->NewStringUTF(env, cret);
80 }
81 else if (type == REG_DWORD)
82 {
83 jclass class_Integer = (*env)->FindClass(env, "java/lang/Integer");
84 /* get the method ID of the constructor */
85 jmethodID id_Integer = (*env)->GetMethodID(env, class_Integer, "<init>", "(I)V");
86 int value = *(int*) cret;
87 /* invoke the constructor */
88 ret = (*env)->NewObject(env, class_Integer, id_Integer, value);
89 }
90 else if (type == REG_BINARY)
91 {
92 ret = (*env)->NewByteArray(env, size);
93 (*env)->SetByteArrayRegion(env, (jarray) ret, 0, size, cret);
94 }
```

```
95 else
96 {
97 (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
98 "Unsupported value type");
99 ret = NULL;
100 }
101
102 free(cret);
103 RegCloseKey(hkey);
104 (*env)->ReleaseStringUTFChars(env, name, cname);
105
106 return ret;
107 }
108
109 JNIEXPORT void JNICALL Java_Win32RegKey_setValue(JNIEnv* env, jobject this_obj,
110 jstring name, jobject value)
111 {
112 const char* cname;
113 jstring path;
114 const char* cpath;
115 HKEY hkey;
116 DWORD type;
117 DWORD size;
118 jclass this_class;
119 jclass class_value;
120 jclass class_Integer;
121 jfieldID id_root;
122 jfieldID id_path;
123 HKEY root;
124 const char* cvalue;
125 int ivalue;
126
127 /* get the class */
128 this_class = (*env)->GetObjectClass(env, this_obj);
129
130 /* get the field IDs */
131 id_root = (*env)->GetFieldID(env, this_class, "root", "I");
132 id_path = (*env)->GetFieldID(env, this_class, "path", "Ljava/lang/String;");
133
134 /* get the fields */
135 root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
136 path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
137 cpath = (*env)->GetStringUTFChars(env, path, NULL);
138
139 /* open the registry key */
140 if (RegOpenKeyEx(root, cpath, 0, KEY_WRITE, &hkey) != ERROR_SUCCESS)
141 {
```

(Continues)

**Listing 12.22 (Continued)**

```
142 (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
143 "Open key failed");
144 (*env)->ReleaseStringUTFChars(env, path, cpath);
145 return;
146 }
147
148 (*env)->ReleaseStringUTFChars(env, path, cpath);
149 cname = (*env)->GetStringUTFChars(env, name, NULL);
150
151 class_value = (*env)->GetObjectClass(env, value);
152 class_Integer = (*env)->FindClass(env, "java/lang/Integer");
153 /* determine the type of the value object */
154 if ((*env)->IsAssignableFrom(env, class_value, (*env)->FindClass(env, "java/lang/String")))
155 {
156 /* it is a string--get a pointer to the characters */
157 cvalue = (*env)->GetStringUTFChars(env, (jstring) value, NULL);
158 type = REG_SZ;
159 size = (*env)->GetStringLength(env, (jstring) value) + 1;
160 }
161 else if ((*env)->IsAssignableFrom(env, class_value, class_Integer))
162 {
163 /* it is an integer--call intValue to get the value */
164 jmethodID id_intValue = (*env)->GetMethodID(env, class_Integer, "intValue", "()I");
165 ivalue = (*env)->CallIntMethod(env, value, id_intValue);
166 type = REG_DWORD;
167 cvalue = (char*)&ivalue;
168 size = 4;
169 }
170 else if ((*env)->IsAssignableFrom(env, class_value, (*env)->FindClass(env, "[B")))
171 {
172 /* it is a byte array--get a pointer to the bytes */
173 type = REG_BINARY;
174 cvalue = (char*)(*env)->GetByteArrayElements(env, (jarray) value, NULL);
175 size = (*env)->GetArrayLength(env, (jarray) value);
176 }
177 else
178 {
179 /* we don't know how to handle this type */
180 (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
181 "Unsupported value type");
182 RegCloseKey(hkey);
183 (*env)->ReleaseStringUTFChars(env, name, cname);
184 return;
185 }
186 /* set the value */
```

```
188 if (RegSetValueEx(hkey, cname, 0, type, cvalue, size) != ERROR_SUCCESS)
189 {
190 (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
191 "Set value failed");
192 }
193
194 RegCloseKey(hkey);
195 (*env)->ReleaseStringUTFChars(env, name, cname);
196
197 /* if the value was a string or byte array, release the pointer */
198 if (type == REG_SZ)
199 {
200 (*env)->ReleaseStringUTFChars(env, (jstring) value, cvalue);
201 }
202 else if (type == REG_BINARY)
203 {
204 (*env)->ReleaseByteArrayElements(env, (jarray) value, (jbyte*) cvalue, 0);
205 }
206 }
207
208 /* helper function to start enumeration of names */
209 static int startNameEnumeration(JNIEnv* env, jobject this_obj, jclass this_class)
210 {
211 jfieldID id_index;
212 jfieldID id_count;
213 jfieldID id_root;
214 jfieldID id_path;
215 jfieldID id_hkey;
216 jfieldID id_maxsize;
217
218 HKEY root;
219 jstring path;
220 const char* cpath;
221 HKEY hkey;
222 DWORD maxsize = 0;
223 DWORD count = 0;
224
225 /* get the field IDs */
226 id_root = (*env)->GetFieldID(env, this_class, "root", "I");
227 id_path = (*env)->GetFieldID(env, this_class, "path", "Ljava/lang/String;");
228 id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
229 id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize", "I");
230 id_index = (*env)->GetFieldID(env, this_class, "index", "I");
231 id_count = (*env)->GetFieldID(env, this_class, "count", "I");
232
233 /* get the field values */
234 root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
235 path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
```

(Continues)

**Listing 12.22 (Continued)**

```
236 cpath = (*env)->GetStringUTFChars(env, path, NULL);
237
238 /* open the registry key */
239 if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) != ERROR_SUCCESS)
240 {
241 (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
242 "Open key failed");
243 (*env)->ReleaseStringUTFChars(env, path, cpath);
244 return -1;
245 }
246 (*env)->ReleaseStringUTFChars(env, path, cpath);
247
248 /* query count and max length of names */
249 if (RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL, NULL, NULL, &count, &maxsize,
250 NULL, NULL, NULL) != ERROR_SUCCESS)
251 {
252 (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
253 "Query info key failed");
254 RegCloseKey(hkey);
255 return -1;
256 }
257
258 /* set the field values */
259 (*env)->SetIntField(env, this_obj, id_hkey, (DWORD) hkey);
260 (*env)->SetIntField(env, this_obj, id_maxsize, maxsize + 1);
261 (*env)->SetIntField(env, this_obj, id_index, 0);
262 (*env)->SetIntField(env, this_obj, id_count, count);
263 return count;
264 }
265
266 JNIEXPORT jboolean JNICALL Java_Win32RegKeyNameEnumeration_hasMoreElements(JNIEnv* env,
267 jobject this_obj)
268 { jclass this_class;
269 jfieldID id_index;
270 jfieldID id_count;
271 int index;
272 int count;
273 /* get the class */
274 this_class = (*env)->GetObjectClass(env, this_obj);
275
276 /* get the field IDs */
277 id_index = (*env)->GetFieldID(env, this_class, "index", "I");
278 id_count = (*env)->GetFieldID(env, this_class, "count", "I");
279
280 index = (*env)->GetIntField(env, this_obj, id_index);
281 if (index == -1) /* first time */
282 {
```

```
283 count = startNameEnumeration(env, this_obj, this_class);
284 index = 0;
285 }
286 else
287 count = (*env)->GetIntField(env, this_obj, id_count);
288 return index < count;
289 }
290
291 JNIEXPORT jobject JNICALL Java_Win32RegKeyNameEnumeration.nextElement(JNIEnv* env,
292 jobject this_obj)
293 {
294 jclass this_class;
295 jfieldID id_index;
296 jfieldID id_hkey;
297 jfieldID id_count;
298 jfieldID id_maxsize;
299
300 HKEY hkey;
301 int index;
302 int count;
303 DWORD maxsize;
304
305 char* cret;
306 jstring ret;
307
308 /* get the class */
309 this_class = (*env)->GetObjectClass(env, this_obj);
310
311 /* get the field IDs */
312 id_index = (*env)->GetFieldID(env, this_class, "index", "I");
313 id_count = (*env)->GetFieldID(env, this_class, "count", "I");
314 id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
315 id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize", "I");
316
317 index = (*env)->GetIntField(env, this_obj, id_index);
318 if (index == -1) /* first time */
319 {
320 count = startNameEnumeration(env, this_obj, this_class);
321 index = 0;
322 }
323 else
324 count = (*env)->GetIntField(env, this_obj, id_count);
325
326 if (index >= count) /* already at end */
327 {
328 (*env)->ThrowNew(env, (*env)->FindClass(env, "java/util/NoSuchElementException"),
329 "past end of enumeration");
330 return NULL;
331 }
```

(Continues)

**Listing 12.22 (Continued)**

```
332 maxlen = (*env)->GetIntField(env, this_obj, id_maxlen);
333 hkey = (HKEY)(*env)->GetIntField(env, this_obj, id_hkey);
334 cret = (char*)malloc(maxlen);
335
336 /* find the next name */
337 if (RegEnumValue(hkey, index, cret, &maxlen, NULL, NULL, NULL, NULL) != ERROR_SUCCESS)
338 {
339 (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
340 "Enum value failed");
341 free(cret);
342 RegCloseKey(hkey);
343 (*env)->SetIntField(env, this_obj, id_index, count);
344 return NULL;
345 }
346
347 ret = (*env)->NewStringUTF(env, cret);
348 free(cret);
349
350 /* increment index */
351 index++;
352 (*env)->SetIntField(env, this_obj, id_index, index);
353
354 if (index == count) /* at end */
355 {
356 RegCloseKey(hkey);
357 }
358
359 return ret;
360 }
```

---

**Listing 12.23 win32reg/Win32RegKeyTest.java**

```
1 import java.util.*;
2
3 /**
4 * @version 1.02 2007-10-26
5 * @author Cay Horstmann
6 */
7 public class Win32RegKeyTest
8 {
9 public static void main(String[] args)
10 {
11 Win32RegKey key = new Win32RegKey(
12 Win32RegKey.HKEY_CURRENT_USER, "Software\\JavaSoft\\Java Runtime Environment");
13
14 key.setValue("Default user", "Harry Hacker");
```

```
15 key.setValue("Lucky number", new Integer(13));
16 key.setValue("Small primes", new byte[] { 2, 3, 5, 7, 11 });
17
18 Enumeration<String> e = key.names();
19
20 while (e.hasMoreElements())
21 {
22 String name = e.nextElement();
23 System.out.print(name + "=");
24
25 Object value = key.getValue(name);
26
27 if (value instanceof byte[])
28 for (byte b : (byte[]) value) System.out.print((b & 0xFF) + " ");
29 else
30 System.out.print(value);
31
32 System.out.println();
33 }
34 }
35 }
```

#### Type Inquiry Functions

- `jboolean IsAssignableFrom(JNIEnv *env, jclass c1, jclass c2)`  
returns `JNI_TRUE` if objects of the first class can be assigned to objects of the second class, and `JNI_FALSE` otherwise. This tests if the classes are the same, or `c1` is a subclass of `c2`, or `c2` represents an interface implemented by `c1` or one of its superclasses.
- `jclass GetSuperclass(JNIEnv *env, jclass c1)`  
returns the superclass of a class. If `c1` represents the class `Object` or an interface, returns `NULL`.

You have now reached the end of the second volume of *Core Java*, completing a long journey in which you encountered many advanced APIs. We started out with topics that every Java programmer needs to know: streams, XML, networking, databases, and internationalization. Three long chapters covered graphics and GUI programming. We concluded with very technical chapters on security, remote methods, annotation processing, and native methods. We hope that you enjoyed your tour through the vast breadth of the Java APIs, and that you will be able to apply your newly gained knowledge in your projects.

# Index

## Numbers

- (minus sign)
  - in permissions, 521
  - in policy files, 520
  - in regular expressions, 129
  - in URLs, 270
- \_ (underscore)
  - in native method names, 941
  - in SQL, 289, 319
  - in URLs, 270
- , (comma)
  - decimal, 372, 378, 686
  - in DTDs, 165
- ; (semicolon)
  - in classpath, 292
  - in method signatures, 962
  - in SQL, 293
  - not needed, in annotations, 455
- : (colon)
  - as delimiter in text files, 63
  - in classpath, 292
  - in permissions, 521
  - in URLs, 258
- != operator (SQL), 289
- ? (question mark)
  - in DTDs, 165
  - in e-mail URIs, 927
  - in glob patterns, 112–129
  - in masks, 692
  - in prepared queries, 309
  - in URLs, 269
- / (slash)
  - ending, in codebase URLs, 516
  - in method signatures, 962
  - in paths, 101
  - in URLs, 258
- . (period)
  - decimal, 372, 378, 686
  - in method signatures, 962
  - in regular expressions, 129–130
  - in URLs, 270
  - leading, in file names, 516
- ..., in paths, 102
- ^ (caret), in regular expressions, 129, 132
- ~ (tilde), in URLs, 270
- ' (single quote), in masks, 692
- '. . .', in SQL, 289
- ". . .", in XML, 146
- (. . .)
  - in method signatures, 962
  - in regular expressions, 131, 133
- [ (array), type code, 87, 961
- {. . .}
  - in annotations, 466
  - in glob patterns, 112
  - in message formatting, 400–404
  - in regular expressions, 132
- [. . .]
  - in DOCTYPE declaration, 164
  - in glob patterns, 112
  - in regular expressions, 129–130
  - in XPath, 190
- @ (at)
  - in annotations, 455
  - in URIs, 258
  - in XPath, 190
- \$ (dollar sign)
  - in native method names, 941
  - in regular expressions, 129–130, 132, 141
- \${. . .}, in policy files, 521
- \* (asterisk)
  - in DTDs, 165
  - in glob patterns, 112
  - in masks, 692
  - in permissions, 521
  - in policy files, 520
  - in regular expressions, 129, 132
- \ (backslash)
  - in glob patterns, 112
  - in paths, 56, 101
  - in permissions (Windows), 520
  - in regular expressions, 129–130, 141
- \\|, in regular expressions, 64

- & (ampersand)
  - in CDATA sections, 149
  - in e-mail URIs, 927
  - in entity references, 148
  - parsing, 165
- &&, in regular expressions, 130
- &#, &#x, in character references, 148
- # (number sign)
  - in masks, 692
  - in message formatting, 403–404
  - in URLs, 258
- % (percent sign)
  - in fortune program, 933
  - in locales, 378
  - in SQL, 289, 319
  - in URLs, 270
- %20, in URIs, 927
- + (plus sign)
  - in DTDs, 165
  - in regular expressions, 129, 132
  - in URLs, 270
- < (left angle bracket)
  - in CDATA sections, 149
  - in message formatting, 404
  - parsing, 165
- <!--. . .-->, <?. . ?.>, <![CDATA[. . .]]>, in XML, 149
- <. . .>, in regular expressions, 131
- ≤ operator, 404
- =, <> operators (SQL), 289
- == operator, in enumerations, 94
- > (right angle bracket), in XML, 149
- | (vertical bar)
  - as delimiter in text files, 63
  - in DTDs, 165, 167
  - in message formatting, 403
  - in regular expressions, 129, 131
- \0, in regular expressions, 130
- 1931 CIE XYZ color specification, 836
- 2D graphics, 765–937
  - printing, 852–862
- Xxx2D, Xxx2D.Float, Xxx2D.Double classes, 770–771
- A**
  - A in masks, 692
- \a, \A, in regular expressions, 130, 132
- abort method (*LoginModule*), 546
- absolute method (*ResultSet*), 323, 327
- AbstractCellEditor class, 629–631
  - isCellEditable method, 630
- AbstractFormatter class
  - getDocumentFilter method, 689, 701
  - stringToValue, valueToString methods, 693, 701
- AbstractListModel class, 589
- AbstractProcessor class, 476
- AbstractSpinnerModel class
  - fireStateChanged method, 706
  - get/setValue methods, 705, 712
  - getNextValue, getPreviousValue methods, 705–706, 712
- AbstractTableModel class, 605
  - isCellEditable method, 628
- accept method (*ServerSocket*), 242, 245
- acceptChanges method (*CachedRowSet*), 331–332
- AccessController class, getContext method, 536
- Action listeners
  - annotating, 457–462
  - for text fields, 683
  - installing, 457
- action.properties file, 449
- ActionListener interface, 457
  - actionPerformed method, 458
- @ActionListenerFor annotation, 457, 472
- ActionListenerInstaller class, 457
- add method
  - of Area, 786–788
  - of AttributeSet, 881
  - of DefaultMutableTreeNode, 642, 650
  - of SystemTray, 932, 936
- addActionListener method
  - injecting into frame constructor, 458
  - of TrayIcon, 937
  - using annotations instead of, 457
- addAttribute method (*AttributesImpl*), 232
- addBatch method (*Statement*), 345–347
- addCellEditorListener method (*CellEditor*), 638
- addChangeListener method (*JTabbedPane*), 737, 741
- addColumn method (*JTable*), 617, 622
- addDocumentListener method (*Document*), 682, 685
- addElement method (*DefaultListModel*), 595
- addFlavorListener method (*Clipboard*), 894
- addHyperlinkListener method (*JEditorPane*), 718
- addListSelectionListener method (*JFrame*), 587
- addRecipient method (*MimeMessage*), 278
- addTab method (*JTabbedPane*), 736, 740
- addTreeModelListener method (*TreeModel*), 672, 680
- addTreeSelectionListener method (*JTree*), 664
- addVetoableChangeListener method (*JComponent*), 759
- AES (Advanced Encryption Standard), 568
  - generating keys in, 569–574

- aes/AESTest.java, 571
- aes/Util.java, 572
- Affine transformations, 802
  - on pixels, 843
- AffineTransform class, 802–804
  - constructor, 803
  - getXXXInstance methods, 802–804
  - setToXXX methods, 802, 804
- AffineTransformOp class, 843
  - constructor, 850
  - TYPE\_XXX fields, 843, 850
- afterLast method (*ResultSet*), 323, 327
- Agent code, 487–488
- Aliases, for namespaces, 172, 197
- allMatch method (*Stream*), 13
- allocate method (*ByteBuffer*), 123, 125
- AllPermission class, 513, 519, 523
- Alpha channel, 807–811
- Alpha composites, 817–818
- AlphaComposite class, 809
  - getInstance method, 809, 817
- & entity reference, 148
- Anchor rectangle, 798
- andFilter method (*RowFilter*), 615, 625
- AnnotatedConstruct interface, 477
- AnnotatedElement interface
  - getAnnotation method, 457, 462, 475, 477
  - getAnnotations, getDeclaredAnnotations, isAnnotationPresent methods, 462
  - getAnnotationsByType methods, 462, 475, 477
- Annotation interface
  - annotationType method, 463
  - equals, hashCode, toString methods, 464
  - extending, 463
- Annotation interfaces, 456, 462–463
  - methods of, 463
  - predefined, 470–475
- Annotation processors, 476
  - at bytecode level, 458, 481
  - at runtime, 458
- Annotations, 455–462
  - applicability of, 472–473
  - documented, 472, 474
  - elements of, 456, 463, 466
  - for compilation, 471
  - for local variables, 467
  - for managing resources, 472
  - for packages, 467
  - generating source code with, 477–480
  - inherited, 474
- marker, 465
- meta, 456, 470–475
- no annotations for, 468
- no circular dependencies in, 466
- processing tools for, 455
- repeatable, 475
- retaining, 472
- single value, 465
- source-level, 475–480
- standard, 470–475
- syntax of, 462–469
- transient, 473
  - vs. Javadoc comments, 455
- Antialiasing, 817–819
- ANY element content (DTD), 165
- anyMatch method (*Stream*), 13
- Apache, 143, 150
- Apache Derby database, 291
  - connecting to, 295
  - default user name/password for, 295
  - driver for, 292
  - populating, 305–309
  - registering driver class for, 294
  - starting, 293
- Apache HttpClient, 274
- Apache Tomcat, 349
- Apollo 11, launch of, 355, 361
- &apos;, entity reference, 148
- append method
  - of *Appendable*, 54–55
  - of *Book*, 873
  - of shape classes, 776, 786
- Appendable interface, 52–55
- appendChild method (*Node*), 209, 212
- Applets
  - accessing browser's cache, 261
  - class loaders for, 496
  - code base of, 510
  - executing from secure intranet, 547
  - not exiting JVM, 509
  - popping up dialog boxes, 261
  - running in browser, 564
  - security mechanisms in, 491–579
  - warning windows in, 526
- appletviewer program, 515, 564
- Application servers, 349, 472
- Applications
  - communicating with native code, 903–921
  - configuring, 144–145
  - data transfers between, 888

- desktop, 927–932
- enterprise, 349–350
- loading, 921
- localizing, 408
- monitoring, 487–488
- paid, 499
- server-side, 267–277
- signing, 561–567
- splash screen for, 921–926
- startup plugins for, 922
- updates of, 932
- with internal frames, 741–744
- `applyPattern` method (`MessageFormat`), 401
- `apply-templates` element (XSLT), 224
- `Arc2D` class, 769, 772–773
- `Arc2D.Double` class, 785
- `ArcMaker` class, 777
- Arcs**
  - bounding rectangle of, 770, 772
  - closure types of, 772–773
  - computing angles of, 773
- Area** class
  - `add` method, 786–788
  - `subtract`, `intersect`, `exclusiveOr` methods, 787–788
- Areas**, 786–788
- ARGB** (Alpha, Red, Green, Blue), 811, 837
- ARRAY** data type (SQL), 347–348
- ArrayIndexOutOfBoundsException**, 975
- Arrays**
  - from stream elements, 19
  - getting from a database, 347
  - in annotation elements, 466
  - in native code, 970–974
  - of primitive types, 973
  - type code for, 87, 961
  - type use annotations in, 467
  - vs. C types, 970–974
- Arrays** class, `stream` method, 8, 36
- ArrayStoreException**, 975
- `asCharBuffer` method (`ByteBuffer`), 123
- ASCII** standard, 67
  - and native code, 950
  - converting native encodings to, 408
  - in property files, 410
  - in regular expressions, 133
- ASM** library, 481–489
- ASP** (Active Server Pages), 268
- ATTLIST declaration** (DTD), 167
- attribute element** (XML Schema), 174
- Attribute** interface, 879
  - `getCategory` method, 881, 886
  - `getName` method, 886
  - implementing, 881
- Attribute sets**, 881
- Attributes** (XML)
  - enumerating, 154
  - for enumerated types, 168
  - in XML Schema, 174
  - legal, 167
  - names of, 146
  - namespace of, 198
  - values of, 146
  - accessing in XPath, 190
  - copying with XSLT, 225
  - default (DTDs), 167
  - normalizing, 168
  - vs. elements, 147–148, 168, 212
- Attributes** interface, `getXxx` methods, 204–205
- AttributeSet** interface, 880
  - `add`, `get` methods, 881, 886
  - `remove`, `toArray` methods, 886
- AttributesImpl** class, `addAttribute`, `clear` methods, 232
- atZone** method (`LocalDateTime`), 361
- AudioPermission** class, 519
- `auth/AuthTest.java`, 534
- `auth/AuthTest.policy`, 535
- `auth/jaas.config`, 535
- `auth/SysPropAction.java`, 535
- Authentication**, 530–546
  - problems of, 556–557
  - role-based, 537
  - separating from business logic, 532
  - through a trusted intermediary, 557–558
- AuthPermission** class, 520
- Autoboxing**, 601
- AutoCloseable** interface, `close` method, 53
- Autocommit mode** (databases), 344
  - turning off, 346
- Autoflushing**, 61
- Autogenerated keys**, 320–321
- Automatic registration**, of JAR files, 294
- Auxiliary files**, generated, 455
- available** method (`InputStream`), 49–50, 727
- average** method (`XxxStream`), 37, 39–40
- AWT** (Abstract Window Toolkit), 765–937
- AWTPermission** class, 518, 526

**B**

- `B` (byte), type code, 87, 961
- `\b, \B`, in regular expressions, 132
- Banding, 854
- Banner class, `getPageCount`, `layoutPages` methods, 863
- Banners, printing, 862–864
- `BaseStream` interface
  - iterator methods, 19–20
  - parallel method, 41–45
  - unordered method, 43, 45
- `BasicFileAttributes` interface, 108
  - methods of, 109
- BasicPermission class, 520
- BasicStroke class, 788–796
- Batch updates (databases), 345–347
- Bean info classes, generated, 455
- BeanInfo class, 178
- `beforeFirst` method (*ResultSet*), 323, 327
- `between` method (*Duration*), 352
- Bevel join, 788–789
- Bézier curves, 775
- `BIG_ENDIAN` constant (*ByteOrder*), 123
- Big-endian order, 70, 407
- Bilinear interpolation, 843
- Binary data
  - converting to Unicode code units, 60
  - reading, 70
  - vs. text, 60
  - writing, 69
- `Bindings` interface, 433
  - get, put methods, 434
- BLOB data type (SQL), 291, 348
- `Blob` interface, 316
  - `get/setBinaryStream`, `getBytes` methods, 316–317
  - `length` method, 317
- BLOBs (binary large objects), 316
  - creating empty, 318
  - placing in database, 316
- Blocking
  - by I/O methods, 49
  - by network connections, 234, 238, 250–257
- Blur filter, 844
- BMP format, 823
- Book class, 862
  - `append`, `getPrintable` methods, 873
- `book/Banner.java`, 867
- `book/BookTestFrame.java`, 865
- `book/PrintPreviewCanvas.java`, 871
- `book/PrintPreviewDialog.java`, 870
- Books, counting words in, 2
- BOOLEAN data type (SQL), 291, 348
- boolean type
  - printing, 61
  - type code for, 87, 961
  - vs. C types, 947
  - writing in binary format, 69
- Bootstrap class loader, 493
- Bounding rectangle, 770
- boxed method (of streams), 36, 39–40
- Bray, Tim, 146
- Breadth-first enumerations, 659
- `breadthFirstEnumeration` method
  - (*DefaultMutableTreeNode*), 659, 663
- `browse` method (*Desktop*), 932
- Browsers
  - bytecode verification in, 506
  - cache of, 261
  - custom, 713
  - forms in, 267–277
  - launching from desktop applications, 927
  - password-protected pages in, 261
  - response page in, 268
  - running applets in, 564
- Buffer class, 124–126
  - capacity method, 126
  - clear method, 125–126
  - flip method, 125–126
  - hasRemaining method, 122
  - limit method, 122
  - mark method, 125–126
  - position method, 126
  - remaining method, 125–126
  - reset method, 125–126
  - rewind method, 125–126
- `BufferedImage` class, 798, 834
  - constructor, 835, 840
  - `getColorModel` method, 837, 840
  - `getRaster` method, 835, 840
  - `TYPE_BYTE_GRAY` field, 838, 840
  - `TYPE_INT_ARGB` field, 835–836, 840
- `BufferedImageOp` interface, 834
  - `filter` method, 842, 850
  - implementing, 842
- BufferedInputStream, BufferedOutputStream classes, 59
- BufferedReader class, `readLine` method, 63
- Buffers, 124–126
  - capacity of, 124
  - flushing, 49, 61
  - in-memory, 53

- limits of, 124  
marks in, 124  
positions in, 117, 124  
traversing all bytes in, 117  
vs. random access, 116
- BufferUnderflowException, 123
- @BugReport annotation, 473
- Business logic, 284
- Butt cap, 788–789
- Button listeners, for a dialog box, 688
- ButtonFrame class, 437, 457
- buttons1/ButtonFrame.java, 442
- buttons2/action.properties, 452
- buttons2/ButtonFrame.java, 451
- buttons3/ButtonFrame.java, 460
- BYTE\_ARRAY class (DocFlavor), 875
- Byte order mark, 407
- byte type  
negative values and unsigned integers, 693  
type code for, 87, 961  
vs. C types, 947
- ByteArrayOutputStream class, 98, 445
- ByteBuffer class, 117, 124–126  
allocate method, 123, 125  
asCharBuffer method, 123  
get method, 117, 122  
getXxx methods, 117, 123  
order method, 118, 123  
put method, 122  
putXxx methods, 118, 123  
wrap method, 123, 125
- bytecodeAnnotations/EntryLogger.java, 483
- bytecodeAnnotations/EntryLoggingAgent.java, 488
- Bytecodes  
engineering, 481–489  
at load time, 486–488  
with hex editor, 507  
verifying, 504–508
- ByteLookupTable class, 844  
constructor, 850
- ByteOrder class, BIG\_ENDIAN, LITTLE\_ENDIAN constants, 123
- Byte-oriented input/output streams, 48
- Bytes, reading/writing, 48–51
- C**
- C (char), type code, 87, 961
- C programming language  
array types in, 970–974  
bootstrap class loader in, 493
- calling Java methods from, 963–970  
database access in, 282  
embedding JVM into, 980–985  
FILE\* type in, 51  
header file generating, 942  
invalid pointers in, 940  
strings in, 949  
types, vs. Java types, 947  
\c, in regular expressions, 130
- C++ programming language  
accessing JNI functions in, 950  
array types in, 971  
embedding JVM into, 980–985  
exceptions in, 975  
for native methods, 940, 943  
invalid pointers in, 940  
pointers to dummy classes in, 965
- Cache, 261
- Cached row sets, 329–334
- CachedRowSet interface, 329–333  
acceptChanges method, 331–332  
execute method, 330, 332  
get/setPageSize method, 330, 332  
get/setTableName method, 331–332  
nextPage method, 330, 332  
populate method, 330, 332  
previousPage method, 332
- CachedRowSetImpl class, 329
- Caesar cipher, 498–499
- Calendar class, 351  
formatting objects of, 386
- call escape (SQL), 319
- call method (*CompilationTask*), 445, 448
- Call stack, during permission checking, 513
- Callable interface, 445
- Callback interface, 538
- CallbackHandler interface, handle method, 545
- CallNonvirtualXxxMethod functions (C), 966, 969
- CallStaticXxxMethod functions (C), 964–965, 970
- CallXxxMethod functions (C), 963, 969
- cancelCellEditing method (*CellEditor*), 631–632, 638
- cancelRowUpdates method (*ResultSet*), 325, 328
- canImport method (*TransferHandler*), 912, 918
- canInsertImage method (*ImageWriter*), 827, 834
- capacity method (*Buffer*), 126
- Carriage return character, displaying, 154
- Cascading windows, 744–745
- Casts, type use annotations in, 468
- Catalogs, 342

- CDATA declaration (DTD), 167–168
- CDATA sections (XML), 149
- Cell editors (Swing), 628–629
  - custom, 629–638
- Cell renderers (Swing)
  - for lists, 595–599
  - for tables, 609, 626
  - for trees, 661–664
- CellEditor* interface
  - `add/removeCellEditorListener` methods, 638
  - `cancelCellEditing` method, 631–632, 638
  - `getCellEditorValue` method, 629, 631–632, 638
  - `isCellEditable` method, 638
  - `shouldSelectCell` method, 631, 638
  - `stopCellEditing` method, 631–632, 638
- Cells (Swing)
  - editing, 628–629
  - rendering, 626–638
  - selecting, 612–614, 626
- Certificates, 531, 553–556
  - and Java Plug-in, 563
  - managing, 560–561
  - publishing fingerprints of, 555
  - root, 563
  - set of, 510
  - signing, 558–561
- CertificateSigner* class, 559
- CGI (Common Gateway Interface), 268
- Chain of trust, 557
- ChangeListener* interface, 737
- `changeUpdate` method (*DocumentListener*), 682, 685
- Channels, 250
  - for files, 117
- Channels class
  - `newInputStream` method, 256
  - `newOutputStream` method, 250, 257
- char type
  - type code for, 87, 961
  - vs. C types, 947
- CHAR\_ARRAY* class (*DocFlavor*), 875
- Character classes, 129
- CHARACTER data type (SQL), 291, 348
- Character encodings, 60, 67–69
  - converting to ASCII, 408
  - platform, 68–69, 404
- Character references (XML), 148
- CharacterData* interface, `getData` method, 153, 162
- Characters
  - antialiasing, 817
  - differences between, 394
- escaping, 64, 129
- in regular expressions, 133
- normalizing, 394
- outlines of, 805
- printing, 61
- turning to upper/lower case, 690
- writing in binary format, 69
- characters method (*ContentHandler*), 200, 204
- charAt* method (*String*), 9
- CharBuffer* class, 53–54, 124
  - `get`, `put` methods, 124
- CharSequence* interface, 54, 134
  - `charAt`, `length` methods, 55
  - `chars` method, 36
  - `codePoints` method, 36, 40
  - splitting by regular expressions, 6
  - `subSequence`, `toString` methods, 55
- Charset class
  - `defaultCharset` method, 68, 406
  - `forName` method, 69
- Checkboxes (Swing), 626
- checked attribute (HTML, XML), 146
- Checked exceptions, 533
- `checkError` method (*PrintWriter*), 61–62
- `checkExit` method (*SecurityManager*), 509, 512
- `checkLogin` method (*SimpleLoginModule*), 538
- `checkPermission` method (*SecurityManager*), 512–513, 523–524
- `checkRead` method (*SecurityManager*), 513
- Child elements (XML), 147
  - getting node list of, 154
  - namespace of, 197
- Child nodes (Swing), 639
  - adding, 642
  - connecting lines for, 645–646
- children* method (*TreeNode*), 658
- choice element (XML Schema), 173
- choice keyword (message formatting), 403
- Church, Alonzo, 355
- Cipher* class, 567–569
  - `doFinal` method, 568, 571, 573, 575
  - `getInstance` method, 567, 573
  - `getXXXSize` methods, 573
  - `init` method, 573
  - `update` method, 568, 571, 573, 575
  - `XXX_MODE` modes, 568
- CipherInputStream* class, `read` method, 575
- CipherOutputStream* class, 574
  - constructor, 575
  - `flush`, `write` methods, 575

Ciphers  
generating keys, 569–574  
public keys in, 575–579  
streams for, 574–575  
symmetric, 567–569

Circular dependencies, in annotations, 466

Class class  
  `getClassLoader` method, 493, 503  
  `getFields` method, 674  
  `getProtectionDomain` method, 514  
  implementing *AnnotatedElement*, 457

Class files, 492  
  corrupted, 504–508  
  encrypted, 498–499  
  format of, 481  
  loading, 492  
  locating, 444  
  modifying, 481–486  
  portability of, 407  
  producing on disk, 445  
  storing in databases, 444  
  transformers for, 487  
  verifying, 504–508

Class loaders, 450, 492–508  
  as namespaces, 496  
  bootstrap, 493  
  context, 495–496  
  creating, 509  
  extension, 493  
  hierarchy of, 494–496  
  separate for each web page, 496  
  specifying, 494  
  system, 493  
  writing, 497–503

class parameter (MIME types), 892

Class path hell, 493

Class references, in native code, 958

.class file extension, 492

Classes  
  adding validation to, 92  
  annotating, 456, 466, 470  
  descriptions of, 86  
  externalizable, 87  
  inheritance trees of, 660  
  loading, 492–494  
  locales supported by, 376  
  nonserializable, 92  
  protection domains of, 512  
  resolving, 492  
  separate for each web page, 496

serializable, 80–81  
versioning, 95–98

Classifier functions, 28

ClassLoader class, 493  
  `defineClass`, `findClass` methods, 498, 503  
  extending, 497  
  `getParent`, `getSystemClassLoader` methods, 503  
  `loadClass` method, 496, 498

ClassLoader inversion, 494

classLoader/Caesar.java, 502

classLoader/ClassLoaderTest.java, 499

CLASSPATH environment variable, 493

CLEAR composition rule, 810

clear method  
  of *AttributesImpl*, 232  
  of *Buffer*, 125–126

clearParameters method (*PreparedStatement*), 316

Client/server applications, 285

Clients  
  configuring securely, 566  
  connecting to servers, 236–238  
  multiple, serving, 245–248

clip method (*Graphics2D*), 767, 805–807, 854

Clipboard, 887–903  
  accessing, 509  
  capabilities of, 888  
  transferring:  
    images, 894  
    object references, 902  
    objects, 898–902  
    text, 888–892

Clipboard class, 888  
  `addFlavorListener` method, 894  
  constructor, 903  
  `get/setContents` methods, 889, 891  
  `getAvailableDataFlavors` method, 894  
  `getData` method, 892  
  `isDataFlavorAvailable` method, 891

ClipboardOwner interface  
  implementing, 888  
  `lostOwnership` method, 888, 892

Clipping, 766, 805–807

Clipping region  
  printing, 854  
  setting, 767

CLOB data type (SQL), 291, 348

Clob interface, 316  
  `get/setCharacterStream` methods, 316, 318  
  `getSubString` method, 316–317  
  `length` method, 317

- CLOBs (character large objects), 316  
  creating empty, 318  
  placing in database, 316  
clone method, 81, 98  
Cloning, 98–100  
close method  
  of *AutoCloseable*, 53  
  of *Connection*, 299, 301, 350  
  of *FileLock*, 128  
  of *Flushable*, 52  
  of *InputStream*, 49–50  
  of *OutputStream*, 51  
  of *ProgressMonitor*, 723, 731  
  of *ResultSet*, 301  
  of *ServerSocket*, 245  
  of *SplashScreen*, 926  
  of *Statement*, 300–301  
  of *XMLStreamWriter*, 222  
*Closeable* interface, 52  
  close method, 52, 54  
  flush method, 52  
Closed nonleaf icons, 648, 661  
closeEntry methods (*ZipXxxStream*), 77–78  
closeOnCompletion method (*Statement*), 300  
closePath method (*Path2D*), 776, 786  
Closure types, 772  
cmd shell, 406  
Code base, 510, 516  
Code generation, annotations for, 455–462, 471  
Code source, 510  
Code units (UTF-16), in regular expressions, 130  
*Codebreakers, The* (Kahn), 498  
codePoints method (*CharSequence*), 36, 40  
CodeSource class, *getXxx* methods, 514  
Collation, 393–400  
collation/CollationTest.java, 396  
CollationKey class, compareTo method, 400  
Collator class, 393  
  compare, equals methods, 399  
  get/setDecomposition methods, 399  
  get/setStrength methods, 399  
  getAvailableLocales method, 399  
  getCollationKey method, 395, 399  
  getInstance method, 399  
collect method (*Stream*), 19, 22, 35  
collecting/CollectingIntoMaps.java, 25  
collecting/CollectingResults.java, 20  
collecting/DownstreamCollectors.java, 31
- Collection* interface  
  parallelStream method, 2–5, 41–46  
  stream method, 2–5  
Collections  
  processing, 2–5  
  vs. streams, 3  
Collections class, sort method, 394  
Collector interface, 19  
Collectors class  
  counting method, 29  
  groupingBy method, 28–31  
  groupingByConcurrent method, 43  
  joining method, 20, 23  
  mapping method, 30, 33  
  maxBy, minBy methods, 30, 33  
  partitioningBy method, 28, 31  
  summarizingXxx methods, 20, 23  
  summingXxx methods, 29, 33  
  toCollection method, 20, 23  
  toConcurrentMap method, 25, 27  
  toList method, 23  
  toMap method, 24–27  
  toSet method, 20, 23, 29  
Collectors, downstream, 29–33  
Color chooser, 629, 898–902  
Color class, 797  
  constructor, 842  
  getRGB method, 842  
  translating values into pixel data, 838  
Color space conversions, 844  
ColorConvertOp class, 843–844  
ColorModel class, 838  
  getDataElements method, 842  
  getRGB method, 837, 842  
Colors  
  components of, 807  
  composing, 808–811  
  dithering, 818  
  interpolating, 797–798  
  negating, 844  
  solid, 766  
Columns (databases)  
  accessing by number, in result set, 299  
  names of, 285  
  number of, 334  
Columns (Swing)  
  accessing, 610  
  adding, 617  
  detached, 601  
  hiding, 617–625

names of, 606  
rendering, 609–610  
resizing, 602, 611–612  
selecting, 612  
`com.sun.rowset` package, 329  
`com.sun.security.auth.module` package, 532  
Combo boxes  
  editing, 629  
  traversal order in, 704  
Comments (XML), 149  
`commit` method  
  of *Connection*, 344–346  
  of *LoginModule*, 546  
Commit or revert behavior, 687–688  
`commitEdit` method (`JFormattedTextField`), 688, 701  
`Comparable` interface, 615  
`Comparator` interface, 394  
Comparators, 614  
`compare` method (`Collator`), 399  
`compareTo` method  
  of `CollationKey`, 400  
  of `Comparable`, 615  
  of `String`, 393  
`Compilable` interface, `compile` method, 437  
Compilation tasks, 443–449  
`CompilationTask` interface, 443–445  
  `call` method, 445, 448  
`compile` method (`Pattern`), 134, 139  
`CompiledScript` interface, `eval` method, 437  
Compiler  
  annotations for, 471  
  invoking, 443  
  just-in-time, 980  
  producing class files on disk, 445  
Compiler API, 443–454  
`compiler/ByteArrayJavaClass.java`, 446  
`compiler/CompilerTest.java`, 452  
`compiler/MapClassLoader.java`, 454  
`compiler/StringBuilderJavaSource.java`, 446  
Complex types, 172  
`complexType` element (XML Schema), 173  
Composite, 809  
`composite/CompositeComponent.java`, 814  
`composite/CompositeTestFrame.java`, 812  
`composite/Rule.java`, 815  
Composition rules, 766–767, 807–817  
*Computer Graphics: Principles and Practice*  
  (Foley), 775, 809, 836  
`concat` method (`Stream`), 11  
Confidential information, transferring, 567  
Configuration files, 126  
`connect` method  
  of `Socket`, 239  
  of `URLConnection`, 259, 262, 266  
*Connection* interface  
  `close` method, 299, 301, 350  
  `commit` method, 344–346  
  `createBlob`, `createClob` methods, 316, 318  
  `createStatement` method, 298–299, 321, 326,  
    344  
  `get/setAutoCommit` methods, 346  
  `getMetaData` method, 333, 342  
  `getWarnings` method, 304  
  `prepareStatement` method, 310, 315, 322, 326  
  `releaseSavepoint` method, 345–346  
  `rollback` method, 344–346  
  `setSavepoint` method, 346  
Connections (databases)  
  closing, 302  
    using row sets after, 329  
  debugging, 279  
  pooling, 350  
  starting new threads, 246  
  `console` method (`System`), 406  
Constructive area geometry operations, 786  
Constructor class, 457  
Constructors  
  annotating, 466  
  invoking from native code, 965–966  
  type use annotations in, 468  
`containsAll` method (`Collection`), 525  
Content types, 260  
`ContentHandler` class, 199, 201  
  `characters` method, 200, 204  
  `start/endDocument` methods, 204  
  `start/endElement` methods, 200–204  
Context class loader, 495–496  
Control points  
  dragging, 777  
  of curves, 774  
  of shapes, 776  
`convertXxxIndexToModel` methods (`JTable`), 614,  
    623  
Convolution operation, 844  
`ConvolveOp` class, 843–845  
  constructor, 851  
Coordinate system  
  custom, 767  
  translating, 856  
Coordinate transformations, 799–804

- Copies class, 879–881  
     getValue method, 882  
 CopiesSupported class, 879  
 copy method (*Files*), 106, 108  
 CORBA (Common Object Request Broker  
     Architecture), Java support of, 494  
*Core Swing* (Topley), 600, 639, 652, 682  
 count method (*Stream*), 3–4, 12, 191  
*Count of Monte Cristo, The* (Dumas), 727  
 counting method (*Collectors*), 29  
 Country codes, 374  
 CRC32 class, 118  
 CRC32 checksum, 79, 116, 118  
 CREATE TABLE statement (SQL), 290  
     executing, 298, 300, 316  
     in batch updates, 345  
 createBindings method (*ScriptEngine*), 433  
 createBlob, createClob methods (*Connection*), 316, 318  
 createFile, createDirectory, createDirectories  
     methods (*Files*), 105–106  
 createGraphics method (*SplashScreen*), 926  
 createImageXxxStream methods (*ImageIO*), 825, 832  
 createPrintJob method (*PrintService*), 874, 877  
 createStatement method (*Connection*), 298–299, 321, 326, 344  
 createTempXxx methods (*Files*), 105–106  
 createTransferable method (*TransferHandler*), 909, 912  
 createXMLStreamReader method (*XMLInputFactory*), 207  
 createXMLStreamWriter method (*XMLOutputFactory*), 214, 221  
 createXxx methods (*Document*), 209, 212  
 createXxxRowSet methods (*RowSetFactory*), 329, 333  
 creationTime method (*BasicFileAttributes*), 109  
 Credit card numbers, transferring, 567  
 crypt program, 570  
*Cryptography and Network Security* (Stallings), 548, 86  
 Cubic curves, 774–775  
 CubicCurve2D class, 769, 774  
 CubicCurve2D.Double class, 785  
 Currencies, 384–385  
     available, 385  
     formatting, 378–383, 691  
     identifiers for, 384  
 Currency class, 384–385  
     getAvailableCurrencies method, 385  
     getCurrencyCode method, 385  
     getDefaultFractionDigits method, 385  
     getInstance method, 384–385  
     getSymbol method, 385  
     toString method, 385  
 Cursor, drop shapes of, 904  
 curveTo method (*Path2D.Float*), 775, 786  
 Custom editors, 629  
 Custom formatters, 693–703  
 Cut and paste, 887–903  
 Cygwin, 944  
     compiling invocation API, 984  
 OpenSSL in, 560
- D**
- D (double), type code, 87, 961  
 d literal (SQL), 318  
 \d, \D, \d, in regular expressions, 131  
 Dashed lines, dash pattern, dash phase, 790  
 Data flavors, 892–894  
 Data sources, 349  
 Data transfer, 887–903  
     between Java and native code, 888  
     classes and interfaces for, 888  
     in Swing, 904–921  
 Data types  
     codes for, 87, 961  
     in Java vs. C, 947  
     mangling names of, 961  
     print services for, 874–875  
 database.properties file, 305, 349  
 DatabaseMetaData interface, 333–343  
     getJDBCXxxVersion methods, 343  
     getMaxConnection method, 343  
     getMaxStatements method, 301, 343  
     getSQLStateType method, 302  
     getTables method, 333, 342  
     supportsBatchUpdates method, 345, 347  
     supportsResultSetXxx methods, 323, 328  
 Databases, 281–350  
     accessing, in C language, 282  
     autonumbering keys in, 320  
     avoiding duplication of data in, 287  
     batch updates for, 345–347  
     caching prepared statements, 311  
     changing data with SQL, 290  
     connections to, 292, 294–297, 306  
     closing, 301, 306  
     in web and enterprise applications, 349–350  
     pooling, 350

drivers for, 283–284  
error handling in, 346  
integrity of, 344  
LOBs in, 316–318  
metadata for, 333–343  
modifying, 330  
native storage for XML in, 349  
numbering columns in, 299  
outer joins in, 319  
populating, 305–309  
saving objects to, 474  
scalar functions in, 319  
schemas for, 342  
setting up parameters in, 330  
starting, 293  
stored procedures in, 319  
storing:  
  class files, 444  
  logins, 537  
structure of, 285, 333  
synchronization of, 331  
tools for, 334  
truncated data from, 303  
URLs of, 292  
*DataFlavor* class, 888, 892–894  
  constructor, 893  
  *getHumanPresentableName* method, 894  
  *getMimeType* method, 893  
  *getRepresentationClass* method, 894  
  *imageFlavor* constant, 895  
  *isMimeTypeEqual* method, 894  
  *javaFileListFlavor* constant, 914  
*DataInput* interface, 70  
  *readFully* method, 71  
  *readUTF* method, 70–71  
  *readXxx* methods, 70–71, 81  
  *skipBytes* method, 71  
*DataInputStream* class, 51, 56  
*DataIO* class, *xxxFixedString* methods, 73–74  
*DataOutput* interface, 69  
  *writeUTF* method, 70, 72  
  *writeXxx* methods, 69, 72, 81  
*DataOutputStream* class, 51  
*DataSource* interface, 349  
*DataTruncation* class, 303  
  *getIndex*, *getParameter* methods, 304  
  *getTransferSize* method, 305  
Date and Time API, 351–370  
  and legacy code, 369–370  
  Date class, 87, 369  
    formatting objects of, 386  
    *read/writeObject* methods, 93  
  DATE data type (SQL), 291, 318, 348  
  Date pickers, 705  
  *DateEditor* class, 712  
  *dateFilter* method (*RowFilter*), 615, 625  
  *DateFormat* class, 387  
    *getXxxInstance*, *setLenient* methods, 691  
  *dateFormat/DateTimeFormatterTest.java*, 388  
  *dateFormat/EnumCombo.java*, 391  
  Dates  
    computing, 358–360  
    editing, 691  
    filtering, 615  
    formatting, 365–368, 372, 385–393  
    lenient, 691  
    literals for, 318  
    local, 355–358  
    parsing, 367  
  *DateTimeFormatter* class, 365–368, 385–393  
    and legacy classes, 369  
    *format* method, 365, 392  
    *ofLocalizedXxx* methods, 365, 385, 392  
    *ofPattern* method, 367  
    *parse* method, 367  
    *toFormat* method, 367  
    *withLocale* method, 365, 392  
  *DateTimeSyntax* class, 882  
  Daylight savings, 361–365  
  DayOfWeek enumeration, 357  
    *getDisplayName* method, 365, 386  
  *dayOfWeekInMonth* method (*TemporalAdjusters*), 359  
  DDL statement (SQL), 300, 316  
  Debugging  
    in JNI, 980  
    JDBC-related problems, 296  
    mail connections, 279  
    of locales, 377  
    policy files, 564  
    streams, 11  
    with telnet, 233–236  
  DECIMAL data type (SQL), 291, 348  
  Decimal separators, 372, 378, 686  
  Declaration annotations, 466–468  
  *decode* method (*URLDecoder*), 277  
  Decryption key, 498  
  default statement, 462

DefaultCellEditor class, 654  
constructor, 638  
variations, 629  
defaultCharset method (`Charset`), 68, 406  
DefaultFormatter class  
extending, 693  
get/setOverwriteMode methods, 692, 702  
toString method, 691  
DefaultHandler class, 201  
DefaultListModel class, 594  
add/removeElement methods, 595  
DefaultMutableTreeNode class, 641, 659–661  
add method, 642, 650  
constructor, 650  
depthFirstEnumeration method, 659  
pathFromAncestorEnumeration method, 660  
setAllowsChildren, setAsksAllowsChildren methods, 648, 650  
setAsksAllowsChildren method, 648  
`XxxFirstEnumeration`, `XxxOrderEnumeration` methods, 659, 663  
defaultPage method (`PrinterJob`), 861  
DefaultRowSorter class  
setComparator, setSortable methods, 614, 624  
setRowFilter method, 615, 624  
DefaultTableCellRenderer class, 626–627  
DefaultTableModel class, isCellEditable method, 628  
DefaultTreeCellRenderer class, 661–664  
setXxxIcon methods, 664  
DefaultTreeModel class, 650, 673  
automatic notification by, 652  
getPathToRoot method, 653  
insertNodeInto method, 652, 658  
isLeaf method, 648  
nodeChanged method, 652, 658  
nodesChanged method, 658  
reload method, 653, 658  
removeNodeFromParent method, 652, 658  
defaultWriteObject method (`ObjectOutputStream`), 92  
defineClass method (`ClassLoader`), 498, 503  
Delayed formatting, 888  
delete method, 107–108  
DELETE statement (SQL), 290  
executing, 298, 300, 316  
in batch updates, 345  
vs. methods of `ResultSet`, 325  
DeleteGlobalRef function (C), 958  
deleteIfExists method (`Files`), 108  
deleteRow method (`ResultSet`), 325, 328  
Delimiters, in text files, 63  
Deployment directory, 564  
@Deprecated annotation, 470–471  
Depth-first enumerations, 659  
depthFirstEnumeration method (`DefaultMutableTreeNode`), 659, 663  
derbyclient.jar file, 292  
DES (Data Encryption Standard), 568  
Desktop class, 921, 927–932  
browse method, 932  
edit method, 927, 932  
getDesktop method, 927, 931  
isDesktopSupported method, 927, 931  
isSupported method, 927, 931  
mail method, 932  
open method, 927, 932  
print method, 927, 932  
Desktop applications, launching, 927–932  
Desktop panes, 741–760  
cascading/tiling, 744–748  
`desktopApp/DesktopAppFrame.java`, 928  
DesktopManager class, 751  
DestroyJavaVM function (C), 981, 985  
Device coordinates, 799  
*Diagnostic* interface  
getKind method, 448  
getMessage method, 449  
getSource method, 448  
getXxxNumber methods, 449  
DiagnosticCollector class, 444  
constructor, 448  
getDiagnostics method, 448  
*DiagnosticListener* interface, 444  
DialogCallbackHandler class, 538  
Dialogs  
cascading/tiling, 744–748  
in internal frames, 750  
validating input fields before closing, 688  
digest method (`MessageDigest`), 549–550  
Digital signatures, 546–567  
verifying, 553–556  
Direct buffers, 973  
Directories  
creating, 105–106  
current, 114  
hierarchical structure of, 639  
printing all subdirectories of, 113  
traversing, 110–114  
user's working, 56  
DirectoryStream interface, 111  
displayMessage method (`TrayIcon`), 937

distinct method (*Stream*), 11–12, 43  
Dithering, 818  
dividedBy method (Instant, Duration), 353  
`dnd/SampleComponents.java`, 907  
`dnd/SwingDnDTest.java`, 907  
`dndImage/imageListDnDFrame.java`, 915  
doAs, doAsPrivileged methods (Subject), 532–533, 536  
*Doc* interface, 874  
*DocAttribute* interface, 879  
  implementing, 881  
  printing attributes of, 882–885  
*DocAttributeSet* interface, 880–881  
*DocFlavor* class, 874–875, 878  
*DocPrintJob* interface  
  `getAttributes` method, 886  
  `print` method, 877  
DOCTYPE declaration (DTD), 164  
  including in output, 210  
*Document* interface  
  `addDocumentListener` method, 682, 685  
  `createXxx` methods, 209, 212  
  `getDocumentElement` method, 150, 161  
  `getLength` method, 685  
  `getText` method, 685  
  implementing, 681  
Document filters, 688–690  
Document flavors, for print services, 874–875  
Document listeners, 682  
*DocumentBuilder* class  
  `newDocument` method, 209, 212, 227  
  `parse` method, 160  
  `setEntityResolver` method, 164, 170  
  `setErrorHandler` method, 170  
*DocumentBuilderFactory* class  
  `is/setIgnoringElementContentWhitespace` methods, 169, 171  
  `is/setNamespaceAware` methods, 174, 198–199, 201, 209  
  `is/setValidating` method, 171  
  `newDocumentBuilder` method, 150, 160, 209  
  `newInstance` method, 150, 160  
  `setNamespaceAware` method, 209  
  `setValidating` method, 169  
@Documented annotation, 470, 472, 474  
*DocumentEvent* interface, `getDocument` method, 685  
*DocumentFilter* class, 689  
  `insertString` method, 688–689, 702  
  `remove` method, 702  
  `replace` method, 689, 702  
*DocumentListener* interface, `XxxUpdate` methods, 682, 685  
doFinal method (Cipher), 568, 571, 573, 575  
DOM (Document Object Model) parser, 149–150, 199  
  supporting PUBLIC identifiers in, 164  
DOM (Document Object Model) tree  
  accessing with XPath, 190–196  
  analyzing, 152–154  
  building, 208–222  
  writing, 210–211  
`dom/TreeViewer.java`, 155  
DOMResult class, 227, 232  
DOMSource class, 213, 226  
DOUBLE data type (SQL), 291, 348  
double type  
  printing, 61  
  type code for, 87, 961  
  vs. C types, 947  
  writing in binary format, 69  
*DoubleBuffer* class, 124  
Double-clicking, on list components, 585  
doubles method (Random), 37, 40  
*DoubleStream* interface, 36–41  
  `average`, `max`, `min`, `sum` methods, 37, 40  
  boxed method, 36, 40  
  `mapToDouble` method, 36  
  `of` method, 39  
  `summaryStatistics` method, 37, 40  
  `toArray` method, 36, 40  
*DoubleSummaryStatistics* class, 20, 23, 37, 41  
doubleValue method (Number), 378  
Downstream collectors, 29–33  
Drag and drop, 903–921  
  file lists in, 914  
  in Swing components, 904–921  
  moving vs. copying with, 903  
  visual feedback for, 912  
Drag sources, 903  
  configuring, 909–912  
draw method (Graphics2D), 767–769, 787  
Drawing  
  shapes, 766–769  
  simple, 765  
drawXxx methods (Graphics), 769  
*DriverManager* class, 294  
  `getConnection` method, 295, 297, 306, 350  
  `setLogWriter` method, 296  
Drop actions, 903–904  
Drop locations, 914

- DROP TABLE statement (SQL), 295  
     executing, 298, 300  
     in batch updates, 345
- Drop targets, 903, 912–921
- DropLocation class (of `JList`)  
     `getIndex` method, 914, 920  
     `isInsert` method, 915, 920
- DropLocation class (of `JTable`), `getXxx`, `isInsertXxx`  
     methods, 920
- DropLocation class (of `JTextComponent`), `getIndex`  
     method, 921
- DropLocation class (of `JTree`)  
     `getChildIndex` method, 920  
     `getPath` method, 914, 920
- DropLocation class (of `TransferHandler`), 914  
     `getDropPoint` method, 920
- DSA (Digital Signature Algorithm), 551–552
- DST, DST\_Xxx composition rules, 810
- DTDHandler class, 201
- DTDs (Document Type Definitions),  
     163–171  
     element content in, 165–166  
     entities in, 168  
     external, 164  
     in XML documents, 147, 164  
     unambiguous, 166  
     URLs for, 164
- Duration class  
     arithmetic operations, 353  
     `between` method, 352  
     immutable, 354  
     `toXxx` methods, 353
- Dynamic links, 981
- Dynamic web pages, 449–454
- E**
- \e, \E, in regular expressions, 130
- Echo servers, 244
- Eclipse, startup plugins for, 922
- Edge detection, 845
- edit method (`Desktop`), 927, 932
- Editor pane (Swing), 681, 712–719  
     edit mode of, 714  
     loading pages in separate threads, 715  
`editorPane/EditorPaneFrame.java`, 716
- Editors, custom, 629
- element element (XML Schema), 172
- Element interface, 477  
     `getAttribute` method, 154, 161, 176  
     `get SimpleName` method, 477
- getTagName method, 151, 161, 198  
     `setAttribute`, `setAttributeNS` methods, 209, 212
- ELEMENT element content (DTD), 165–166
- Elements (XML)  
     child, 147  
     accessing in XPath, 191  
     namespace of, 197  
     constructing, 209  
     counting, in XPath, 191  
     empty, 146  
     getting node list of, 154  
     legal attributes of, 167  
     names of, 151, 198  
     root, 147, 172  
     trimming whitespace in, 153  
     vs. attributes, 147–148, 168, 212
- Ellipse2D class, 769
- Ellipses, bounding rectangle of, 770
- E-mails, 927  
     launching from desktop applications, 927  
     sending, 277–280  
     terminating lines in, 278
- employee/Employee.c, 959
- employee/Employee.java, 959
- employee/EmployeeTest.java, 958
- empty method  
     of `Optional`, 16  
     of `Stream`, 5, 8
- EMPTY element content (DTD), 165
- Empty tags (XML), 146
- encode method (`URLEncoder`), 277
- Encryption, 567–579  
     final block padding in, 568  
     of class files, 498–499
- end method (`Matcher`), 135, 137, 140–141
- End cap styles, 788–790
- End points, 774
- End tags (XML), 146
- endDocument method (`ContentHandler`), 204
- endElement method (`ContentHandler`), 200–204
- End-of-line character. *See* Line feed
- Enterprise applications, 349–350
- Enterprise JavaBeans (EJBs), 285  
     hot deployment of, 496
- Entity references (XML), 148, 168
- Entity resolvers, 150
- ENTITY, ENTITIES attribute types (DTDs), 167–168
- EntityResolver class, 201  
     `resolveEntity` method, 164, 170
- entries method (`ZipFile`), 80

Entry class, 616  
  `getXxx` methods, 625  
EntryLogger class, 487  
`EntryLoggingAgent.mf` file, 487  
enum keyword, 94  
EnumCombo class, 387  
enumeration element (XML Schema), 172  
*Enumeration* interface, 80  
  `hasMoreElements` method, 987–989  
  `nextElement` method, 659, 987–990  
Enumerations  
  of nodes, in a tree, 659–661  
  typesafe, 94–95  
  using attributes for, 168  
`EnumSyntax` class, 882  
`EOFException`, 974  
Epoch, 93, 352  
equals method  
  of *Annotation*, 464  
  of *Collator*, 399  
  of *Permission*, 523  
  of *Set*, 525  
Error handlers  
  in native code, 974–979  
  installing, 170  
Error messages  
  control over, 445  
  listening to, 444  
`ErrorHandler` class, 201  
  `error`, `fatalError`, `warning` methods,  
    170–171  
Escape hatch mechanism, 650  
escape keyword (SQL), 319  
Escapes  
  in regular expressions, 64, 129  
  in SQL, 318–319  
*Essential XML* (Box et al.), 143, 222  
Euro symbol, 384, 406  
eval method  
  of *CompiledScript*, 437  
  of *ScriptEngine*, 431–433  
evaluate method (*XPath*), 191, 196  
Event handlers, annotating, 457–462  
Event listeners, 178, 455  
Event queues (AWT), accessing, 509  
EventHandler class, 458  
EventListenerList class, 673  
Events, transitional, 584  
Evins, Jim, 609  
evn pointer (C), 950

Exceptions  
  checked, 533  
  from native code, 974–979  
  in C++, 975  
  in SQL, 302–305  
  type use annotations in, 468  
`ExceptionXxx` functions (C), 975–976, 979  
Exclusive lock, 127  
exclusiveOr method (*Area*), 787–788  
`exec/ExecSQL.java`, 306  
*ExecutableElement* interface, 477  
execute method  
  of *RowSet*, *CachedRowSet*, 330, 332  
  of *Statement*, 300, 306, 319, 321  
executeBatch method (*Statement*), 346–347  
executeLargeBatch method (*Statement*), 347  
executeQuery method  
  of *PreparedStatement*, 310, 316  
  of *Statement*, 298, 300, 323  
executeUpdate method  
  of *PreparedStatement*, 310, 316  
  of *Statement*, 298, 300, 321, 344  
exists method (*File*s), 108–109  
EXIT statement (SQL), 293  
exit method (*System*), 509  
exportAsDrag method (*TransferHandler*), 910, 912  
exportDone method (*TransferHandler*), 911–912  
Extension class loader, 493  
extern "C", in native methods (C++), 943  
External entities, 168  
*Externalizable* interface, *xxxExternal* methods,  
  93–94

## F

`F` (`float`), type code, 87, 961  
`\f`, in regular expressions, 130  
Factoring algorithms, 552  
`fatalError` method (*ErrorHandler*), 170–171  
Field class  
  `getName`, `getType` methods, 674  
  implementing *AnnotatedElement*, 457  
Fields  
  accessing:  
    from another class, 509  
    from native code, 956–961  
  annotating, 456, 470  
  transient, 92  
File class, `toPath` method, 102–103  
File lists, 914  
File permissions, 520

- File pointers, 72  
File systems, POSIX-compliant, 109  
`file:` URI scheme, 257, 517  
`file.encoding` system property, 68  
`file.separator` property, 521  
`FileChannel` class  
    `lock`, `tryLock` methods, 126–128  
    `open`, `map` methods, 117, 122  
`FileHandler` class, 406  
`FileInputStream` class, 55–59, 513, 523, 726  
    constructor, 58  
    `getChannel` method, 121  
    `read` method, 48  
`fileKey` method (*BasicFileAttributes*), 109  
`FileLock` class  
    `close` method, 128  
    `isShared` method, 127  
`FileNotFoundException`, 273  
`FileOutputStream` class, 55–59  
    constructor, 59  
    `getChannel` method, 121  
`FilePermission` class, 511, 517  
`FileReader` class, 513  
`Files`  
    accessing, 509  
    channels for, 117  
    closing, 110  
    configuration, 126  
    copying, 106  
    counting lines in, 727  
    creating, 105–106  
    deleting, 107  
    drag and drop of, 904–905  
    encrypting/decrypting, 574  
    filtering, 112  
        by file suffixes, 825  
    generated automatically, 455, 475  
    handling, 445  
    hierarchical structure of, 639  
    I/O modes of, 76  
    in desktop applications, 927  
    locating, 444  
    locking, 126–128  
    memory-mapped, 116–128  
    moving, 106  
    random-access, 72–76  
        vs. buffered, 116  
    reading, 104–105  
        as a string, 104  
        by one byte, 48–51  
    numbers from, 56  
    permissions for, 523  
    total number of bytes in, 73  
    traversing, 111–115  
    with multiple images, 825–834  
    writing, 104–105  
`Files` class, 100, 104–114  
    `copy` method, 106, 108  
    `createXxx` methods, 105–106  
    `delete` method, 107–108  
    `deleteIfExists` method, 108  
    `exists` method, 108–109  
    `find` method, 111  
    `getBytes` method, 104  
    `getOwner` method, 108  
    `isXxx` methods, 108–109  
    `lines` method, 6, 8  
    `list` method, 110  
    `move` method, 106, 108  
    `newDirectoryStream` method, 111, 114  
    `newXxxStream`, `newBufferedXxx` methods, 104–105  
    `readAllXxx` methods, 104  
    `readAttributes` method, 109  
    `size` method, 108–109  
    `walk` method, 110  
    `write` method, 104  
`Files` class, `walkFileTree` method, 112–114  
`FileSystem` class, `getPath` method, 115–116  
`FileSystems` class, `newFileSystem` method, 115–116  
`FileTime` class, and legacy classes, 369  
`FileVisitor` interface, 113–114  
    `visitFile`, `visitFileFailed` methods, 112  
    `xxxVisitDirectory`, `visitXxx` methods, 112  
`fill` method (`Graphics2D`), 767–768, 787  
`Filling`, 766–767, 797  
`fillXxx` methods (`Graphics`), 769  
`filter` method  
    of `BufferedImageOp`, 842, 850  
    of `Stream`, 3–4, 9–10, 13  
`FilteredRowSet` interface, 329  
`Filters`  
    for images, 842–851  
    for numbers, 615  
    for streams, 726  
    for table rows, 615–617  
    for user input, 688–690  
    glob patterns for, 112  
    implementing, 616  
`FilterXxxStream` classes, 57  
`Final` block padding, 568

find method  
  of *Files*, 111  
  of *Matcher*, 137, 140  
findAny method (*Stream*), 13  
findClass method (*ClassLoader*), 498, 503  
FindClass function (C), 957, 960, 965  
findColumn method (*ResultSet*), 301  
findXxx methods (*Stream*), 13  
Fingerprints, 86, 546–567  
  different for a class and its objects, 89  
fireStateChanged method (*AbstractSpinnerModel*), 706  
first method (*ResultSet*), 323, 327  
firstDayOfXxx methods (*TemporalAdjusters*), 359  
#FIXED attribute (DTD), 167  
Fixed-size patterns, 692  
Fixed-size records, 73–74  
flatMap method  
  of *Optional*, 16–19  
  of *Stream*, 10  
*FlavorListener* interface, flavorsChanged method, 893–894  
flavormap.properties file, 887  
flip method (*Buffer*), 125–126  
FLOAT data type (SQL), 291, 348  
float type  
  printing, 61  
  type code for, 87, 961  
  vs. C types, 947  
  writing in binary format, 69  
FloatBuffer class, 124  
Floating-point numbers, 372, 378–383, 691  
flush method  
  of *CipherOutputStream*, 575  
  of *Closeable*, 52  
  of *Flushable*, 52, 55  
  of *OutputStream*, 49, 51  
*Flushable* interface, 52–53  
  close method, 52  
  flush method, 52, 55  
fn keyword (SQL), 319  
Focus listeners, 683  
Folder icons, 648, 661  
Font render context, 805  
Fonts  
  antialiasing, 817, 819  
  displaying, 596  
forEach method (*Stream*), 19, 22  
Forest (Swing), 639, 647  
forLanguageTag method (*Locale*), 377  
format method  
  of *DateTimeFormatter*, 365, 392  
  of *Format*, 402  
  of *MessageFormat*, 401–402  
  of *NumberFormat*, 379, 383  
Formatters  
  and locales, 373  
  custom, 693–703  
  installing, 689  
  supported by *JFormattedTextField*, 691–693  
Formatting  
  copy and paste, 887  
  delayed, 888  
  of currencies, 691  
  of dates, 372, 385–393  
  of messages, 400–404  
  of numbers, 372, 378–383, 691  
formatting/Formatting.java, 368  
Forms, processing, 267–277  
forName method (*Charset*), 69  
fortune program (UNIX), 933  
ForwardingJavaFileManager class  
  constructor, 449  
  getFileForOutput method, 449  
  getJavaFileForOutput method, 445, 450  
fprintf function (C), 963  
Fractals, 838  
Frame class, 450  
Frames (Swing)  
  cascading/tiling, 744–748  
  closing, 748  
  dialogs in, 750  
  dragging, 751–760  
  grabbers of, 742  
  icons for, 743  
  internal, 741–752  
  managing on desktop, 751  
  positioning, 744  
  resizable, 745  
  selected, 744, 747  
  states of, 745  
  visibility of, 744  
  with two nested split panes, 732  
from method (*Instant*), 369  
FROM statement (SQL), 288  
FTP (File Transfer Protocol), 262  
ftp: URI scheme, 257, 262, 713  
Function interface, identity method, 24  
@FunctionalInterface annotation, 470

**G**

- \G, in regular expressions, 132
- Garbage collection
  - and arrays, 972
  - and native methods, 951
- GeneralPath class, 769, 775
  - constructor, 785
- generate method (*Stream*), 5, 8, 36
- @Generated annotation, 470–471
- generateKey method (*KeyGenerator*), 569, 574
- Generic types, type use annotations in, 467
- Gestures (Swing), 903
- get method
  - of *AttributeSet*, 882, 886
  - of *Bindings*, 434
  - of *ByteBuffer*, 117, 122
  - of *CharBuffer*, 124
  - of *Optional*, 15–16
  - of *Paths*, 101, 103
  - of *ScriptEngine*, 433
  - of *ScriptEngineManager*, 433
  - of *Supplier*, 8
- GET method (HTML), 269–270
- getActions method (*Permission*), 523
- getAddress method (*InetAddress*), 240–241
- getAdvance method (*TextLayout*), 807
- getAllByName method (*InetAddress*), 240–241
- getAllFrames method (*JDesktopPane*), 745–748, 757
- getAllowsChildren method (*TreeNode*), 649
- getAllowUserInteraction method (*URLConnection*), 266
- getAnnotation method (*AnnotatedElement*), 457, 462, 475, 477
- getAnnotations, getDeclaredAnnotations methods
  - (*AnnotatedElement*), 462
- getAnnotationsByType method (*AnnotatedElement*), 462, 475, 477
- GetArrayLength function (C), 972–973
- getAscent method (*TextLayout*), 807
- getAsXxx methods (*OptionalXxx* classes), 37, 41
- getAttribute method (*Element*), 154, 161, 176
- getAttributes method
  - of *DocPrintJob*, 886
  - of *Node*, 154, 161
  - of *PrintService*, 886
- getAttributeXxx methods (*XMLStreamReader*), 206, 208
- getAuthority method (*URI*), 258
- getAutoCommit method (*Connection*), 346
- getAutoCreateRowSorter method (*JTable*), 602, 604
- getAvailableCurrencies method (*Currency*), 385
- getAvailableDataFlavors method (*Clipboard*), 894
- getAvailableLocales method
  - of *Collator*, 399
  - of *NumberFormat*, 376, 379, 383
- getAverage method (*XxxSummaryStatistics*), 20, 23, 41
- getBackground method (*JList*), 596, 599
- getBinaryStream method (*Blob*), 316–317
- getBlob method (*ResultSet*), 316–317
- getBlockSize method (*Cipher*), 573
- GetBooleanArrayElements function (C), 972–974
- GetBooleanArrayRegion function (C), 973–974
- GetBooleanField function (C), 960
- getBounds method (*SplashScreen*), 926
- getBundle method (*ResourceBundle*), 409–412
- getByName method (*InetAddress*), 240–241
- GetByteArrayElements function (C), 972–974, 989
- GetByteArrayRegion function (C), 973–974
- GetByteField function (C), 960
- getBytes method
  - of *Blob*, 316–317
  - of *Files*, 104
- getCandidateLocales method ( *ResourceBundle.Control*), 410
- getCategory method (*Attribute*), 881, 886
- getCellEditorValue method (*CellEditor*), 629, 631–632, 638
- getCellSelectionEnabled method (*JTable*), 622
- getCertificates method (*CodeSource*), 514
- getChannel method (*FileXxxStream*, *RandomAccessFile*), 121
- getChar method (*ByteBuffer*), 117, 123
- getCharacterStream method (*Clob*), 316, 318
- GetCharArrayElements function (C), 972–974
- GetCharArrayRegion function (C), 973–974
- getCharContent method (*SimpleJavaFileObject*), 445, 449
- GetCharField function (C), 960
- getChild method (*TreeModel*), 154, 672–674, 680
- getChildAt method (*TreeNode*), 658
- getChildCount method
  - of *TreeModel*, 672–674, 679
  - of *TreeNode*, 658
- getChildIndex method (*JTree.DropLocation*), 920
- getChildNodes method (*Node*), 151, 161
- getClassLoader method (*Class*), 493, 503
- getClip method (*Graphics*), 806, 854
- getBlob method (*ResultSet*), 316–317
- getCodeSource method (*ProtectionDomain*), 514
- getCollationKey method (*Collator*), 395, 399

getColorModel method (*BufferedImage*), 837, 840  
getColumn method  
  of *JTable.DropLocation*, 920  
  of *TableColumnModel*, 623  
getColumnClass method (*TableModel*), 609, 621  
getColumnCount method  
  of *ResultSetMetaData*, 334  
  of *TableModel*, 605–606, 608  
getColumnModel method (*JTable*), 622  
getColumnName method (*TableModel*), 606, 608  
getColumnNumber method  
  of *Diagnostic*, 449  
  of *SAXParseException*, 171  
getColumnSelectionAllowed method (*JTable*), 622  
getColumnXxx methods (*ResultSetMetaData*), 334, 343  
getCommand method (*RowSet*), 332  
getComponent method (*TransferSupport*), 919  
getComponentAt method (*JTabbedPane*), 740  
getConcurrency method (*ResultSet*), 323–324, 327  
getConnection method (*DriverManager*), 295, 297,  
  306, 350  
getConnectTimeout method (*URLConnection*), 266  
getContent method (*URLConnection*), 267  
getContentPane method (*JInternalFrame*), 758  
getContents method (*Clipboard*), 891  
getContentXxx methods (*URLConnection*), 260, 263,  
  267  
getContext method  
  of *AccessController*, 536  
  of *ScriptEngine*, 434  
getContextClassLoader method (*Thread*), 495, 503  
getCount method (*XxxSummaryStatistics*), 23, 41  
getCountry method (*Locale*), 377  
getCurrencyCode method (*Currency*), 385  
getCurrencyInstance method (*NumberFormat*), 378,  
  383–384, 691  
getData method  
  of *CharacterData*, 153, 162  
  of *Clipboard*, 892  
getDataElements method  
  of *ColorModel*, 842  
  of *Raster*, 837, 841  
getDataFlavors method (*TransferSupport*), 919  
getDate method  
  of *ResultSet*, 299, 301  
  of *URLConnection*, 260, 263, 267  
getDateInstance, getDateDateTimeInstance methods  
  (*DateFormat*), 691  
getDayOfMonth methods (Date and Time API),  
  356–357, 363  
getDecomposition method (*Collator*), 399  
getDefault method (*Locale*), 376–377  
getDefaultEditor method (*JTable*), 637  
getDefaultFractionDigits method (*Currency*), 385  
getDefaultName method (*NameCallback*), 545  
getDefaultRenderer method (*JTable*), 627, 637  
getDescent method (*TextLayout*), 807  
getDesktop method (*Desktop*), 927, 931  
getDesktopPane method (*JInternalFrame*), 759  
getDiagnostics method (*DiagnosticCollector*), 448  
GetDirectBufferXxx functions (C), 973  
getDisplayCountry, getDisplayLanguage methods  
  (*Locale*), 377  
getDisplayName method  
  of *DayOfWeek*, 365, 386  
  of *Locale*, 376–377, 379  
  of *Month*, 365, 386  
getDocument method (*DocumentEvent*), 685  
getDocumentElement method (*Document*), 150, 161  
getDocumentFilter method (*AbstractFormatter*), 689,  
  701  
getDouble method  
  of *ByteBuffer*, 117, 123  
  of *ResultSet*, 299, 301  
GetDoubleArrayElements function (C), 972–974  
GetDoubleArrayRegion function (C), 973–974  
GetDoubleField function (C), 956, 960  
getDoXxx methods (*URLConnection*), 259, 265  
getDropAction method (*TransferSupport*), 919  
getDropLocation method (*TransferSupport*), 914, 919  
getDropPoint method (*TransferHandler.DropLocation*),  
  920  
getElementAt method (*ListModel*), 589, 593  
getEnclosedElements method (*TypeElement*), 477  
getEngineXxx methods (*ScriptEngineManager*), 431  
getEntry method (*ZipFile*), 80  
getErrorCode method (*SQLException*), 302, 304  
getErrorStream method (*HttpURLConnection*), 273, 277  
getErrorHandler method (*ScriptContext*), 434  
getExpiration method (*URLConnection*), 260, 263,  
  267  
getExtension method (*ScriptEngineFactory*), 431  
GetFieldID function (C), 957, 960  
getFields method (*Class*), 674  
getFileForOutput method (*ForwardingJavaFileManager*),  
  449  
getFileName method (*StackTraceElement*), 103  
getFilePointer method (*RandomAccessFile*), 72, 76  
getFileSuffixes method (*ImageReaderWriterSpi*), 833  
getFillsViewportHeight method (*JTable*), 604

getFirstChild method (*Node*), 153, 161  
getFloat method (*ByteBuffer*), 117, 123  
GetFloatArrayElements function (C), 972–974  
GetFloatArrayRegion function (C), 973–974  
GetFloatField function (C), 960  
getFocusLostBehavior method (*JFormattedTextField*), 688, 701  
getFontRenderContext method (*Graphics2D*), 805, 807  
getForeground method (*JList*), 596, 599  
getFormatNames method (*ImageReaderWriterSpi*), 833  
getFragment method (*URI*), 258  
getFrameIcon method (*JInternalFrame*), 759  
getHeaderXxx methods (*URLConnection*), 260–262, 267  
getHeight method  
  of *ImageReader*, 826, 833  
  of *PageFormat*, 854, 861  
getHost method (*URI*), 258  
getHostXxx methods (*InetAddress*), 241  
getHour method (Date and Time API), 360, 363  
getHumanPresentableName method (*DataFlavor*), 894  
getIconAt method (*JTabbedPane*), 740  
getIdentifier method (*Entry*), 625  
getIfModifiedSince method (*URLConnection*), 266  
getImage method (*TrayIcon*), 936  
getImageableXxx methods (*PageFormat*), 855, 861–862  
getImageURL method (*SplashScreen*), 926  
getImageXxxByXxx methods (*ImageIO*), 824, 832  
getIndex method  
  of *DataTruncation*, 304  
  of *JList.DropLocation*, 914, 920  
  of *JTextComponent.DropLocation*, 921  
getIndexOfChild method (*TreeModel*), 672, 680  
getInputStream method  
  of *Socket*, 237–238, 242  
  of *URLConnection*, 260, 267, 271, 273  
  of *ZipFile*, 80  
getInstance method  
  of *AlphaComposite*, 809, 817  
  of *Cipher*, 567, 573  
  of *Collator*, 399  
  of *Currency*, 384–385  
  of *KeyGenerator*, 574  
  of *Locale*, 393  
  of *MessageDigest*, 548–550  
getInt method  
  of *ByteBuffer*, 117, 123  
  of *ResultSet*, 299, 301  
GetIntArrayElements function (C), 972–974  
GetIntArrayRegion function (C), 973–974  
getIntegerInstance method (*NumberFormat*), 689  
getInterface method (*Invocable*), 436  
GetIntField function (C), 956, 960, 989  
getInvalidCharacters method (*MaskFormatter*), 703  
getJavaFileForOutput method  
  (*ForwardingJavaFileManager*), 445, 450  
getJavaFileObjectsFromXxx methods  
  (*StandardJavaFileManager*), 448  
getJDBCXxxVersion methods (*DatabaseMetaData*), 343  
getKeys method (*ResourceBundle*), 412–413  
getKind method (*Diagnostic*), 448  
getLanguage method (*Locale*), 377  
getLastChild method (*Node*), 154, 161  
getLastModified method (*URLConnection*), 260, 263, 267  
getLastPathComponent method (*TreePath*), 651, 658  
getLastSelectedPathComponent method (*JTree*), 652, 657  
getLayoutOrientation method (*JList*), 587  
getLeading method (*TextLayout*), 807  
getLength method  
  of *Attributes*, 204  
  of *Document*, 685  
  of *NamedNodeMap*, 162  
  of *NodeList*, 151, 162  
getLineNumber method  
  of *Diagnostic*, 449  
  of *SAXParseException*, 171  
getListCellRendererComponent method  
  (*ListCellRenderer*), 595–599  
getLocale method (*MessageFormat*), 402  
getLocalHost method (*InetAddress*), 240–241  
getLocalName method  
  of *Attributes*, 204  
  of *Node*, 198–199  
  of *XMLStreamReader*, 208  
getLocation method (*CodeSource*), 514  
getLong method (*ByteBuffer*), 117, 123  
GetLongArrayElements function (C), 972–974  
GetLongArrayRegion function (C), 973–974  
GetLongField function (C), 960  
getMax method (*XxxSummaryStatistics*), 20, 23, 41  
getMaxConnections method (*DatabaseMetaData*), 343  
getMaximum method (*JProgressBar*), 730  
getMaxStatements method (*DatabaseMetaData*), 301, 343  
getMessage method (*Diagnostic*), 449

getMetaData method  
  of *Connection*, 333, 342  
  of *ResultSet*, 334, 343  
getMethodCallsSyntax method (*ScriptEngineFactory*),  
  435  
GetMethodID function (C), 966, 969  
getMimeType method (*DataFlavor*), 893  
getMimeTypes method (*ScriptEngineFactory*), 431  
getMIMETypes method (*ImageReaderWriterSpi*), 833  
getMin method (*XxxSummaryStatistics*), 23, 41  
getMinimum method (*JProgressBar*), 730  
getMinute method (Date and Time API), 360,  
  363  
getMnemonicAt method (*JTabbedPane*), 741  
getModel method  
  of *Entry*, 625  
  of *JList*, 595  
getMonth, getMonthValue methods (Date and Time  
  API), 356, 363  
getMoreResults method (*Statement*), 320  
getName method  
  of *Attribute*, 886  
  of *Field*, 674  
  of *NameCallback*, 545  
  of *Permission*, 525, 530  
  of *Principal*, 537  
  of *PrintService*, 874  
  of *UnixPrincipal*, 531  
  of *XMLStreamReader*, 208  
  of *ZipEntry*, 79  
  of *ZipFile*, 80  
getNames method (*ScriptEngineFactory*), 431  
getNamespaceURI method (*Node*), 198–199  
getNano method (Date and Time API), 360,  
  363  
getNewValue method (*PropertyChangeEvent*), 750,  
  759  
getNextEntry method (*ZipInputStream*), 77–78  
getNextException method (*SQLException*), 302, 304  
getNextSibling method (*Node*), 154, 161  
getNextValue method (*AbstractSpinnerModel*),  
  705–706, 712  
getNextWarning method (*SQLWarning*), 304  
getNodeXxx methods (*Node*), 154, 161, 198  
getNumberInstance method (*NumberFormat*), 378, 383,  
  691  
getNumXxx methods (*ImageReader*), 826, 833  
getObject method  
  of *ResourceBundle*, 411–412  
  of *ResultSet*, 299, 301  
GetObjectArrayElement function (C), 972–973  
GetObjectClass function (C), 957–958  
GetObjectField function (C), 956, 960  
getOffset method (*ZonedDateTime*), 363  
getOrientation method (*PageFormat*), 862  
getOriginatingProvider method  
  of *ImageReader*, 824, 833  
  of *ImageWriter*, 834  
getOutputSize method (*Cipher*), 573  
getOutputStream method  
  of *Socket*, 238, 242  
  of *URLConnection*, 260, 267, 270  
getOverwriteMode method (*DefaultFormatter*), 702  
getOwner method (*File*), 108  
getPageCount method (*Banner*), 863  
getPageSize method (*CachedRowSet*), 332  
getParameter method (*DataTruncation*), 304  
getParent method  
  of *ClassLoader*, 503  
  of *Path*, 103  
  of *TreeNode*, 658, 660  
getParentNode method (*Node*), 161  
getPassword method  
  of *PasswordCallback*, 546  
  of *RowSet*, 331  
getPath method  
  of *FileSystem*, 115–116  
  of *JTree.DropLocation*, 914, 920  
  of *TreeSelectionEvent*, 671  
  of *URI*, 258  
getPaths method (*TreeSelectionEvent*), 665, 671  
getPathToRoot method (*DefaultTreeModel*), 653  
getPercentInstance method (*NumberFormat*), 378, 383,  
  691  
getPixel, getPixels methods (*Raster*), 836, 841  
getPlaceholder, getPlaceholderCharacter methods  
  (*MaskFormatter*), 703  
getPointCount method (*ShapeMaker*), 776  
getPopupMenu method (*TrayIcon*), 936  
getPort method (*URI*), 258  
getPreferredSize method (*JComponent*), 596–597,  
  685  
getPreviousSibling method (*Node*), 161  
getPreviousValue method (*AbstractSpinnerModel*),  
  705–706, 712  
getPrincipals method (*Subject*), 536  
getPrintable method (*Book*), 873  
getPrinterJob method (*PrinterJob*), 852, 861  
getPrintService method (*StreamPrintServiceFactory*),  
  878–879

**getPrompt** method  
 of *NameCallback*, 545  
 of *PasswordCallback*, 546  
**getPropertyName** method (*PropertyChangeEvent*), 759  
**getProtectionDomain** method (*Class*), 514  
**getPrototypeCellValue** method (*JList*), 593  
**getQName** method (*Attribute*), 204  
**getQualifiedName** method (*TypeElement*), 477  
**getQuery** method (*URI*), 258  
**getRaster** method (*BufferedImage*), 835, 840  
**getReader** method (*ScriptContext*), 434  
**getReaderXxx** methods (*ImageIO*), 825, 832  
**getReadTimeout** method (*URLConnection*), 266  
**getRepresentationClass** method (*DataFlavor*), 894  
**getRequestProperties** method (*URLConnection*), 266  
**getResultSet** method (*Statement*), 300  
**getRGB** method  
 of *Color*, 842  
 of *ColorModel*, 837, 842  
**getRoot** method  
 of *Path*, 103  
 of *TreeModel*, 154, 672–674, 679  
**getRotateInstance** method (*AffineTransform*), 802–803  
**getRow** method  
 of *JTable.DropLocation*, 920  
 of *ResultSet*, 323, 327  
**getRowCount** method (*TableModel*), 605–606, 608  
**getRowXxx** methods (*JTable*), 622  
**getSavepointXxx** methods (*Savepoint*), 347  
**getScaleInstance** method (*AffineTransform*), 802–803  
**getSecond** method (*Date and Time API*), 360, 363  
**getSelectedComponent** method (*JTabbedPane*), 740  
**getSelectedIndex** method (*JTabbedPane*), 737, 740  
**getSelectedValue** method (*JList*), 584, 587  
**getSelectedValuesList** method (*JList*), 585, 587  
**getSelectionMode** method (*JList*), 587  
**getSelectionModel** method (*JTable*), 622  
**getSelectionPath** method (*JTree*), 651, 657, 665, 671  
**getSelectionPaths** method (*JTree*), 665, 671  
**getSelectionXxx** methods (*JList*), 596, 599  
**getShearInstance** method (*AffineTransform*), 802, 804  
**getShort** method (*ByteBuffer*), 117, 123  
**GetShortArrayElements** function (C), 972–974  
**GetShortArrayRegion** function (C), 973–974  
**GetShortField** function (C), 960  
**getSimpleName** method (*Element*), 477  
**getSize** method  
 of *ListModel*, 589, 593  
 of *ZipEntry*, 79  
**getSource** method (*Diagnostic*), 448  
**getSourceActions** method (*TransferHandler*), 909, 912  
**getSourceDropActions** method (*TransferSupport*), 919  
**getSplashScreen** method (*SplashScreen*), 923, 926  
**getSQLState** method (*SQLException*), 302, 304  
**getSQLStateType** method (*SQLException*), 302  
**getStandardFileManager** method (*JavaCompiler*), 447  
**GetStaticFieldID**, *GetStaticXxxField* functions (C), 960–961  
**GetStaticMethodID** function (C), 964, 970  
**getStrength** method (*Collator*), 399  
**getString** method  
 of *JProgressBar*, 730  
 of *ResourceBundle*, 410, 412  
 of *ResultSet*, 299, 301  
**getStringArray** method (*ResourceBundle*), 412  
**GetStringChars**, *GetStringLength* functions (C), 953  
**GetStringRegion**, *GetStringUTFRegion*, *GetStringUTFLength* functions (C), 952  
**GetStringUTFChars** function (C), 951–952, 954, 989  
**getStringValue** method (*Entry*), 625  
**getSubject** method (*LoginContext*), 536  
**getSubString** method (*Clob*), 316–317  
**getSum** method (*XxxSummaryStatistics*), 23, 41  
**GetSuperclass** function (C), 1001  
**getSymbol** method (*Currency*), 385  
**getSystemClassLoader** method (*ClassLoader*), 503  
**getSystemClipboard** method (*Toolkit*), 888, 891  
**getSystemJavaCompiler** method (*ToolProvider*), 443  
**getSystemTray** method (*SystemTray*), 932, 936  
**getTabComponentAt**, *getTabLayoutPolicy* methods  
 (*JTabbedPane*), 741  
**getTabCount** method (*JTabbedPane*), 740  
**getTableCellEditorComponent** method (*TableCellEditor*), 629, 631, 638  
**getTableCellRendererComponent** method  
 (*TableCellEditor*), 626, 637  
**getTableName** method (*CachedRowSet*), 332  
**getTables** method (*DatabaseMetaData*), 333, 342  
**getTagName** method (*Element*), 151, 161, 198  
**getTask** method (*JavaCompiler*), 444, 448  
**Getter/setter pairs.** See *Properties*  
**getText** method  
 of *Document*, 685  
 of *XMLStreamReader*, 208  
**getTimeInstance** method (*DateFormat*), 691  
**getTitleAt** method (*JTabbedPane*), 740

getTooltip method (*TrayIcon*), 936  
getTransferable method (*TransferSupport*), 913  
getTransferData method (*Transferable*), 892  
getTransferDataFlavors method (*Transferable*),  
    894  
getTransferSize method (*DataTruncation*), 305  
getTranslateInstance method (*AffineTransform*), 802,  
    804  
getTrayIconSize method (*SystemTray*), 936  
getTreeCellRendererComponent method  
    (*TreeCellRenderer*), 662–663  
getType method  
    of *Field*, 674  
    of *ResultSet*, 323, 326  
getUpdateCount method (*Statement*), 300, 320  
getURI method (*Attribute*), 204  
getURL method  
    of *HyperlinkEvent*, 715, 719  
    of *RowSet*, 331  
getUseCaches method (*URLConnection*), 266  
getUserDropAction method (*TransferSupport*), 919  
getUserInfo method (*URI*), 258  
getUsername method (*RowSet*), 331  
getValidCharacters method (*MaskFormatter*), 703  
getValue method  
    of *AbstractSpinnerModel*, 705, 712  
    of *Attributes*, 205  
    of *Copies*, 882  
    of *Entry*, 625  
    of *JFormattedTextField*, 687, 701  
    of *JProgressBar*, 730  
    of *JSpinner*, 704, 710  
    of *Win32RegKey*, 987–988  
getValueAt method (*TableModel*), 605–606, 608  
getValueContainsLiteralCharacters method  
    (*MaskFormatter*), 703  
getValueCount method (*Entry*), 625  
getValueIsAdjusting method (*ListSelectionEvent*),  
    584  
getVendorName, getVersion methods  
    (*IIOServiceProvider*), 825, 833  
getVisibleRowCount method (*JList*), 587  
getWarnings method (*Connection*, *ResultSet*, *Statement*),  
    304  
getWidth method  
    of *ImageReader*, 826, 833  
    of *PageFormat*, 854, 861  
getWriter method (*ScriptContext*), 434  
getWriterXxx methods (*ImageIO*), 825, 832  
getYear method (*Date and Time API*), 356, 363  
GIF format, 823  
    animated, 825, 921  
    image manipulations on, 846  
    printing, 874  
GlassFish server, 349  
Glob patterns, 112  
GMail, 278–279  
Gnome (GNU Object Model Environment)  
    drag and drop in, 904  
    supporting API operations in, 927  
Gnu C compiler, 943–944  
Gödel's theorem, 504  
GradientPaint class, 797  
    constructor, 797–798  
    cyclic parameter, 798  
grant keyword, 516, 532, 564  
*Graphic Java™* (Geary), 600, 639  
Graphics class, 765  
    drawXxx, fillXxx methods, 769  
    get/setClip methods, 805–806, 854  
Graphics2D class, 765–937  
    clip method, 767, 805–807, 854  
    draw method, 767–769, 787  
    fill method, 767–768, 787  
    getFontRenderContext method, 805, 807  
    rotate, scale, shear methods, 800, 804  
    setComposite method, 767, 809, 816  
    setPaint method, 767, 797–798  
    setRenderingHint, setRenderingHints methods, 766,  
        817–819, 823  
    setStroke method, 767, 788, 796  
    setTransform method, 802, 804  
        transform method, 767, 802, 804  
        translate method, 800, 804, 864  
GregorianCalendar class, toZonedDateTime method,  
    369  
Grid bag layout, 175–179  
GridBagConstraints class, 175  
GridBagLayout class, 175–189  
GridBagPane class, 179  
Groovy programming language, 430–431,  
    438–439  
group method (*Matcher*), 135, 140–141  
groupCount method (*Matcher*), 141  
groupingBy method (*Collectors*), 28–31  
groupingByConcurrent method (*Collectors*), 43  
GSS-API, 579  
&gt;; entity reference, 148  
GUI (Graphical User Interface), scripting  
    events for, 437–442

**H**

\H in masks, 692  
 \h, \H, in regular expressions, 131  
 Half-closing connections, 249  
*Handbook of Applied Cryptography, The*  
 (Menezes et al.), 551  
 handle method (*CallbackHandler*), 545  
 handleGetObject method (*ResourceBundle*), 412–413  
 Handles (Swing), 644, 661  
*hash/Digest.java*, 549  
 hashCode method  
   of *Annotation*, 464  
   of *Permission*, 523  
*HashXxxAttributeSet* classes, 852, 880  
 Haskell programming language, 430  
 hasMoreElements method (*Enumeration*), 987–989  
 hasNext method (*XMLStreamReader*), 208  
 hasRemaining method (*Buffer*), 122  
 Header information, from server, 259  
 Headers (Swing tables)  
   rendering, 627  
   scrolling, 601  
*HelloNative/HelloNative.c*, 943  
*HelloNative/HelloNative.h*, 942  
*HelloNative/HelloNative.java*, 941  
*HelloNative/HelloNativeTest.java*, 945  
 Hex editors  
   creating class files in, 505  
   modifying bytecodes with, 507  
 Hidden commands, in XML comments, 149  
 HORIZONTAL\_SPLIT value (*JSplitPane*), 732  
 HORIZONTAL\_WRAP value (*JList*), 583  
 Hosts  
   and IP addresses, 239–241  
   local, 240  
 Hot deployment, 496  
 HTML (HyperText Markup Language)  
   attributes in, 148  
   displaying with *JEditorPane*, 681, 712–719  
   end and empty tags in, 146  
   forms in, 268  
   generating from XML files, 222–225  
   mixing with JSP, 449  
   printing, 874  
   vs. XML, 145–146  
 HTMLDocument class, 681  
 HTTP (Hypertext Transfer Protocol), 285  
   request headers in, 261–262  
   using SSL over, 579  
 http: URI scheme, 257, 517, 713

https: URI scheme, 257, 579  
 HttpURLConnection class, *getErrorStream* method, 273, 277  
 HyperlinkEvent class, *getURL* method, 715, 719  
 HyperlinkListener interface, *hyperlinkUpdate* method, 715, 718  
 Hyperlinks, 713  
**I**  
 I (int), type code, 87, 961  
 I/O streams. *See Input streams, Output streams*  
 IANA (Internet Assigned Numbers Authority), 361  
 IBM, 143, 150  
 IBM DB2 database, 291  
 IBM437 encoding, 406  
 ICC profiles, 836  
 Icons  
   for frames, 743  
   in column headers, 627  
   in table cells, 626  
   in trees, 648, 661  
   one-touch expand, 732  
   tray, 932  
 ID, IDREF, IDREFS attribute types (DTDs), 167–168  
 identity method (*Function*), 24  
 Identity (do-nothing) transformation, 213  
 Identity values, 34  
 IDs, uniqueness of, 168, 178  
 IETF BCP 47, 377  
 ifPresent method (*Optional*), 14–15, 41  
 IIIOImage class, 827, 834  
 IIIServiceProvider class, *getVendorName*, *getVersion* methods, 825, 833  
 IllegalAccessException, 674  
 IllegalArgumentException, 207, 706, 710, 712, 976  
 IllegalStateException, 24, 826, 914  
 Imageable area, 855  
 imageFlavor method (*DataFlavor*), 895  
 ImageInputStream class, “seek forward only” of, 825  
 ImageIO class  
   *createImageXxxStream* methods, 825, 832  
   determining image type, 823–824  
   *getImageXxxByXxx* methods, 824, 832  
   *getReaderXxx*, *getWriterXxx* methods, 825, 832  
   read, write methods, 823, 831  
 ImageIO/ImageIOFrame.java, 828  
 ImageOutputStream interface, 827

- imageProcessing/ImageProcessingFrame.java, 846  
ImageReader class, 824
  - getHeight, getWidth methods, 826, 833
  - getNumXxx methods, 826, 833
  - getOriginatingProvider method, 824, 833
  - read method, 832
  - readThumbnail method, 833
  - setInput method, 832  
ImageReaderWriterSpi class, getXxx methods, 833  
Images
  - blurring, 844
  - buffered, 798
    - types of, 835
  - color-model-specific values of, 837
  - constructing from pixels, 835–842
  - converting between Java and native formats, 895
  - edge detection of, 845
  - filtering, 842–851
  - getting size of, before reading, 826
  - incremental rendering of, 834
  - manipulating, 834–851
  - metadata in, 827
  - multiple, in a file, 825–834
  - printing, 852–862, 874, 876
  - readers/writers for, 823–834
  - rotating, 818, 843
  - scaling, 818, 820
  - superimposing, 807–808
  - thumbnails for, 826
  - transferring via clipboard, 894  
imageTransfer/ImageTransferFrame.java, 897  
ImageTransferable class, 895, 909  
ImageWriter class, 824, 827
  - canInsertImage method, 827, 834
  - getOriginatingProvider method, 834
  - setOutput method, 834
  - write, writeInsert methods, 827, 834  
IMAP (Internet Message Access Protocol), 579
  - implements specification, type use annotations in, 468  
#IMPLIED attribute (DTD), 167  
implies method
  - of Permission, 523–524, 530
  - of ProtectionDomain, 514  
import statement, 496  
importData method (TransferHandler), 912–914, 918  
include method (RowFilter), 616, 624  
INCLUDE environment variable, 984  
  
Incremental rendering, 834  
Indexed color model, 844  
indexOfComponent, indexOfTab methods (JTabbedPane), 737, 740  
indexOfTabComponent method (JTabbedPane), 741  
IndexOutOfBoundsException, 826  
InetAddress class, 239–241
  - getXxx methods, 240–241  
inetAddress/InetAddressTest.java, 240  
InetSocketAddress class, isUnresolved method, 256  
Infinite trees, 675  
Inheritance trees, 303  
@Inherited annotation, 471, 474  
init method
  - of Cipher, 573
  - of KeyGenerator, 574  
Initialization blocks, for shared libraries, 947  
initialize method
  - of LoginModule, 546
  - of SimpleLoginModule, 538  
INPUT\_STREAM class (DocFlavor), 875  
Input streams, 48–59
  - and Unicode, 48
  - as input source, 150
  - buffered, 57–59
  - byte processing in, 57
  - byte-oriented, 48
  - chaining, 57
  - closing, 49
  - encoding for, 60
  - filters for, 55–59
  - hierarchy of, 51–55
  - keeping open, 249
  - monitoring progress of, 726–731
  - objects in, 80–100
  - total number of bytes in, 727  
Input validation mask, 685  
InputSource class, 171  
InputStream class, 48–51, 52
  - available method, 49–50, 727
  - close method, 49–50
  - mark, markSupported methods, 50
  - markSupported method, 50
  - read method, 48–50
  - reset method, 50
  - skip method, 50  
InputStreamReader class, 60, 726  
InputVerifier class, verify method, 690  
INSERT statement (SQL), 290
  - and autogenerated keys, 321

- INSERT statement (SQL) (*continued*)  
executing, 298, 300, 316  
in batch updates, 345  
vs. methods of *ResultSet*, 325  
*insertNodeInto* method (*DefaultTreeModel*), 652, 658  
*insertRow* method (*ResultSet*), 325, 328  
*insertString* method (*DocumentFilter*), 688–689,  
    702  
*insertTab* method (*JTabbedPane*), 736, 740  
*insertUpdate* method (*DocumentListener*), 682, 685  
*Inside Java™ 2 Platform Security* (Gong et al.),  
    492  
*installUI* method (*LayerUI*), 764  
Instance fields  
    accessing from native code, 956–960  
    annotating, 466  
*instanceof* keyword, and type use annotations,  
    468  
Instant class, 352  
    and legacy classes, 369  
    arithmetic operations, 353  
    from method, 369  
    immutable, 354  
    now method, 352  
Instrumentation API, 487  
*int* type  
    printing, 61  
    storing, 70  
    type code for, 87, 961  
    vs. C types, 947  
    writing in binary format, 69  
*IntBuffer* class, 124  
INTEGER data type (SQL), 291, 348  
Integers  
    ranges of, 36  
    supported locales for, 686  
    validating input of, 686  
*IntegerSyntax* class, 882  
@interface declaration, 456, 462  
Interfaces  
    accessing script classes with, 436  
    annotating, 466, 470  
    implementing in script engines, 435  
Internal frames (Swing), 741–752  
    cascading/tiling, 744–748  
    closing, 748  
    dialogs in, 750  
    dragging, 751–757  
    grabbers of, 742  
    icons for, 743  
managing on desktop, 751  
positioning, 744  
resizable, 745  
selected, 744  
states of, 745  
visibility of, 744  
*internalFrame/DesktopFrame.java*, 752  
*InternalFrameListener* interface, *internalFrameClosing*  
    method, 751  
InternationalFormatter class, 689  
Internationalization, 371–428  
Interpolation, 843  
    for gradients, 797–798  
    strategies of, 843  
    when transforming images, 818, 820, 843  
Interruptible sockets, 250–257  
*interruptible/InterruptibleSocketTest.java*, 251  
*intersect* method (*Area*), 787–788  
*ints* method (*Random*), 37, 40  
*InputStream* interface, 36–41  
    average, max, min, sum methods, 37, 39  
    boxed method, 36, 39  
    mapToInt method, 36  
    of, range, rangeClosed methods, 36, 38  
    summaryStatistics method, 37, 39  
    toArray method, 36, 38  
*IntSummaryStatistics* class, 20, 23, 37, 41  
*intValue* method (*Number*), 379  
Invalid pointers (C, C++), 940  
InvalidPathException, 101  
*Invocable* interface, 435  
    getInterface method, 436  
    invokeXXX methods, 435–436  
Invocation API, 980–985  
*invocation/InvocationTest.c*, 981  
IOException, 237, 716  
IP addresses, 235, 239–241  
    validating, 693–694  
IPP (Internet Printing Protocol) 1.1, 885  
IPv6 addresses, 239  
isAfter, isBefore methods (Date and Time API),  
    356, 360, 363  
isAfterLast method (*ResultSet*), 324, 327  
isAnnotationPresent method (*AnnotatedElement*), 462  
IsAssignableFrom function (C), 989, 1001  
isBeforeFirst method (*ResultSet*), 324, 327  
isCanceled method (*ProgressMonitor*), 723, 731  
isCellEditable method  
    of *AbstractCellEditor*, 630  
    of *AbstractTableModel*, 628

- of *CellEditor*, 638
  - of *DefaultTableModel*, 628
  - of *TableModel*, 608, 628
  - isCharacters* method (*XMLStreamReader*), 208
  - isClosable* method (*JInternalFrame*), 758
  - isClosed* method
    - of *JInternalFrame*, 758
    - of *ResultSet*, 301
    - of *Socket*, 239
    - of *Statement*, 300
  - isConnected* method (*Socket*), 239
  - isContinuousLayout* method (*JSplitPane*), 735
  - isDataFlavorAvailable* method (*Clipboard*), 891
  - isDataFlavorSupported* method (*Transferable*), 892
  - isDesktopSupported* method (*Desktop*), 927, 931
  - isDirectory* method
    - of *BasicFileAttributes*, 109
    - of *ZipEntry*, 79
  - isDrop* method (*TransferSupport*), 914, 919
  - isEchoOn* method (*PasswordCallback*), 546
  - isEditValid* method (*JFormattedTextField*), 687, 690, 701
  - isEndElement* method (*XMLStreamReader*), 208
  - isExecutable* method (*Files*), 108–109
  - isFirst* method (*ResultSet*), 324, 327
  - isGroupingUsed* method (*NumberFormat*), 383
  - isHidden* method (*Files*), 108–109
  - isIcon* method (*JInternalFrame*), 745, 758
  - isIconifiable* method (*JInternalFrame*), 758
  - isIgnoringElementContentWhitespace* method (*DocumentBuilderFactory*), 171
  - isImageAutoSize* method (*TrayIcon*), 937
  - isIndeterminate* method (*JRootPane*), 730
  - isInputShutdown* method (*Socket*), 249
  - isInsert* method (*JList.DropLocation*), 915, 920
  - isInsertXxx* methods (*JTable.DropLocation*), 920
  - isLast* method (*ResultSet*), 324, 327
  - isLeaf* method
    - of *DefaultTreeModel*, 648
    - of *TreeModel*, 649, 672, 680
    - of *TreeNode*, 648–649
  - isLeap* method (Date and Time API), 356
  - isMaximizable* method (*JInternalFrame*), 758
  - isMaximum* method (*JInternalFrame*), 758
  - isMimeTypeEqual* method (*DataFlavor*), 894
  - isNamespaceAware* method
    - of *DocumentBuilderFactory*, 199
    - of *SAXParserFactory*, 203
  - isNegative* method (*Instant, Duration*), 353
  - ISO 216 paper sizes, 412
  - ISO 3166–1 country codes, 374
  - ISO 4217 currency identifiers, 384
  - ISO 639–1 language codes, 374
  - ISO 8601 format, 318, 471
  - ISO 8859–1 standard, 60, 68
  - isOneTouchExpandable* method (*JSplitPane*), 734
  - isOutputShutdown* method (*Socket*), 249
  - isParseIntegerOnly* method (*NumberFormat*), 383
  - isPresent* method (*Optional*), 15–16
  - iSQL-Viewer program, 334
  - isReadable* method (*Files*), 108–109
  - isRegularFile* method
    - of *BasicFileAttributes*, 109
    - of *Files*, 108–109
  - isResizable* method (*JInternalFrame*), 758
  - isSelected* method (*JInternalFrame*), 747, 758
  - isShared* method (*FileLock*), 127
  - isStartElement* method (*XMLStreamReader*), 208
  - isStringPainted* method (*JProgressBar*), 730
  - isSupported* method
    - of *Desktop*, 927, 931
    - of *SystemTray*, 932, 936
  - isSymbolicLink* method
    - of *BasicFileAttributes*, 109
    - of *Files*, 108–109
  - isUnresolved* method (*InetSocketAddress*), 256
  - isValidating* method
    - of *DocumentBuilderFactory*, 171
    - of *SAXParserFactory*, 203
  - isVisible* method (*JInternalFrame*), 759
  - isWhiteSpace* method (*XMLStreamReader*), 208
  - isWritable* method (*Files*), 108–109
  - isZero* method (*Instant, Duration*), 353
  - item* method
    - of *NamedNodeMap*, 162
    - of *NodeList*, 151, 162, 169
  - Iterable* interface, 111
  - iterate* method (*Stream*), 5, 8, 12, 36
  - Iterator* interface, 298
  - iterator* method
    - of *BaseStream*, 19–20
    - of *SQLException*, 302, 304
  - Iterators, 19
- J**
- J* (*long*), type code, 87, 961
  - JAAS (Java Authentication and Authorization Service), 530–546
    - configuration files in, 531, 536
    - login modules in, 537–546

- jaas/jaas.config, 545  
jaas/JAATest.java, 544  
jaas/JAATest.policy, 545  
jaas/SimpleCallbackHandler.java, 543  
jaas/SimpleLoginModule.java, 541  
jaas/SimplePrincipal.java, 540  
jaas.config file, 533  
**JAR files**  
    automatic registration in, 294  
    class loaders in, 495  
    code base of, 510  
    for plugins, 494  
    manifest of, 77, 921  
    resources in, 408  
    signing, 555–556  
jar: URI scheme, 257  
jarray type (C), 988  
jarsigner program, 555–556, 563  
JarXxxStream classes, 77  
Java programming language  
    deployment directory of, 564, 566  
    internationalization support in, 371  
    platform-independent, 70  
    security of, 510–514  
    vs. SQL, 311  
Java 2D API, 765–937  
    features supported in, 766  
    rendering pipeline, 766–768  
Java EE (Java Platform, Enterprise Edition),  
    285  
Java Plug-in, loading signed code, 563  
java program  
    -javaagent option, 487  
    jdbc.drivers property in, 294  
    -noverify option, 506  
    security managers in, 516  
    -splash option, 921  
Java Rich Internet Applications Guide, 566  
*Java™ Virtual Machine Specification, The*  
    (Lindholm/Yellin), 506  
java.awt.AlphaComposite API, 817  
java.awt.BasicStroke API, 796  
java.awt.Color API, 842  
java.awt.datatransfer API, 888  
java.awt.datatransfer.Clipboard API, 891–892, 894,  
    903  
java.awt.datatransfer.ClipboardOwner API, 892  
java.awt.datatransfer.DataFlavor API, 893–894  
java.awt.datatransfer.FlavorListener API, 894  
java.awt.datatransfer.Transferable API, 892, 894  
java.awt/Desktop API, 931–932  
java.awt.dnd API, 904  
java.awt.font.TextLayout API, 807  
java.awt.geom package, 92  
java.awt.geom.AffineTransform API, 803–804  
java.awt.geom.Arc2D.Double API, 785  
java.awt.geom.Area API, 788  
java.awt.geom.CubicCurve2D.Double API, 785  
java.awt.geom.GeneralPath API, 785  
java.awt.geom.Path2D API, 786  
java.awt.geom.Path2D.Float API, 786  
java.awt.geom.QuadCurve2D.Double API, 785  
java.awt.geom.RoundRectangle2D.Double API, 785  
java.awt.GradientPaint API, 798  
java.awt.Graphics API, 806  
java.awt.Graphics2D API, 768, 796, 798, 804, 807,  
    816, 823  
java.awt.imageAffineTransformOp API, 850  
java.awt.image.BufferedImage API, 840  
java.awt.image.BufferedImageOp API, 850  
java.awt.image.ByteLookupTable API, 850  
java.awt.image.ColorModel API, 842  
java.awt.image.ConvolveOp API, 851  
java.awt.image.Kernel API, 851  
java.awt.image.LookupOp API, 850  
java.awt.image.Raster API, 841  
java.awt.image.RescaleOp API, 850  
java.awt.image.ShortLookupTable API, 851  
java.awt.image.WritableRaster API, 841  
java.awt.print.Book API, 873  
java.awt.print.PageFormat API, 861–862  
java.awt.print.Printable API, 860  
java.awt.print.PrinterJob API, 861, 873  
java.awt.RenderingHints API, 823  
java.awt.SplashScreen API, 926  
java.awt.SystemTray API, 936  
java.awt.TexturePaint API, 799  
java.awt.Toolkit API, 891  
java.awt.TrayIcon API, 936–937  
java.beans.PropertyChangeEvent API, 759  
java.beans.PropertyVetoException API, 760  
java.beans.VetoableChangeListener API, 759  
java.io.BufferedReader API, 59  
java.io.BufferedOutputStream API, 59  
java.io.Closeable API, 54  
java.io.DataInput API, 71  
java.io.DataOutput API, 72  
java.io.File API, 103  
java.io.File.separator constant, 56  
java.io.FileInputStream API, 58, 121

- java.io.FileOutputStream API, 59, 121  
java.io.Flushable API, 55  
java.io.InputStream API, 49–50  
java.io.ObjectInputStream API, 85  
java.io.ObjectOutputStream API, 85  
java.io.OutputStream API, 50–51  
java.io.PrintWriter API, 62  
java.io.PushbackInputStream API, 59  
java.io.RandomAccessFile API, 76, 121  
java.lang, java.lang.annotation packages,  
  annotations in, 470  
java.lang.annotation.Annotation API, 463–464  
java.lang.Appendable API, 55  
java.lang.CharSequence API, 40, 55  
java.lang.Class API, 503, 514  
java.lang.ClassLoader API, 503  
java.lang.Readable API, 55  
java.lang.reflect.AnnotatedElement API, 462  
java.lang.SecurityManager API, 513  
java.lang.System API, 946  
java.lang.Thread API, 503  
java.net package  
  socket connections in, 238  
  supporting IPv6 addresses in, 240  
  URLs vs. URIs in, 257  
java.net.HttpURLConnection API, 277  
java.net.InetAddress API, 241  
java.net.InetSocketAddress API, 256  
java.net.ServerSocket API, 245  
java.net.Socket API, 238–239, 249  
java.net.URL API, 265  
java.net.URLClassLoader API, 503  
java.net.URLConnection API, 265–267  
java.net.URLDecoder API, 277  
java.net.URLEncoder API, 277  
java.nio package, 247, 250  
  direct buffers in, 973  
  memory mapping in, 117  
java.nio.Buffer API, 122, 126  
java.nio.ByteBuffer API, 122–123  
java.nio.channels.Channels API, 256–257  
java.nio.channels.FileChannel API, 122, 128  
java.nio.channels.FileLock API, 128  
java.nio.channels.SocketChannel API, 256  
java.nio.CharBuffer API, 124  
java.nio.file.attribute.BasicFileAttributes API, 109  
java.nio.file.Files API, 8, 104–106, 108–109, 114  
java.nio.file.FileSystem API, 116  
java.nio.file.FileSystems API, 116  
java.nio.file.Path API, 103  
java.nio.file.Paths API, 103  
java.nio.file.SimpleFileVisitor API, 115  
java.policy file, 515, 565  
.java.policy file, 515–516  
java.security package, 492, 547  
java.security file, 515  
java.security.CodeSource API, 514  
java.security.MessageDigest API, 550  
java.security.Permission API, 530  
java.security.Principal API, 537  
java.security.PrivilegedAction API, 536  
java.security.PrivilegedExceptionAction API, 537  
java.security.ProtectionDomain API, 514  
java.sql.Blob API, 317  
java.sql.Clob API, 317–318  
java.sql.Connection API, 299, 304, 315, 318, 326,  
  342, 346  
java.sql.DatabaseMetaData API, 328, 342–343, 347  
java.sql.DataTruncation API, 304–305  
java.sql.DriverManager API, 297  
java.sql.PreparedStatement API, 316  
java.sql.ResultSet API, 300–301, 304, 317,  
  326–328, 343  
java.sql.ResultSetMetaData API, 343  
java.sql.Savepoint API, 347  
java.sql.SQLXxx APIs, 304  
java.sql.Statement API, 300, 304, 320–321, 347  
java.text.CollationKey API, 400  
java.text.Collator API, 399  
java.text.Format API, 402  
java.text.MessageFormat API, 401–402  
java.text.Normalizer API, 400  
java.text.NumberFormat API, 383  
java.text.SimpleDateFormat API, 712  
java.time.format.DateTimeFormatter API, 392  
java.time.LocalXxx APIs, 393  
java.time.ZonedDateTime API, 393  
java.util.Arrays API, 8  
java.util.Collection API, 5, 46  
java.util.Currency API, 385  
java.util.function.Supplier API, 8  
java.util.Locale API, 377  
java.util.Optional API, 15–16, 19  
java.util.OptionalXxx APIs, 41  
java.util.Random API, 40  
java.util.regex.Matcher API, 140–141  
java.util.regex.Pattern API, 8, 139–140  
java.util.ResourceBundle API, 412–413  
java.util.Stream API, 35  
java.util.stream.BaseStream API, 20, 45

- java.util.stream.Collectors* API, 23, 27, 29, 33  
*java.util.stream.DoubleStream* API, 39–40  
*java.util.stream.IntStream* API, 38–39  
*java.util.stream.LongStream* API, 39  
*java.util.stream.Stream* API, 4, 10–13, 22  
*java.util.Xxx.SummaryStatistics* APIs, 41  
*java.util.zip.ZipEntry* API, 79  
*java.util.zip.ZipFile* API, 80  
*java.util.zip.ZipInputStream* API, 78  
*java.util.zip.ZipOutputStream* API, 78–79  
*javac* program  
    -encoding option, 407  
    -XprintRounds option, 480  
*JavaCompiler* interface  
    getStandardFileManager method, 447  
    getTask method, 444, 448  
*Javadoc*, 472  
*javaFileListFlavor* method (*DataFlavor*), 914  
*JavaFileManager* interface, 444–445  
*JavaFileObject* interface, 444–445  
*javah* program, 942, 990  
*JavaHelp*, 713  
*JavaMail*, 278  
*JavaOne* conference, 284  
*javap* program, 962  
*JavaScript*, 430, 438  
    javax.annotation package, annotations in, 470  
    javax.crypto.Cipher API, 573  
    javax.crypto.CipherXxxStream APIs, 575  
    javax.crypto.KeyGenerator API, 574  
    javax.crypto.spec.SecretKeySpec API, 574  
    javax.imageio package, 823  
    javax.imageio.IIIOImage API, 834  
    javax.imageio.ImageIO API, 831–832  
    javax.imageio.ImageReader API, 832–833  
    javax.imageio.ImageWriter API, 834  
    javax.imageio.spi.IIIServiceProvider API, 833  
    javax.imageio.spi.ImageReaderWriterSpi API, 833  
    javax.print.attribute.Attribute API, 886  
    javax.print.attribute.AttributeSet API, 886  
    javax.print.DocPrintJob API, 877, 886  
    javax.print.PrintService API, 877, 886  
    javax.print.PrintServiceLookup API, 877  
    javax.print.SimpleDoc API, 878  
    javax.print.StreamPrintServiceFactory API, 879  
    javax.script.Bindings API, 434  
    javax.script.Compilable API, 437  
    javax.script.CompiledScript API, 437  
    javax.script.Invocable API, 436  
    javax.script.ScriptContext API, 434  
    javax.script.ScriptEngine API, 433–434  
    javax.script.ScriptEngineFactory API, 431  
    javax.script.ScriptEngineManager API, 431, 433  
    javax.security.auth.callback.CallbackHandler API, 545  
    javax.security.auth.callback.NameCallback API, 545  
    javax.security.auth.callback.PasswordCallback API, 546  
    javax.security.auth.login.LoginContext API, 536  
    javax.security.auth.spi.LoginModule API, 546  
    javax.security.auth.Subject API, 536  
    javax.sql package, 349  
    javax.sql.RowSet API, 331–332  
    javax.sql.rowset API, 329  
    javax.sql.rowset.CachedRowSet API, 332  
    javax.sql.rowset.RowSetFactory API, 333  
    javax.sql.rowset.RowSetProvider API, 332  
    javax.swing.AbstractSpinnerModel API, 712  
    javax.swing.CellEditor API, 638  
    javax.swing.DefaultCellEditor API, 638  
    javax.swing.DefaultListModel API, 595  
    javax.swing.DefaultRowSorter API, 624  
    javax.swing.event.DocumentEvent API, 685  
    javax.swing.event.DocumentListener API, 685  
    javax.swing.event.HyperlinkEvent API, 719  
    javax.swing.event.HyperlinkListener API, 718  
    javax.swing.event.ListSelectionListener API, 588  
    javax.swing.event.TreeModelEvent API, 680  
    javax.swing.event.TreeModelListener API, 680  
    javax.swing.event.TreeSelectionEvent API, 671  
    javax.swing.event.TreeSelectionListener API, 671  
    javax.swing.JColorChooser API, 909  
    javax.swing.JComponent API, 650, 685, 759, 909  
    javax.swing.JDesktopPane API, 757  
    javax.swing.JEditorPane API, 718  
    javax.swing.JFileChooser API, 909  
    javax.swing.JFormattedTextField API, 701  
    javax.swing.JFormattedTextField.AbstractFormatter API, 701  
    javax.swing.JFrame API, 919  
    javax.swing.JInternalFrame API, 757–759  
    javax.swing.JLayer API, 763–764  
    javax.swing.JList API, 587, 593, 595, 599, 909, 919  
    javax.swing.JList.DropLocation API, 920  
    javax.swing.JProgressBar API, 730  
    javax.swing.JSpinner API, 710  
    javax.swing.JSpinner.DateEditor API, 712  
    javax.swing.JPanel API, 734–735  
    javax.swing.JTabbedPane API, 740–741  
    javax.swing.JTable API, 604, 622–623, 637, 909, 919

javax.swing.JTable.DropLocation API, 920  
javax.swing.JTree API, 649, 657, 671, 909, 919  
javax.swing.JTree.DropLocation API, 920  
javax.swing.ListCellRenderer API, 599  
javax.swingListModel API, 593  
javax.swingListSelectionModel API, 624  
javax.swing.plaf.LayerUI API, 764  
javax.swing.ProgressMonitor API, 731  
javax.swing.ProgressMonitorInputStream API, 731  
javax.swing.RowFilter API, 624–625  
javax.swing.RowFilter.Entry API, 625  
javax.swing.SpinnerXxxModel APIs, 711  
javax.swing.table.TableCellEditor API, 638  
javax.swing.table.TableCellRenderer API, 637  
javax.swing.table.TableColumn API, 623, 637  
javax.swing.table.TableColumnModel API, 623  
javax.swing.table.TableModel API, 608, 621  
javax.swing.table.TableRowSorter API, 624  
javax.swing.table.TableStringConverter API, 624  
javax.swing.text.DefaultFormatter API, 702  
javax.swing.text.Document API, 685  
javax.swing.text.DocumentFilter API, 702  
javax.swing.text.JTextComponent API, 909, 919  
javax.swing.text.JTextComponent.DropLocation API, 921  
javax.swing.text.MaskFormatter API, 703  
javax.swing.TransferHandler API, 909, 912, 918  
javax.swing.TransferHandler.DropLocation API, 920  
javax.swing.TransferHandler.TransferSupport API, 919  
javax.swing.tree.DefaultMutableTreeNode API, 650, 663  
javax.swing.tree.DefaultTreeCellRenderer API, 664  
javax.swing.tree.DefaultTreeModel API, 650, 658  
javax.swing.tree.MutableTreeNode API, 649  
javax.swing.tree.TreeCellRenderer API, 663  
javax.swing.tree.TreeModel API, 649, 679–680  
javax.swing.tree.TreeNode API, 649, 658  
javax.swing.tree.TreePath API, 658  
javax.tools.Diagnostic API, 448–449  
javax.tools.DiagnosticCollector API, 448  
javax.tools.ForwardingJavaFileManager API, 449  
javax.tools.JavaCompiler API, 447–448  
javax.tools.JavaCompiler.CompilationTask API, 448  
javax.tools.SimpleJavaFileObject API, 449  
javax.tools.StandardJavaFileManager API, 448  
javax.tools.Tool API, 447  
javax.xml.parsers.DocumentBuilder API, 160, 170, 212  
javax.xml.parsers.DocumentBuilderFactory API, 160,  
    171, 199  
javax.xml.parsers.SAXParser API, 203  
javax.xml.parsers.SAXParserFactory API, 203  
javax.xml.stream.XMLInputFactory API, 207  
javax.xml.stream.XMLOutputFactory API, 221  
javax.xml.stream.XMLStreamReader API, 208  
javax.xml.stream.XMLStreamWriter API, 221–222  
javax.xml.transform.dom.DOMResult API, 232  
javax.xml.transform.dom.DOMSource API, 213  
javax.xml.transform.sax.SAXSource API, 231  
javax.xml.transform.stream.StreamResult API, 213  
javax.xml.transform.stream.StreamSource API, 231  
javax.xml.transform.Transformer API, 213  
javax.xml.transform.TransformerFactory API, 213, 231  
javax.xml.xpath.XPath API, 196  
javax.xml.xpath.XPathFactory API, 195  
JAXP (Java API for XML Processing), 150  
jboolean type (C), 947  
jbooleanArray type (C), 970  
jbyte type (C), 947  
jbyteArray type (C), 970  
jchar type (C), 947, 949  
jcharArray type (C), 970  
JCheckBox class, 629  
jclass type (C), 965  
JColorChooser class  
    drag-and-drop behavior of, 905  
    setDragEnabled method, 909  
JComboBox class, 629  
JComponent class  
    addVetoableChangeListener method, 759  
    attaching verifiers to, 690  
    extending, 596  
    getPreferredSize method, 596–597, 685  
    paint method, 626, 766  
    paintComponent method, 596–597, 766  
    putClientProperty method, 645, 650, 751  
    setPreferredSize method, 685  
    setTransferHandler method, 906, 909  
JDBC API, 281–350  
    configuration of, 291–297  
    debugging, 296  
    design of, 282–285  
    tracing, 296  
    uses of, 284–285  
    versions of, 281  
JDBC API Tutorial and Reference (Fisher et al.), 326, 349  
JDBC drivers  
    escape syntax in, 318–319  
    JAR files for, 292  
    registering classes for, 294  
    scrollable/updatable result sets in, 323  
    types of, 283–284

- JDBC/ODBC bridge, not available in Java 8, 283  
*JdbcRowSet* interface, 329  
*JDesktopPane* class, 742  
    *getAllFrames* method, 745–748, 757  
    no built-in cascading/tiling in, 744–748  
    *setDragMode* method, 751, 757  
*JDialog* class, 750  
*JDK* (Java Development Kit)  
    Apache Derby database, 291  
    DOM parser, 150  
    keytool program, 553  
    native2ascii program, 408  
    policytool program, 522  
    serialver program, 96  
    src.jar file, 981  
    StylePad demo, 682  
    SunJCE ciphers, 568  
    `jdouble` type (C), 947  
    `jdoubleArray` type (C), 970  
*JEditorPane* class, 681, 712–719  
    *addHyperlinkListener* method, 718  
    drag-and-drop behavior of, 905  
    *setEditable* method, 714  
    *setPage* method, 714, 718  
*JFileChooser* class  
    drag-and-drop behavior of, 905  
    *setDragEnabled* method, 909  
    `jfloat` type (C), 947  
    `jfloatArray` type (C), 970  
*JFormattedTextField* class, 685–703  
    *commitEdit* method, 688, 701  
    constructor, 701  
    drag-and-drop behavior of, 905  
    get/setFocusLostBehavior methods, 688, 701  
    get/setValue methods, 687–688, 701  
    *isEditValid* method, 687, 690, 701  
    *setText* method, 688  
    supported formatters, 691–693  
*JFrame* class, 742  
    adding transfer handler to, 912  
    *setTransferHandler* method, 919  
    `jint` type (C), 947  
    `jintArray` type (C), 970  
*JInternalFrame* class, 748–750  
    constructor, 742, 757  
    get/setContentPane methods, 758  
    get/setFrameIcon methods, 743, 759  
    *getDesktopPane* method, 759  
    *is setClosable* methods, 758  
    *is setClosed* methods, 748, 758  
    *is setIcon* methods, 745, 758  
    *is setIconifiable* methods, 758  
    *is setMaximizable* methods, 758  
    *is setMaximum* methods, 745, 758  
    *is setResizable* methods, 758  
    *is setSelected* methods, 744, 747, 758  
    *is setVisible* methods, 744, 759  
    *moveToXXX* methods, 758  
    no built-in cascading/tiling in, 744–748  
    *reshape* method, 743, 758  
    *show* method, 759  
*JLabel* class  
    extending, 661–662  
    using with *JList*, 597  
*JLayer* class, 760–764  
    constructor, 763  
    *setLayerEventMask* method, 764  
*JList* class, 582–588  
    *addListSelectionListener* method, 587  
    as generic type, 582  
    configuring for custom renderers, 597  
    constructor, 587, 593  
    drag-and-drop behavior of, 905, 913–914  
    *get/setLayoutOrientation* methods, 587  
    *get/setPrototypeCellValue* methods, 593  
    *get/setSelectionMode* methods, 583, 587  
    *get/setVisibleRowCount* methods, 583, 587  
    *getBackground*, *getForeground*, *getSelectionXxx* methods, 596, 599  
    *getModel* method, 595  
    *getSelectedValue* method, 584, 587  
    *getSelectedValuesList* method, 585, 587  
    *HORIZONTAL\_WRAP*, *VERTICAL*, *VERTICAL\_WRAP* values, 583  
    *setCellRenderer* method, 597, 599  
    *setDragEnabled* method, 909  
    *setDropMode* method, 913, 919  
    *setFixedCellXxx* methods, 593  
    visual appearance of data vs. data storage in, 588  
*JList.DropLocation* class  
    *getIndex* method, 914, 920  
    *isInsert* method, 915, 920  
    `jlong` type (C), 947  
    `jlongArray` type (C), 970  
    JNDI service, 349  
        class loaders in, 495  
    `JndiLoginModule` class, 532

- JNI (Java Native Interface), 940–1001  
array elements in, 970–974  
calling convention in, 950  
debugging mode of, 980  
error handling in, 974–979  
functions in C++, 950  
invoking Java methods in, 963–970  
online documentation for, 952  
`JNI_CreateJavaVM` function (C), 980–981, 985  
`JNI_OnLoad`, `JNI_OnUnload` methods (C), 947  
`jni.h` file, 947  
`JNICALL`, `JNIREPORT` macros, 942  
`JobAttributes` class (obsolete), 886  
`jobject` type (C), 965, 971, 988  
`jobjectArray` type (C), 970  
`Join` styles, 788–789  
joining method (`Collectors`), 20, 23  
`JoinRowSet` interface, 329  
`JOptionPane` class, `showInternalXxxDialog` methods, 750  
`JPanel` class, 857  
    adding layers to, 760  
`JPasswordField` class, 681  
    drag-and-drop behavior of, 905  
`JPEG` format, 823  
    image manipulations on, 846  
    printing, 874  
    reading, 824  
    transparency in, 921  
`JProgressBar` class, 719–722, 923  
    constructor, 730  
    `get/setMaximum`, `get/setMinimum` methods, 719, 730  
    `get/setString` methods, 720, 730  
    `getValue` method, 730  
    `is/setIndeterminate` methods, 720, 730  
    `is/setStringPainted` methods, 720, 730  
    `setValue` method, 719, 730  
`js.properties` file, 438  
`JScrollPane` class  
    with `JList`, 582, 590  
    with `JTabbedPane`, 736  
    with `JTable`, 601  
`JSF` (JavaServer Faces), 268  
`jshort` type (C), 947  
`jshortArray` type (C), 970  
`JSP` (JavaServer Pages), 449–454  
`JSpinner` class, 703–712  
    constructor, 710  
    `getValue` method, 704, 710  
    `setEditor`, `setValue` methods, 710  
`JSplitPane` class, 732–735  
    constructor, 734  
    `HORIZONTAL_SPLIT`, `VERTICAL_SPLIT` values, 732  
    `is/setContinuousLayout` methods, 733, 735  
    `is/setOneTouchExpandable` methods, 732, 734  
    `setXxxComponent` methods, 735  
`JSSE` (Java Secure Socket Extension), 579  
`jstring` type (C), 950–951, 965, 988  
`JTabbedPane` class, 735–741  
    `addChangeListener` method, 737, 741  
    `addTab` method, 736, 740  
    constructor, 736, 740  
    `get/setComponentAt` methods, 740  
    `get/setIconAt` methods, 740  
    `get/setMnemonicAt` methods, 737, 741  
    `get/setSelectedIndex` methods, 736–737, 740  
    `get/setTitleAt` methods, 740  
    `getSelectedComponent` method, 740  
    `getTabComponentAt`, `getTabLayoutPolicy` methods, 741  
    `getTabCount` method, 740  
    `indexOfComponent`, `indexOfTab` methods, 737, 740  
    `indexOfTabComponent` method, 741  
    `insertTab`, `removeTabAt` methods, 736, 740  
    `setTabXxx` methods, 737, 741  
`JTable` class, 599–638  
    `addColumn` method, 617, 622  
    asymmetric, 608  
    cell editors, automatically installed, 629  
    constructor, 604  
    `convertXxxIndexToModel` methods, 614, 623  
    default rendering actions, 609  
    drag-and-drop behavior of, 905, 913–914  
    `get/setAutoCreateRowSorter` methods, 602, 604, 614  
    `get/setCellSelectionEnabled` methods, 613, 622  
    `get/setColumnSelectionAllowed` methods, 613, 622  
    `get/setDefaultRenderer` methods, 626–627, 637  
    `get/setFillsViewportHeight` methods, 602, 604  
    `getColumnModel` method, 622  
    `getDefaultValue` method, 637  
    `getRowXxx` methods, 622  
    `getSelectionModel` method, 622  
    `moveColumn`, `removeColumn` methods, 617, 623  
    not generic, 601  
    `print` method, 602, 604  
    resize modes, 611  
    `setAutoResSizeMode` method, 611, 622  
    `setDragEnabled` method, 909  
    `setDropMode` method, 913, 919

- JTable** class (*continued*)  
     setRowHeight, setRowMargin, setRowSelectionAllowed  
         methods, 612, 622  
     setRowsSorter method, 614, 623
- JTextArea** class, 681  
     drag-and-drop behavior of, 905  
     extending, 524
- JTextComponent** class  
     drag-and-drop behavior of, 913–914  
     setDragEnabled method, 909  
     setDropMode method, 913, 919
- JTextField** class, 681  
     drag-and-drop behavior of, 905  
     with DefaultCellEditor, 629
- JTextPane** class, 681  
     drag-and-drop behavior of, 905
- JTree** class, 639–680  
     addTreeSelectionListener method, 664  
     constructor, 640, 649  
     drag-and-drop behavior of, 905, 913–914  
     getLastSelectedPathComponent method, 652, 657  
     getSelectionPath method, 651, 657, 665, 671  
     getSelectionPaths method, 665, 671  
     identifying nodes, 650  
     makeVisible method, 653, 657  
     scrollPathToVisible method, 653, 657  
     setDragEnabled method, 909  
     setDropMode method, 913, 919  
     setRootVisible method, 647, 649  
     setShowsRootHandles method, 646, 649
- JTree.DropLocation** class  
     getChildIndex method, 920  
     getPath method, 914, 920
- JUnit** 4 tool, 456
- Just-in-time compiler**, 980
- jvm** pointer (C), 980–981
- JVM** (Java virtual machine)  
     bootstrap class loader in, 493  
     class files in, 492  
     creating, 980  
     embedding into native code, 980–985  
     passing command-line arguments to, 515  
     terminating, 509–530, 981  
     transferring object references in, 902
- K**  
     \k, in regular expressions, 131
- Kerberos** protocol, 531, 579
- Kernel** class, 845, 851
- Kernel**, of a convolution, 844
- KEY\_Xxx** rendering hints, 817–818
- Key/value pairs.** *See* Properties
- Keyboard**, reading from, 48, 60
- KeyGenerator** class, 569  
     generateKey method, 569, 574  
     getInstance method, 574  
     init method, 574
- KeyPairGenerator** class, 576
- Keys**  
     autogenerated, 320–321  
     generating, 569–574  
     primary, 320
- keystore** keyword, 563
- KeyStoreLoginModule** class, 532
- Keystores**, 553–556, 563  
     referencing in policy files, 563
- Keystrokes**  
     filtering, 685  
     monitoring, 682
- keytool** program, 553–556
- Krb5LoginModule** class, 532
- L**
- L**  
     (object), type code, 87, 961  
     in masks, 692
- Lambda expressions**, using with `map` method, 9
- Landscape orientation**, 802
- Language codes**, 374
- Language Model API**, 476–477
- Language tags**, 377
- last** method (`ResultSet`), 323, 327
- lastInMonth** method (`TemporalAdjusters`), 359
- lastXxxTime** methods (`BasicFileAttributes`), 109
- layer**/`ColorFrame.java`, 761
- Layers** (Swing), 760–764
- LayerUI** class, 760–764  
     extending, 760  
     installUI, uninstallUI methods, 764  
     paint method, 764  
     processXxxEvent methods, 761, 764
- Layout algorithm**, 862–863
- layoutPages** method (`Banner`), 863
- LCD displays**, 836
- LD\_LIBRARY\_PATH** environment variable, 946, 985
- LDAP** (Lightweight Directory Access Protocol), 579
- Leaf icons**, 661

- Leap seconds, 352  
Leap years, 356  
*Learn SQL The Hard Way* (Shaw), 285  
*Learning SQL* (Beaulieu), 285  
Leaves (Swing), 639, 647, 672  
  icons for, 648, 661  
Legacy data, converting into XML, 227  
length method  
  of *Blob*, *Clob*, 317  
  of *CharSequence*, 55  
  of *RandomAccessFile*, 73, 76  
LIB environment variable, 984  
Libraries, replacing with newer versions, 494  
LIKE statement (SQL), 289, 319  
limit method (*Stream*), 10–11, 43, 122  
Line feed, 61, 405  
  in e-mails, 278  
  in regular expressions, 132  
Line2D class, 769  
lines method (*Files*), 6, 8  
Lines, reading, 6  
lineTo method (*Path2D.Float*), 775, 786  
Linux operating system  
  compiling invocation API, 984  
  Java deployment directory in, 564  
  library path in, 946  
  OpenSSL in, 560  
  using GNU C compiler, 943  
list method (*Files*), 110  
List boxes (Swing), 582–588  
  constructing, 582  
  from arrays/vectors, 594–595  
  new components for each cell, 597  
double-clicking in, 585  
fixed cell sizes in, 590  
layout orientation of, 583  
scrolling, 582, 590  
selecting multiple items in, 583, 585  
using with split panes, 733  
values of:  
  computing when called, 589  
  editing, 593–595  
  rendering, 595–599  
  very long, 589–590  
List cell renderers, 595–599  
List interface, 704  
List models (Swing), 588–593  
List selection events, 584  
list/ListFrame.java, 585  
*ListCellRenderer* interface, 595, 597  
  getListCellRendererComponent method, 595–599  
*ListDataListener* interface, 589  
*ListModel* interface  
  getElementAt, getSize methods, 589, 593  
  implementing, 588–589, 594  
listRendering/FontCellRenderer.java, 598  
ListResourceBundle class, 411  
ListSelectionEvent class, 584  
  getValueIsAdjusting method, 584  
*ListSelectionListener* interface, valueChanged  
  method, 584, 588  
*ListSelectionModel* interface  
  selection modes, 612  
  setSelectionMode method, 612, 624  
LITTLE\_ENDIAN constant (*ByteOrder*), 123  
Little-endian order, 70, 118, 407  
Load time, 486  
loadClass method  
  of *ClassLoader*, 496, 498  
  of *URLClassLoader*, 494  
loadLibrary method (*System*), 944, 946  
LOBs (large objects), 316–318  
  creating empty, 318  
  placing in database, 316  
  reading, 316  
Local hosts, 240  
Local names, 198  
Local variables, annotating, 466  
LocalDate class  
  and legacy classes, 369  
  methods of, 356–357, 386, 393  
Localdates/LocalDates.java, 358  
LocalDateTime class, 361  
  and legacy classes, 369  
  atZone method, 361  
  methods of, 386, 393  
Locale, 373–377  
  constructor, 377  
  debugging, 377  
  forLanguageTag method, 377  
  get/setDefault methods, 376–377  
  getCountry method, 377  
  getDisplayCountry, getDisplayLanguage methods,  
    377  
  getDisplayName method, 376–377, 379  
  getInstance method, 393  
  getLanguage method, 377  
  toLanguageTag method, 375, 377  
  toString method, 377

- Locales, 28, 372–377  
and resources bundles, 409–410  
current, 401  
default, 365, 376  
display names of, 376  
formatting styles for, 366, 386–387  
numbers in, 379, 686  
predefined, 375  
supported by classes, 376  
variants in, 373, 410
- LocalTime class, 360–361  
and legacy classes, 369  
methods of, 360, 386, 393
- lock method (`FileChannel`), 126–128
- Locks, 126–128  
for the tail portion of a file, 127  
shared, 127  
unlocking, 127
- Log files, 406
- Log messages, adding to classes, 481–486
- @LogEntry annotation, 481
- Logging, code generation for, 455
- LoggingPermission class, 520
- LoginContext class, 531  
constructor, 536  
`getSubject` method, 536  
`login`, `logout` methods, 531, 536
- LoginException, 536
- LoginModule interface, methods of, 546
- Logins  
committed, 540  
modules for, 532  
custom, 537–546  
separating from action code, 539
- Long class, `MAX_VALUE` constant, 127
- LONG NVARCHAR data type (SQL), 348
- long type  
printing, 61  
type code for, 87, 961  
vs. C types, 947  
writing in binary format, 69
- LONG VARCHAR data type (SQL), 348
- LongBuffer class, 124
- longList/LongListFrame.java, 591
- longList/WordListModel.java, 592
- longs method (`Random`), 37, 40
- LongStream interface, 36–41  
average, `max`, `min`, `sum` methods, 37, 39  
boxed method, 36, 39
- mapToLong method, 36  
of method, 39
- range, `rangeClosed` methods, 36, 39
- summaryStatistics method, 37, 39
- `toArray` method, 36, 39
- LongSummaryStatistics class, 20, 23, 37, 41
- Look-and-feel  
displaying trees in, 645  
handles for subtrees in, 661  
hot keys in, 686  
selecting multiple nodes in, 665
- lookingAt method (`Matcher`), 140
- Lookup tables, 411
- LookupOp class, 843–844  
constructor, 850
- lookupPrintServices method (`PrintServiceLookup`), 874, 877
- lookupStreamPrintServiceFactories method (`StreamPrintServiceFactory`), 879
- LookupTable class, 844
- lostOwnership method (`ClipboardOwner`), 888, 892
- LSB (least significant byte), 70
- LSSoutput interface, 211
- LSSerializer interface, 210
- &lt;, entity reference, 148
- M**
- Mac OS X  
character encodings in, 404  
OpenSSL in, 560  
resources in, 408
- mail method (`Desktop`), 932
- Mail messages/headers, 277–280
- mail/MailTest.java, 279
- mailto: URI scheme, 927
- main method  
executing, 493  
setting security managers in, 516
- makeShape method (`ShapeMaker`), 776–777
- makeVisible method (`JTree`), 653, 657
- Mandelbrot set, 837–838
- Mangling names, 943, 961
- Manifest files, 77, 921
- map method  
of `FileChannel`, 117, 122  
of `Optional`, 14–15  
of `Stream`, 9–10
- Map interface, 817
- mapping method (`Collectors`), 30, 33

**Maps**

concurrent, 25

merging, 24–27, 43

`mapToXxx` methods (`XxxStream`), 36

**mark method**

of `Buffer`, 125–126

of `InputStream`, 50

**Marker annotations**, 465

`markSupported` method (`InputStream`), 50

**MaskFormatter** class, 692–693

constructor, 703

`get/setInvalidCharacters` methods, 692, 703

`get/setPlaceholder` methods, 703

`get/setValidCharacters` methods, 692, 703

`get/setValueContainsLiteralCharacters` methods, 703

`getPlaceholderCharacter` method, 703

**overtype mode**, 693

`setPlaceholderCharacter` method, 693, 703

*Mastering Regular Expressions* (Friedl), 134

**match attribute** (XSLT), 224

`match/HrefMatch.java`, 137

**matcher** method (`Pattern`), 134, 140

**Matcher** class

`end` method, 135, 137, 140–141

`find` method, 137, 140

`group` method, 135, 140–141

`groupCount` method, 141

`lookingAt` method, 140

`matches` method, 134

`quoteReplacement` method, 141

`replaceXxx` methods, 139, 141

`reset` method, 141

`start` method, 135, 137, 140–141

`matches` method (`Matcher`), 134

**Matrices**, transformations of, 802

**max** method (of streams), 12, 37, 39–40

**MAX\_VALUE** constant (`Long`), 127

**maxBy** method (`Collectors`), 30, 33

**maxOccurs** attribute (XML Schema), 174

**MD5** algorithm, 548–549

**MDI** (multiple document interface), 741–742

**Memory addresses**, vs. serial numbers, 84

**Memory mapping**, 116–128

modes of, 117

`memoryMap/MemoryMapTest.java`, 119

**Message digests**, 547–550

**Message signing**, 550–553

**MessageDigest** class

`digest` method, 549–550

extending, 548

`getInstance` method, 548–550

`reset` method, 550

`update` method, 549–550

**MessageFormat** class, 400–404, 414

`applyPattern` method, 401

constructor, 401

`format` method, 401–402

`get/setLocale` methods, 402

ignoring the first limit, 404

**Messages**, formatting, 400–404

**Meta-annotations**, 456, 472–475

**Metadata** (databases), 333–343

**Metal look-and-feel**

hot keys in, 686

internal frames in, 742–744

one-touch expand icons in, 732

selecting multiple nodes in, 665

trees in, 644–645

**Method** class, 457

**Method references**, type use annotations in, 468

**Method verification errors**, 506, 508

**Methods**

adding logging messages to, 481–486

annotating, 456, 466, 470

calling from native code, 963–970

getters/setters, generated automatically, 480

instance, 963–964

mangling names of, 943, 961

native, 939–1001

of annotation interfaces, 463

overriding, 471

signatures of, 961–963

static, 964–965

vetoable, 748

**Microsoft**

compiler, 944

invocation API in, 984

ODBC API of, 282

SQL Server, 291

single active statements in, 301

**MIME (Multipurpose Internet Mail Extensions)**, 824

**MIME types**, 892–894

for print services, 875

**MimeMessage** class, methods of, 278

**min** method (of streams), 12, 37, 39–40

**minBy** method (`Collectors`), 30, 33

minOccurs attribute (XML Schema), 174  
minus, minusXxx methods (Date and Time API), 353, 356–357, 360, 363  
MissingResourceException, 409  
Miter join, 788–789  
Miter limit, 790  
Mixed content (XML), 147  
  parsing, 166  
Model-view-controller design pattern, 588  
Modernist painting example, 211  
Modified UTF-8, 70–72, 407  
  and native code, 949–952  
Monads, 10  
Month enumeration, 355  
  getDisplayName method, 365, 386  
MonthDay class, 357  
Mouse, double-clicking on list components, 585  
move method (Files), 106, 108  
moveColumn method (JTable), 617, 623  
moveTo method (Path2D.Float), 775–776, 786  
moveToXxx methods (JInternalFrame), 758  
moveToXxxRow methods (ResultSet), 325, 327  
MSB (most significant byte), 70  
Multiple-page printing, 861–864  
multipliedBy method (Instant, Duration), 353  
*MutableTreeNode* interface  
  implementing, 641  
  setUserObject method, 642, 649  
MySQL database, 291

## N

\n  
  as line feed, 61, 154, 405  
  in e-mails, 278  
  in regular expressions, 130–131  
NameCallback class, 538  
  constructor, 545  
  methods of, 545  
NamedNodeMap interface, getLength, item methods, 162  
names method (Win32RegKey), 987  
Namespaces, 196–199  
  activating processing of, 174  
  aliases (prefixes) for, 172, 197  
  of attributes, 198  
  of child elements, 197  
  using class loaders as, 496  
Nashorn engine, 430–431  
National character strings, 348

National Institute of Standards and Technology, 234, 548  
native keyword, 940  
Native applications  
  communicating with Java, 903–921  
  data transfers between Java and, 888  
Native methods, 939–1001  
  and garbage collection, 951  
  array elements in, 970–974  
  class references in, 958  
  compiling, 943  
  enumerating keys with, 989  
  error handling in, 974–979  
  exceptions in, 975  
  instance fields in, 956–960  
  invoking Java constructors in, 965–966  
  linking to Java, 946  
  naming, 941–942  
  overloading, 941  
  reasons to use, 940  
  registry access functions in, 988–1001  
  static, 941  
  static fields in, 960–961  
  strings in, 949  
native2ascii program, 408, 410, 414  
NCHAR data type (SQL), 348  
NCHAR, NCLOB data types (SQL), 348  
negated method (Instant, Duration), 353  
Negative byte values, 693  
Nervous text applet, 546  
NetBeans, startup plugins for, 922  
NetPermission class, 519  
Networking, 233–280  
  blocking worker threads indefinitely, 723  
  debugging, 233–236  
  getting huge images from, 826  
newBufferedXxx methods (Files), 104  
newDirectByteBuffer function (C), 973  
newDirectoryStream method (Files), 111, 114  
newDocument method (DocumentBuilder), 209, 212, 227  
newDocumentBuilder method (DocumentBuilderFactory), 150, 160, 209  
newFactory method (RowSetProvider), 329, 332  
newFileSystem method (FileSystems), 115–116  
newGlobalRef function (C), 958  
newInputStream method  
  of Channels, 256  
  of Files, 104–105

newInstance method  
  of DocumentBuilderFactory, 150, 160  
  of SAXParserFactory, 201, 203  
  of TransformerFactory, 213  
  of XMLInputFactory, 207  
  of XMLOutputFactory, 214, 221  
  of XPathFactory, 191, 195  
NewObject function (C), 965, 970, 974  
newOutputStream method  
  of Channels, 250, 257  
  of Files, 104–105  
newPath method (XPathFactory), 195  
newSAXParser method (SAXParserFactory), 201, 203  
NewString function (C), 953  
NewStringUTF function (C), 950–952, 954, 988  
newTransformer method (TransformerFactory), 213, 231  
NewXxxArray functions (C), 973, 988  
next method  
  of *ResultSet*, 298, 300, 321  
  of *XMLStreamReader*, 208  
next, nextOrSame methods (TemporalAdjusters), 359  
nextElement method (*Enumeration*), 659, 987–990  
nextPage method (*CachedRowSet*), 330, 332  
NMTOKEN, NMTOKENS attribute types (DTDs), 167–168  
*Node* interface  
  appendChild method, 209, 212  
  getAttributes method, 154, 161  
  getChildNodes method, 151, 161  
  getFirstChild method, 153, 161  
  getLastChild method, 154, 161  
  getLocalName method, 198–199  
  getNamespaceURI method, 198–199  
  getNextSibling method, 154, 161  
  getNodeXxx methods, 154, 161, 198  
  getParentNode method, 161  
  getPreviousSibling method, 161  
  subinterfaces of, 151  
Node renderer, 648  
nodeChanged method (DefaultTreeModel), 652, 658  
*Nodelist* interface, 151  
  getLength method, 151, 162  
  item method, 151, 162, 169  
Nodes (Swing), 639  
  adding/removing, 652  
  changes in, 653  
  child, 639, 642  
  collapsed, 654  
  connecting lines for, 645–646  
  currently selected, 650  
  editing, 654, 673  
  enumerating, 659–661  
  expanding, 654  
  handles for, 644, 646, 661  
  highlighting, 662  
  identifying, by tree paths, 650  
  making visible, 653  
  parent, 639, 642  
  rendering, 661–664  
  root, 639–647  
  row positions of, 652  
  searching, for a given user object, 660, 666  
  selecting, 665  
  user objects for, 642  
    changing, 652  
nodesChanged method (DefaultTreeModel), 658  
Nondeterministic parsing, 166  
noneMatch method (*Stream*), 13  
Noninterference, of stream operations, 43  
@NotNull annotation, 467–468  
normalize method (*Path*), 102–103  
Normalized color values, 835  
Normalizer class, 395  
  normalize method, 400  
NoSuchAlgorithmException, 550, 573  
NoSuchElementException, 15, 990  
notFilter method (*RowFilter*), 615, 625  
NotSerializableException, 92  
now method (Date and Time API), 352, 356, 360, 363  
NT logins, 531  
NTLoginModule class, 532–533  
NTUserPrincipal class, 533  
NullPointerException, 12, 976  
Number class  
  doubleValue method, 378  
  intValue method, 379  
numberFilter method (*RowFilter*), 615, 625  
NumberFormat class, 378–383, 691  
  format method, 379, 383  
  get/setXxxDigits methods, 383  
  getAvailableLocales method, 376, 379, 383  
  getCurrencyInstance method, 378, 383–384, 691  
  getIntegerInstance method, 689  
  getNumberInstance method, 378, 383, 691  
  getPercentInstance method, 378, 383, 691  
  is/setGroupingUsed methods, 383

- NumberFormat class (*continued*)  
  `isSetParseIntegerOnly` methods, 383  
  `parse` method, 378–379, 383  
  `setCurrency` method, 384  
  supported locales, 376  
`numberFormat/NumberFormatTest.java`, 380  
NumberFormatException, 683  
Numbers  
  filtering, 615  
  floating-point, 372, 378–383, 691  
  formatting, 372, 378–383, 691  
    supported locales for, 379  
  with C, 947  
in regular expressions, 131, 133  
printing, 61  
random, 37  
reading:  
  from files, 56  
  from ZIP archives, 58  
  using locales, 378  
truly random, 570  
writing in binary format, 69  
NUMERIC data type (SQL), 291, 348  
NVARCHAR data type (SQL), 348
- O**
- Object class, `clone` method, 81, 98  
Object inspection tree, 671–680  
Object references, and clipboard, 902  
Object serialization, 80–100  
  cloning with, 98–100  
  file format for, 85–92  
  MIME type for objects in, 893  
  modifying default, 92–94  
  of singletons, 94–95  
  serial numbers for, 82–83  
ObjectInputStream class, 81  
  constructor, 85  
  `readObject` method, 81, 85, 93  
ObjectOutputStream class, 80  
  constructor, 85  
  `defaultWriteObject` method, 92  
  `writeObject` method, 80, 85, 92  
Objects  
  cloning, 98–100  
  fingerprints of, 87  
  life cycle of, 472  
  MIME types for, 893  
  printing, 61  
  reading from an input stream, 81
- saving:  
  in database, 474  
  in output streams, 80, 82  
  in text format, 63–67  
serializable, 80–85  
transferring via clipboard, 898–902  
transmitting over network connection, 84  
type code for, 87, 961  
versioning, 95–98  
`objectStream/ObjectStreamTest.java`, 84  
ODBC API, 282, 284  
of method  
  of Date and Time API, 356, 360–363  
  of `DoubleStream`, 39  
  of `IntStream`, 36, 38  
  of `LongStream`, 39  
  of `Optional`, 16  
  of `Stream`, 5, 8  
ofDateAdjuster method (`TemporalAdjusters`), 359  
OffsetDateTime class, 364  
ofInstant method (`ZonedDateTime`), 363  
ofLocalizedXxx methods (`DateTimeFormatter`), 365, 385, 392  
ofNullable method (`Optional`), 16  
ofPattern method (`DateTimeFormatter`), 367  
oj literal (SQL), 319  
One-touch expand icon, 732  
open method  
  of Desktop, 927, 932  
  of FileChannel, 117, 122  
  of SocketChannel, 250, 256  
openConnection method (`URL`), 259, 265  
Opened nonleaf icons, 648, 661  
openOutputStream method (`SimpleJavaFileObject`), 445, 449  
OpenSSL toolkit, 560–561  
openStream method (`URL`), 257, 265  
Operating system  
  character encodings in, 404  
  resources in, 408  
Optional class, 12–17  
  creating values of, 16  
  empty method, 16  
  flatMap method, 16–19  
  get method, 15–16  
  ifPresent method, 14–15, 41  
  isPresent method, 15–16  
  map method, 14–15  
  of, `ofNullable` methods, 16  
  orElse method, 12, 14–15, 41

orElseGet method, 14–15, 41  
orElseThrow method, 15  
optional keyword, 532  
`optional/OptionalTest.java`, 17  
OptionalXxx classes, 37, 41  
Oracle JVM implementation, 973  
order method (`ByteBuffer`), 118, 123  
ORDER BY statement (SQL), 299  
orElse method (`Optional`), 12, 14–15, 41  
orElseGet method (`Optional`), 14–15, 41  
orElseThrow method (`Optional`), 15  
orFilter method (`RowFilter`), 615, 625  
`org.w3c.dom` package, 150  
`org.w3c.dom.CharacterData` API, 162  
`org.w3c.dom.Document` API, 161, 212  
`org.w3c.dom.Element` API, 161, 212  
`org.w3c.dom.NamedNodeMap` API, 162  
`org.w3c.dom.Node` API, 161, 199, 212  
`org.w3c.dom.NodeList` API, 162  
`org.xml.sax.Attributes` API, 204–205  
`org.xml.sax.ContentHandler` API, 204  
`org.xml.sax.EntityResolver` API, 170  
`org.xml.sax.ErrorHandler` API, 171  
`org.xml.sax.helpers.AttributesImpl` API, 232  
`org.xml.sax.InputSource` API, 171  
`org.xml.sax.SAXParseException` API, 171  
`org.xml.sax.XMLReader` API, 232  
Outer joins, 319  
Outline dragging, 751–760  
`OutOfMemoryError`, 976  
output element (XSLT), 224  
Output streams, 48–59  
  and Unicode, 48  
  buffered, 57–59  
  byte processing in, 57  
  byte-oriented, 48  
  closing, 49, 249  
  filters for, 55–59  
  hierarchy of, 51–55  
  objects in, 80–100  
`OutputStream` class, 48, 52–53, 214  
  close method, 51  
  flush method, 49, 51  
  write method, 48, 50–51  
`OutputStreamWriter` class, 60  
`OverlappingFileLockException`, 127  
Overloading, 941  
`@Override` annotation, 470–471  
Overtype mode, 693  
Overwrite mode, 692

## P

\p, \P, in regular expressions, 130  
Package class, implementing `AnnotatedElement`, 457  
`package-info.java` file, 467  
Packages  
  annotating, 466–467, 470  
  avoiding name clashes with, 196, 496  
packets, 238  
Padding schemes, 568  
Page setup dialog box, 856–857  
`Pageable` interface  
  implementing, 862  
  objects, printing, 874  
`PageAttributes` class (obsolete), 886  
`pageDialog` method (`PrinterJob`), 856–857, 861  
`PageFormat` class  
  `getHeight`, `getWidth` methods, 854, 861  
  `getImageableXxx` methods, 855, 861–862  
  `getOrientation` method, 862  
Pages  
  measurements of, 855  
  multiple, printing, 862–864  
  orientation of, 802, 856, 862  
paint method  
  of `JComponent`, 626, 766  
  of `LayerUI`, 764  
  of `PanelLayer`, 760  
Paint, 797–799  
Paint interface, 797  
paintComponent method  
  of `JComponent`, 596–597, 766  
  of `StrokePanel`, 791  
Paper margins, 854  
Paper sizes, 412, 855  
parallel method (`BaseStream`), 41–45  
`parallel/ParallelStreams.java`, 44  
parallelStream method (`Collection`), 2–5, 41–46  
Parameter variables, annotating, 466  
Parent nodes (Swing), 639, 642  
parse method  
  of `DateTimeFormatter`, 367  
  of `DocumentBuilder`, 160  
  of `LocalXxx`, `ZonedDateTime`, 386, 393  
  of `NumberFormat`, 378–379, 383  
  of `SAXParser`, 201, 203  
  of `XMLReader`, 232  
Parsed character data, 165  
`ParseException`, 379, 383, 693

- Parsers, 149–162  
  checking uniqueness of IDs in, 168, 178  
  pull, 205  
  validating in, 163
- Parsing (XML), 149–162  
  nondeterministic, 166  
  with XML Schema, 174
- `partitioningBy` method (`Collectors`), 28, 31
- `PasswordCallback` class, 538  
  constructor, 546  
  `get setPassword` methods, 546  
  `getPrompt` method, 546  
  `isEchoOn` method, 546
- Password-protected resources, 261–262
- `Path` interface, 101–103  
  `getParent`, `getFileName`, `getRoot` methods, 103  
  `normalize`, `relativize` methods, 102–103  
  `resolve`, `resolveSibling` methods, 102–103  
  `toAbsolutePath`, `toFile` methods, 102–103
- `Path2D` class, `append`, `closePath` methods, 776, 786
- `Path2D.Float` class  
  `curveTo`, `lineTo`, `quadTo` methods, 775, 786  
  `moveTo` method, 775–776, 786
- `pathFromAncestorEnumeration` method  
  (`DefaultMutableTreeNode`), 660
- Paths (file system), 101–103  
  absolute vs. relative, 56, 101–102  
  checking properties of, 108–109  
  filtering, 111  
  relativizing, 102  
  resolving, 58, 101  
  root component of, 101  
  separators in, 56, 101
- Paths (graphics)  
  closing, 776  
  drawing, 775–776
- `Paths` class, 115  
  `get` method, 101, 103
- `Pattern` class, 134  
  `compile` method, 134, 139  
  `matcher` method, 134, 140  
  `split` method, 139–140  
  `splitAsStream` method, 6, 8
- Patterns, 128–141
- #PCDATA element content (DTD), 165
- PDF format, printing, 874
- `peek` method (`Stream`), 11–12
- Pentium processor, little-endian order in, 70
- Percentages, formatting, 378–383,  
  691
- Performance  
  of encryption algorithms, 576  
  of file operations, 116–124
- `Period` class, 357
- Perl programming language, regular  
  expressions in, 134
- Permission class  
  constructor, 530  
  `equals` method, 523  
  extending, 522  
  `getActions` method, 523  
  `getName` method, 525, 530  
  `hashCode` method, 523  
  `implies` method, 523–524, 530
- Permission files, 511
- permission keyword, 517, 520
- Permissions, 510–514  
  call stack of, 513  
  checking, 509–510  
  class hierarchy of, 512  
  commonly used classes for, 517–520  
  for files, 520  
  for users, 531  
  implementing, 522–530  
  implying other permissions, 524  
  in policy files, 514–522  
  mapping code sources to, 510  
  order of, 523  
  property, 521  
  restricting to certain users, 533  
  socket, 521  
  targets of, 520–521  
  permissions/`PermissionTest.java`, 528  
  permissions/`WordCheckPermission.java`, 526
- Permutations, of a string, 705–706
- `@Persistent` annotation, 474
- Personal data, transferring, 567
- Pixels  
  affine transformations on, 843  
  antialiasing, 818–819  
  average value of, 844  
  composing, 807–817  
  interpolating, 797–798, 818, 820, 843  
  reading, 835  
  setting individual, 835–842
- Placeholder characters, in masks, 693
- Placeholders, in message formatting,  
  400–404
- `PlainDocument` class, 681, 689
- Plugins, loading, 494

- `plus, plusXxx` methods (Date and Time API), 353, 356–357, 360, 362–363
- PNG format, 823  
animated, 921  
printing, 874
- Points, in typography, 855
- Policy class, 511, 515
- Policy files, 514–522  
and Java Plug-in, 563  
building, 563–565  
debugging, 564  
locations for, 515  
parsing, 525  
platform-independent, 521  
referencing keystores in, 563  
system properties in, 521  
user roles in, 530–546  
visibility of, 525
- Policy managers, 514–522
- `policytool` program, 522
- Polygons, 769, 776
- `populate` method (*CachedRowSet*), 330, 332
- Pop-up menus, for tray icons, 933
- `PopupMenu` class, 933
- Porter-Duff rules, 808–811
- Portrait orientation, 856
- Ports, 235  
blocking, 234  
in URIs, 258
- `position` method (*Buffer*), 126
- `position` function (XPath), 225
- POSIX-compliant file systems, 109
- PosixFileAttributes* interface, 109
- POST method (HTML), 269–271
- `post/PostTest.java`, 274
- `@PostConstruct` annotation, 470, 472
- PostgreSQL database, 291  
connecting to, 295  
drivers for, 292
- Postorder traversal, 659
- `postOrderEnumeration` method (*DefaultMutableTreeNode*), 659, 663
- PostScript format  
printing, 874, 878  
writing to, 878
- Predefined character class names, 129, 133
- Predefined character classes, 130
- `@PreDestroy` annotation, 470, 472
- `premain` method (Instrumentation API), 487
- `preOrderEnumeration` method (*DefaultMutableTreeNode*), 659, 663
- Prepared statements, 309–316  
caching, 311  
executing, 310
- PreparedStatement* interface  
`clearParameters` method, 316  
`executeXxx, setXxx` methods, 310, 316
- `prepareStatement` method (*Connection*), 310, 315, 322, 326
- `previous` method (*ResultSet*), 323, 327
- previous, previousOrSame methods (*TemporalAdjusters*), 359
- `previousPage` method (*CachedRowSet*), 332
- `preVisitDirectory, postVisitDirectory` methods  
of *FileVisitor*, 112  
of *SimpleFileVisitor*, 115
- Primary keys, 320
- Primitive types  
arrays of, 973  
I/O in binary format in, 51  
streams of, 36–41
- Principal* interface, `getName` method, 537
- Principals (logins), 532
- `print` method  
of *Desktop*, 927, 932  
of *DocPrintJob*, 877  
of *JTable*, 602, 604
- Print dialog box, 852  
displaying page ranges in, 854, 862  
native, 853, 857
- Print preview, 864–873
- Print services, 874–878  
document flavors for, 874–875  
for images, 876  
stream, 878–879
- `print/PrintComponent.java`, 859
- `print/PrintTestFrame.java`, 858
- Printable* interface  
implementing, 852, 857  
objects, printing, 874  
`print` method, 852, 860, 862, 864–865
- Printer graphics context, 863
- PrinterException, 853
- PrinterJob class  
`defaultPage` method, 861  
`getPrinterJob` method, 852, 861  
`pageDialog` method, 856–857, 861  
`print` method, 853–854, 861  
`printDialog` method, 852–853, 861

- PrinterJob class (*continued*)  
    setPageable method, 862, 873  
    setPrintable method, 861
- printf function (C), 947
- printf1/Printf1.c, 948
- printf1/Printf1.java, 948
- printf1/Printf1Test.java, 949
- printf2/Printf2.c, 954
- printf2/Printf2.java, 954
- printf2/Printf2Test.java, 953
- printf3/Printf3.c, 967
- printf3/Printf3.java, 967
- printf3/Printf3Test.java, 966
- printf4/Printf4.c, 976
- printf4/Printf4.java, 978
- printf4/Printf4Test.java, 979
- Printing, 851–887  
    clipped areas, 854  
    counting pages during, 854  
    images, 852–862  
    layout of, 862–863  
    multipage documents, 861–864  
    number of copies for, 879  
    page orientation of, 802, 856  
    paper sizes in, 855  
    quality of, 882  
    selecting settings for, 852  
    starting, 509, 852  
    text, 852–862  
    using:  
        banding for, 854  
        transformations for, 864
- Printing attributes, 879–887  
    adding, 881  
    categories of, 881–882  
    checking values of, 882  
    hierarchy of, 880  
    retrieving, 882
- PrintJob class (obsolete), 852
- PrintJobAttribute interface, 879  
    printing attributes of, 882–885
- PrintJobAttributeSet interface, 880
- println method (`System.out`), 405–406
- PrintPreviewCanvas class, 864
- PrintPreviewDialog class, 864–873
- PrintQuality class, 882
- PrintRequestAttribute interface, 879  
    printing attributes of, 882–885
- PrintRequestAttributeSet interface, 852, 880
- PrintService interface  
    createPrintJob method, 874, 877  
    getAttributes method, 886  
    getName method, 874
- printService/PrintServiceTest.java, 876
- PrintServiceAttribute interface, 879  
    printing attributes of, 882–885
- PrintServiceAttributeSet interface, 880
- PrintServiceLookup class, lookupPrintServices method, 874, 877
- PrintStream class, 61
- PrintWriter class, 56, 60–62  
    checkError method, 61–62  
    constructor, 62  
    print method, 61–62, 963–964  
    printf, println methods, 61–62
- Privacy Enhanced Mail (PEM) format, 560
- Private keys, 550–563, 576
- PrivilegedAction interface, 539  
    implementing, 532  
    run method, 532, 536
- PrivilegedExceptionAction interface, 533  
    run method, 537
- processAnnotations method (`ActionListenerInstaller`), 457–458
- Processing instructions (XML), 149
- Processing tools, 455
- Processor interface, 476
- processXxxEvent methods (`LayerUI`), 761, 764
- Programmer’s Day, 356
- Programs. *See* Applications
- Progress bars (Swing), 719–722  
    completion percentage in, 720  
    indeterminate, 720  
    minimum/maximum values of, 719  
    on a splash screen, 922–923
- Progress monitors (Swing), 722–726  
    cancellation requests in, 723  
    for input streams, 726–731  
    timers in, 723
- progressBar/ProgressBarFrame.java, 720
- ProgressMonitor class, 722–726  
    close method, 723, 731  
    constructor, 731  
    isCanceled method, 723, 731  
    setMillisToXxx methods, 724  
    setNote method, 731  
    setProgress method, 723, 731
- progressMonitor/ProgressMonitorFrame.java, 724

ProgressMonitorInputStream class, 726–731  
  constructor, 731  
  read method, 727  
progressMonitorInputStream/TextFrame.java, 728  
Properties  
  constrained, 748–750  
  generated automatically, 480  
Properties class, 144  
Property files, 144  
  character encoding of, 410  
  event handlers in, 438  
  for resources bundles, 409–410  
  for string resources, 408  
  for strings, 410  
  no passwords in, 278  
Property permissions, 521  
@Property annotation, 480  
PropertyChangeEvent class  
  getNewValue method, 750, 759  
  getPropertyNames method, 759  
PropertyPermission class, 517  
PropertyVetoException class, 744–750  
  constructor, 760  
Protection domains, 512  
ProtectionDomain class  
  constructor, 514  
  getCodeSource, implies methods, 514  
Prototype cell values, 590  
Proxy objects, 457  
  for annotated methods, 458  
PUBLIC identifier (DTD), 210  
Public certificates, keystore for, 563  
Public key ciphers, 550–558, 575–579  
  performance of, 576  
Public Key Cryptography Standard (PKCS)  
  #5, 568  
Pull parsers, 205  
PushbackInputStream class, 57  
  constructor, 59  
  unread method, 59  
put method  
  of *Bindings*, 434  
  of ByteBuffer, 122  
  of CharBuffer, 124  
  of ScriptEngine, 433  
  of ScriptEngineManager, 433  
putClientProperty method (`JComboBox`), 645, 650,  
  751  
putNextEntry method (`ZipOutputStream`), 77–78  
putXxx methods (ByteBuffer), 118, 123

**Q**  
\Q, in regular expressions, 130  
QBE (query by example) tools, 287  
QuadCurve2D class, 769, 774  
QuadCurve2D.Double class, 785  
Quadratic curves, 774–775  
quadTo method (`Path2D.Float`), 775, 786  
Qualified names, 198  
Quantifiers, 129  
Queries (databases), 288–290  
  by example, 287  
  executing, 298, 309–321  
  multiple, 301  
  populating row sets with results of, 330  
  preparing, 309–316  
  returning multiple results, 319–320  
query/QueryTest.java, 312  
&quot;, entity reference, 148  
quoteReplacement method (`Matcher`), 141

**R**  
R programming language, 430, 438  
\r line feed character, 61, 154, 405  
  in e-mails, 278  
\r, \R, in regular expressions, 130, 132  
Race conditions, 42  
Random class, 570  
  methods of, 37, 40  
Random numbers, 37  
Random-access files, 72–76  
randomAccess/RandomAccessTest.java, 74  
RandomAccessFile class, 72–76, 116  
  constructor, 76  
  getChannel method, 121  
  getFilePointer method, 72, 76  
  length method, 73, 76  
  seek method, 72, 76  
Randomness, 570  
range, rangeClosed methods (`XxxStream`), 36,  
  38–39  
Raster class  
  getDataElements method, 837, 841  
  getPixel, getPixels methods, 836, 841  
rasterImage/RasterImageFrame.java, 839  
read method  
  of CipherInputStream, 575  
  of FileInputStream, 48  
  of ImageIO, 823, 831  
  of ImageReader, 832  
  of InputStream, 48–50

read method (*continued*)  
  of *ProgressMonitorInputStream*, 727  
  of *ZipInputStream*, 77  
read/fontdialog.xml, 185  
read/gridbag.dtd, 187  
read/gridbag.xsd, 188  
read/GridBagPane.java, 181  
read/GridBagTest.java, 179  
*Readable* interface, 52  
  read method, 53, 55  
*ReadableByteChannel* interface, 250  
readAllXxx methods (*Files*), 104  
readAttributes method (*Files*), 109  
Reader class, 48, 52  
  read method, 51  
READER class (*DocFlavor*), 875  
readExternal method (*Externalizable*), 93–94  
readFixedString method (*DataIO*), 73–74  
readLine method (*Console*), 63  
readObject method  
  of *Date*, 93  
  of *ObjectInputStream*, 81, 85, 93  
ReadOnlyBufferException, 117  
readResolve method (*Serializable*), 95  
readThumbnail method (*ImageReader*), 833  
readXxx methods (*DataInput*), 70–71, 81  
REAL data type (SQL), 291, 348  
Receiver parameters, 469  
Rectangle2D class, 769  
RectangularShape class, 770  
reduce method (*Stream*), 33–35  
Reductions, 12, 33–35  
ref attribute (XML Schema), 173  
Reflection  
  accessing fields of another class with, 509  
  constructing:  
    class tree with, 665  
    static field names with, 387  
  enumerating fields from a variable with,  
    674  
ReflectPermission class, 519  
regex/RegexTest.java, 136  
regexFilter method (*RowFilter*), 615, 625  
Registry editor, 986, 990  
Registry keys, 987–989  
Regular expressions, 128–141  
  escapes in, 64, 129  
  filtering, 615  
  formatters for, 694  
  grouping in, 133, 135–136  
  in DTDs, 165  
  predefined character classes in, 129, 133  
  quantifiers in, 129  
  replacing all matches with, 139  
relative method (*ResultSet*), 323, 327  
Relative URLs, 563  
Relativization, of an absolute URL, 259  
relativize method (*Path*), 102–103  
releaseSavepoint method (*Connection*), 345–346  
ReleaseStringChars function (C), 953  
ReleaseStringUTFChars function (C), 951–952,  
  954  
ReleaseXxxArrayElements functions (C), 972, 974  
reload method (*DefaultTreeModel*), 653, 658  
remaining method (*Buffer*), 125–126  
Remote objects, MIME type for, 893  
remove method  
  of *AttributeSet*, 886  
  of *DocumentFilter*, 702  
  of *SystemTray*, 936  
removeActionListener method (*TrayIcon*), 937  
removeCellEditorListener method (*CellEditor*), 638  
removeColumn method (*JTable*), 617, 623  
removeElement method (*DefaultListModel*), 595  
removeNodeFromParent method (*DefaultTreeModel*), 652,  
  658  
removeTabAt method (*JTabbedPane*), 736, 740  
removeTreeModelListener method (*TreeModel*), 672,  
  680  
removeUpdate method (*DocumentListener*), 682, 685  
RenderableImage interface, 874  
Rendering (Swing)  
  cells, 626–638  
  columns, 609–610  
  headers, 627  
  lists, 595–599  
  nodes, 661–664  
Rendering hints, 766, 817–823  
Rendering pipeline, 766–768  
RenderingHints class, 817, 823  
renderQuality/RenderQualityTestFrame.java, 820  
Renjin project, 430–431, 439  
@Repeatable annotation, 471  
replace method (*DocumentFilter*), 689, 702  
replaceXxx methods (*Matcher*), 139, 141  
required keyword, 532  
#REQUIRED attribute (DTD), 167  
requisite keyword, 532  
RescaleOp class, 843, 850  
Rescaling operation, 843

- reset method
  - of Buffer, 125–126
  - of InputStream, 50
  - of Matcher, 141
  - of MessageDigest, 550
- reshape method (`JInternalFrame`), 743, 758
- resolve, resolveSibling methods (`Path`), 102–103
- resolveEntity method (`EntityResolver`), 164, 170
- Resolving
  - classes, 492
  - relative URLs, 259
- Resource bundles, 408–413
  - loading, 410–411
  - locating, 409–410
  - lookup tables for, 411
  - naming, 411
  - searching for, 411
- Resource editor, 408
- @Resource annotation, 349, 470, 472
- ResourceBundle class
  - extending, 411–412
  - getBundle method, 409–412
  - getKeys method, 412–413
  - getObject method, 411–412
  - getString method, 410, 412
  - getStringArray method, 412
  - handleGetObject method, 412–413
- ResourceBundle.Control class, getCandidateLocales method, 410
- Resources
  - annotations for managing, 472
  - hierarchy of, 409
  - injection, 472
- @Resources annotation, 470
- Response headers, 262–263
- Response page, 268
- Result interface, 226–227, 349
- Result sets (databases)
  - accessing columns in, 299
  - analyzing, 298
  - closing, 302
  - for multiple queries, 301
  - iterating over rows in, 321
  - metadata for, 334
  - numbering rows in, 323
  - order of rows in, 299
  - retrieving multiple, 319–320
  - scrollable, 321–324
  - updatable, 321, 324–328
- ResultSet interface, 329
  - absolute method, 323, 327
  - beforeFirst, afterLast methods, 323, 327
  - cancelRowUpdates method, 325, 328
  - close method, 301
  - concurrency values, 322, 324, 326, 328
  - deleteRow method, 325, 328
  - findColumn method, 301
  - first, last methods, 323, 327
  - getBlob, getClob methods, 316–317
  - getConcurrency method, 323–324, 327
  - getMetaData method, 334, 343
  - getRow method, 323, 327
  - getType method, 323, 326
  - getWarnings method, 304
  - getXxx methods, 299, 301
  - insertRow method, 325, 328
  - isClosed method, 301
  - isFirst, isLast, isBeforeFirst, isAfterLast methods, 324, 327
  - iteration protocol, 298
  - moveToXxxRow methods, 325, 327
  - next method, 298, 300, 321
  - previous method, 323, 327
  - relative method, 323, 327
  - type values, 322, 324, 326, 328
  - updateObject method, 301
  - updateXxx methods, 324–325, 328
- ResultSetMetaData interface, 334
  - getColumnXxx methods, 334, 343
- Retention policies, 472
- @Retention annotation, 456, 470, 472
- retire/Retire.java, 415
- retire/RetireResources\_de.java, 427
- retire/RetireResources\_zh.java, 427
- retire/RetireResources.java, 426
- retire/RetireStrings\_de.properties, 428
- retire/RetireStrings\_zh.properties, 428
- retire/RetireStrings.properties, 428
- Retirement calculator example, 413–428
- RETURN\_GENERATED\_KEYS field (`Statement`), 321
- rewind method (Buffer), 125–126
- RFC 2368, 927
- RFC 2396, 258
- RFC 2616, 261
- RFC 2911, 885
- RFC 821, 278
- RGB color model, 807, 836
- Rivest, Ronald, 548
- Role-based authentication, 537

- rollback method (*Connection*), 344–346  
 Root certificate, 563  
 Root component (file system), 101  
 Root element (XML), 147, 154  
     referencing schemas in, 172  
 Root node (Swing), 639–647  
     handles for, 646–647  
     horizontal lines separating children of, 646  
 rotate method (*Graphics2D*), 800, 804  
 Rotation, 800  
     and interpolating pixels, 818, 843  
     with center point, 801  
 Round cap, round join, 788  
 Round join, 789  
 Rounded rectangles, 772  
*RoundEnvironment* interface, 477  
*RoundRectangle2D* class, 769, 772  
*RoundRectangle2D.Double* class, 785  
 Row sets (databases), 328–333  
     cached, 329–334  
     constructing, 329  
     modifying, 330  
     page size of, 330  
*RowFilter* class, 615–617  
     include method, 616, 624  
     xxxFilter methods, 615, 625  
*RowFilter.Entry* class, 616  
*RowId* interface, 348  
 ROWID data type (SQL), 348  
 Rows (databases), 285  
     deleting/inserting, 325  
     iterating through, 324  
     order of, in result set, 299  
     retrieving, 348  
     selecting, 288  
     updating, 325  
 Rows (Swing)  
     filtering, 615–617  
     height of, 612  
     hiding, 616  
     margins of, 612  
     position, in a node, 652  
     resizing, 612  
     selecting, 602, 612  
     sorting, 602, 614–615  
         turning off, 614  
*RowSet* interface, 328–332  
     execute method, 330, 332  
     get/setCommand methods, 330, 332  
     get/setPassword methods, 330–331  
     get/setURL methods, 330–331  
     get setUsername methods, 330–331  
*RowSetFactory* interface, createXxxRowSet methods, 329, 333  
*RowServiceProvider* class, newFactory method, 329, 332  
 RSA algorithm, 551, 576  
*rsa/RSATest.java*, 577  
*rt.jar* file, 493–494  
 RTF (Rich Text Format), 712  
 Ruby programming language, 430  
 run method  
     of *PrivilegedAction*, 532, 536  
     of *PrivilegedExceptionAction*, 537  
     of *Tool*, 447  
*Runnable* interface, 245  
 Runtime class, exit method, 509  
*runtimeAnnotations/ActionlistenerFor.java*, 461  
*runtimeAnnotations/ActionListenerInstaller.java*, 459  
*RuntimePermission* class, 518
- S**
- S (short), type code, 87, 961  
 \s, \\$, in regular expressions, 131  
*@SafeVarargs* annotation, 470  
 Sample values, 835  
 Sandbox, 510–514  
 SASL (Simple Authentication and Security Layer), 579  
 Save points (databases), 345  
*Savepoint* interface, getSavepointXxx methods, 347  
 SAX (Simple API for XML) parser, 149, 199–205  
     activating namespace processing in, 201  
*sax/SAXTest.java*, 202  
*SAXParseException* class, getXxxNumber methods, 171  
*SAXParser* class, parse method, 201, 203  
*SAXParserFactory* class  
     is/setNameSpaceAware methods, 203  
     is/isValidating methods, 203  
     newInstance, newSAXParser methods, 201, 203  
         setFeature method, 202  
*SAXResult* class, 227  
*SAXSource* class, 226–227  
     constructor, 231  
 Scalar functions, 318–319  
 scale method (*Graphics2D*), 800, 804  
 Scaling, 800  
     and interpolating pixels, 818, 820  
 Scanner class, 62, 250  
     useLocale method, 380

- Scheduling applications  
and time zones, 355, 361  
computing dates for, 358–360  
schema element (XML Schema), 174
- Schemas, 342
- Script engines, 430–431  
adding variable bindings to, 432  
implementing Java interfaces, 435  
invoking, 430  
invoking functions in, 435–436  
`script/ScriptTest.java`, 439  
`ScriptContext` interface, 433  
  `getXxx/setXxx` methods of, 434
- `ScriptEngine` interface  
  `createBindings` method, 433  
  `eval` method, 431–433  
  `get`, `put` methods, 433  
  `getContext` method, 434
- `ScriptEngineFactory` interface  
  `getExtensions` method, 431  
  `getMethodCallSyntax` method, 435  
  `getMimeTypes` method, 431  
  `getNames` method, 431
- `ScriptEngineManager` class  
  `get` method, 433  
  `getEngineXxx` methods, 431  
  `put` method, 433
- Scripting languages, 430–442  
advantages of, 430  
supported, 430
- Scripts  
accessing classes through Java interfaces  
  in, 436  
compiling, 437  
executing, 437  
  concurrently in multiple threads, 432  
invoking, 431  
  multiple, 432  
redirecting I/O of, 434  
using Java method call syntax in, 435
- ScrollPane (Swing)  
with lists, 582, 590  
with tabbed panes, 736  
with tables, 601  
  with trees, 653–654
- `scrollPathToVisible` method (`JTree`), 653, 657
- Seconds, leap, 352
- Secret key, generating, 570
- `SecretKey` interface, 569
- `SecretKeySpec` class, 574
- Secure random generator, 570
- `SecureRandom` class, `setSeed` method, 570
- Securing Java* (McGraw/Felten), 513
- Security, 491–579  
  bytecode verification, 504–508  
  class loaders, 492–508  
  code signing, 561–567  
  different levels of, 547  
  digital signatures, 546–567  
  encryption, 567–579  
  user authentication, 530–546
- Security managers, 509–530  
  custom, 510
- Security policy, 510
- `SecurityException`, 510, 512
- `SecurityManager` class  
  `checkExit` method, 509, 512  
  `checkPermission` method, 512–513, 523–524  
  `checkRead` method, 513
- `SecurityPermission` class, 519
- `seek` method (`RandomAccessFile`), 72, 76
- “Seek forward only” mode (`ImageInputStream`), 825
- select attribute (XSLT), 225
- SELECT statement (SQL), 288–289  
executing, 298  
for LOBs, 316  
multiple, in a query, 319  
not supported in batch updates, 345
- Selection models, 612
- separator constant (`File`), 56
- Separators (file system), 56, 101
- sequence element (XML Schema), 173
- Serial numbers, 82–83  
  vs. memory addresses, 84
- `serialClone/SerialCloneTest.java`, 98
- `Serializable` class, 98
- `Serializable` interface, 81, 87, 474  
  `readResolve` method, 95
- `@Serializable` annotation, 474
- `SerializablePermission` class, 519
- Serialization, 80–100  
  cloning with, 98–100  
  file format for, 85–92  
  MIME type for objects in, 893  
  modifying default, 92–94  
  of singletons, 94–95  
  serial numbers for, 82–83
- `serialTransfer/SerialTransferFrame.java`, 900
- `serialver` program, 96

serialVersionUID constant, 96  
server/EchoServer.java, 243  
Servers  
    accessing, 257–277  
    connecting clients to, 236–238  
    implementing, 241–249  
    invoking programs, 268  
Server-side programs, 267–277  
ServerSocket class, 241–249  
    accept method, 242, 245  
    close method, 245  
    constructor, 245  
SERVICE\_FORMATTED class (*DocFlavor*), 875  
Service provider interfaces, 824  
Servlets, 268, 449–454  
    hot deployment of, 496  
Session class, setDebug method, 279  
Set interface, containsAll, equals methods, 525  
set/Item.java, 485  
set/SetTest.java, 486  
setAllowsChildren, setAsksAllowsChildren methods  
    (DefaultMutableTreeNode), 648, 650  
setAllowUserInteraction method (*URLConnection*), 259,  
    261, 266  
setAttribute, setAttributeNS methods (*Element*), 209,  
    212  
setAutoCommit method (*Connection*), 346  
setAutoCreateRowSorter method (*JTable*), 602, 604,  
    614  
setAutoResizeMode method (*JTable*), 611, 622  
setBinaryStream method (*Blob*), 317  
SetBooleanArrayRegion function (C), 973  
SetBooleanField function (C), 961  
setBottomComponent method (*JSplitPane*), 735  
SetByteArrayRegion function (C), 973–974, 988  
SetByteField function (C), 961  
setCellEditor method (*TableColumn*), 629, 637  
setCellRenderer method  
    of *JList*, 597, 599  
    of  *TableColumn*, 637  
setCellSelectionEnabled method (*JTable*), 613, 622  
setCharacterStream method (*Clob*), 318  
SetCharArrayRegion function (C), 973–974  
SetCharField function (C), 961  
setClip method (*Graphics*), 805–806, 854  
setClosable method (*JInternalFrame*), 758  
setClosed method (*JInternalFrame*), 748, 758  
setClosedIcon method (*DefaultTreeCellRenderer*), 664  
setColumnSelectionAllowed method (*JTable*), 613, 622  
setCommand method (*RowSet*), 330, 332  
setComparator method (*DefaultRowSorter*), 614, 624  
setComponentAt method (*JTabbedPane*), 740  
setComposite method (*Graphics2D*), 767, 809, 816  
setConnectTimeout method (*URLConnection*), 259,  
    266  
setContentHandler method (*XMLReader*), 232  
setContentPane method (*JInternalFrame*), 758  
setContents method (*Clipboard*), 889, 891  
setContextClassLoader method (*Thread*), 495, 503  
setContinuousLayout method (*JSplitPane*), 733, 735  
setCRC method (*ZipEntry*), 79  
setCurrency method (*NumberFormat*), 384  
setDataElements method (*WritableRaster*), 837, 841  
setDate method (*PreparedStatement*), 310, 316  
setDebug method (*Session*), 279  
setDecomposition method (*Collator*), 399  
setDefault method (*Locale*), 376–377  
setDefaultNamespace method (*XMLStreamWriter*), 221  
setDefaultRenderer method (*JTable*), 626  
setDouble method (*PreparedStatement*), 310, 316  
SetDoubleArrayRegion function (C), 973–974  
SetDoubleField function (C), 956, 961  
setDoXxx methods (*URLConnection*), 259–260, 265,  
    270  
setDragEnabled method (of Swing components),  
    904, 909–910  
setDragMode method (*JDesktopPane*), 751, 757  
setDropAction method (*TransferSupport*), 919  
setDropMode method (of Swing components),  
    913, 919  
setEditable method, 714  
setEditor method (*JSpinner*), 710  
setEntityResolver method (*DocumentBuilder*), 164,  
    170  
setErrorHandler method (*DocumentBuilder*), 170  
setErrorWriter method (*ScriptContext*), 434  
setFeature method (*SAXParserFactory*), 202  
setFillsViewportHeight method (*JTable*), 602, 604  
setFixedCellXxx methods (*JList*), 593  
SetFloatArrayRegion function (C), 973–974  
SetFloatField function (C), 961  
setFocusLostBehavior method (*JFormattedTextField*),  
    688, 701  
setIconIcon method (*JInternalFrame*), 743, 759  
setFrom method (*MimeMessage*), 278  
setGroupingUsed method (*NumberFormat*), 383  
setHeaderXxx methods ( *TableColumn*), 627, 637  
setIcon method (*JInternalFrame*), 758  
setIconAt method (*JTabbedPane*), 740  
setIconifiable method (*JInternalFrame*), 758

setIfModifiedSince method (`URLConnection`), 259, 261, 266  
setIgnoringElementContentWhitespace method (`DocumentBuilderFactory`), 169, 171  
setImage, setImageAutoSize methods (`TrayIcon`), 936  
setImageURL method (`SplashScreen`), 926  
setIndeterminate method (`JRootPane`), 720, 730  
setInput method (`ImageReader`), 832  
setInt method (`PreparedStatement`), 310, 316  
SetIntArrayRegion function (C), 973–974  
SetIntField function (C), 956, 961, 989  
setInvalidCharacters method (`MaskFormatter`), 692, 703  
setLayerEventMask method (`JLayer`), 764  
setLayoutOrientation method (`JList`), 587  
setLeafIcon method (`DefaultTreeCellRenderer`), 664  
setLeftComponent method (`JSplitPane`), 735  
setLenient method (`DateFormat`), 691  
setLevel method (`ZipOutputStream`), 78  
setLocale method (`MessageFormat`), 402  
setLogWriter method (`DriverManager`), 296  
SetLongArrayRegion function (C), 973–974  
SetLongField function (C), 961  
setMaximizable method (`JInternalFrame`), 758  
setMaximum method  
  of `JInternalFrame`, 745, 758  
  of `JProgressBar`, 719, 730  
setMaximumXxxDigits, setMinimumXxxDigits methods (`NumberFormat`), 383  
setMaxWidth method (`TableColumn`), 611, 623  
setMethod method  
  of `ZipEntry`, 79  
  of `ZipOutputStream`, 78  
setMillisToXxx methods (`ProgressMonitor`), 724  
setMinimum method (`JProgressBar`), 719, 730  
setMinWidth method (`TableColumn`), 611, 623  
setMnemonicAt method (`JTabbedPane`), 737, 741  
setName method (`NameCallback`), 545  
setNamespaceAware method  
  of `DocumentBuilderFactory`, 174, 198–199, 201, 209  
  of `SAXParserFactory`, 203  
setNote method (`ProgressMonitor`), 731  
SetObjectArrayElement function (C), 972, 974–975  
SetObjectField function (C), 956, 961  
setOneTouchExpandable method (`JSplitPane`), 732, 734  
setOpenIcon method (`DefaultTreeCellRenderer`), 664  
setOutput method (`ImageWriter`), 834  
setOutputProperty method (`Transformer`), 213  
setOverwriteMode method (`DefaultFormatter`), 702  
setPage method (`JEditorPane`), 714, 718  
setPageable method (`PrinterJob`), 862, 873  
setPageSize method (`CachedRowSet`), 330, 332  
setPaint method (`Graphics2D`), 767, 797–798  
setParseIntegerOnly method (`NumberFormat`), 383  
setPassword method  
  of `PasswordCallback`, 546  
  of `RowSet`, 330–331  
setPixel, setPixels methods (`WritableRaster`), 835, 841  
setPlaceholder method (`MaskFormatter`), 703  
setPlaceholderCharacter method (`MaskFormatter`), 693, 703  
setPopupMenu method (`TrayIcon`), 936  
setPreferredSize method (`JComponent`), 685  
setPreferredWidth method (`TableColumn`), 611, 623  
setPrefix method (`XMLStreamWriter`), 221  
setPrintable method (`PrinterJob`), 861  
setProgress method (`ProgressMonitor`), 723, 731  
setProperty method (`XMLInputFactory`), 206–207  
setPrototypeCellValue method (`JList`), 593  
setReader method (`ScriptContext`), 434  
setReadTimeout method (`URLConnection`), 259, 266  
setRenderingHint, setRenderingHints methods  
  (`Graphics2D`), 766, 817–819, 823  
setRequestProperty method (`URLConnection`), 259, 261, 266  
setResizable method  
  of `JInternalFrame`, 758  
  of `TableColumn`, 611, 623  
setRightComponent method (`JSplitPane`), 735  
setRootVisible method (`JTree`), 647, 649  
setRowFilter method (`DefaultRowSorter`), 615, 624  
setRowHeight, setRowMargin, setRowSelectionAllowed methods (`JTable`), 612, 622  
setRowSorter method (`JTable`), 614, 623  
Sets  
  comparing, 525  
  flattening, 10  
setSavepoint method (`Connection`), 346  
setSecurityManager method (`System`), 516  
setSeed method (`SecureRandom`), 570  
setSelected method (`JInternalFrame`), 744, 758  
setSelectedIndex method (`JTabbedPane`), 736, 740  
setSelectionMode method  
  of `JList`, 583, 587  
  of `ListSelectionModel`, 612, 624  
SetShortArrayRegion function (C), 973–974  
SetShortField function (C), 961  
setShowsRootHandles method (`JTree`), 646, 649

setSize method (*ZipEntry*), 79  
setSortable method (*DefaultRowSorter*), 614, 624  
setSoTimeout method (*Socket*), 238–239  
*SetStaticXxxField* functions (C), 960–961  
setStrength method (*Collator*), 399  
setString method (*PreparedStatement*), 310, 316  
setString, *setStringPainted* methods (*JProgressBar*), 720, 730  
setStringConverter method (*TableRowSorter*), 624  
setStroke method (*Graphics2D*), 767, 788, 796  
setSubject method (*MimeMessage*), 278  
setTableName method (*CachedRowSet*), 331–332  
setTabXxx methods (*JTabbedPane*), 737, 741  
setText method  
    of *JFormattedTextField*, 688  
    of *MimeMessage*, 278  
setTitleAt method (*JTabbedPane*), 740  
setTooltip method (*TrayIcon*), 936  
setTopComponent method (*JSplitPane*), 735  
setToXxx methods (*AffineTransform*), 802, 804  
setTransferHandler method  
    of *JComponent*, 906, 909  
    of *JFrame*, 919  
setTransform method (*Graphics2D*), 802, 804  
setURL method (*RowSet*), 330–331  
setUseCaches method (*URLConnection*), 259, 261, 266  
setUsername method (*RowSet*), 330–331  
setUserObject method (*MutableTreeNode*), 642, 649  
setValidating method  
    of *DocumentBuilderFactory*, 169, 171  
    of *SAXParserFactory*, 203  
setValidCharacters method (*MaskFormatter*), 692, 703  
setValue method  
    of *AbstractSpinnerModel*, 705, 712  
    of *JFormattedTextField*, 687–688, 701  
    of *JProgressBar*, 719, 730  
    of *JSpinner*, 710  
    of *Win32RegKey*, 988  
setValueAt method (*TableModel*), 608, 632  
setValueContainsLiteralCharacters method  
    (*MaskFormatter*), 703  
setVisible method (*JInternalFrame*), 744, 759  
setVisibleRowCount method (*JList*), 583, 587  
setWidth method (*TableColumn*), 611, 623  
setWriter method (*ScriptContext*), 434  
SGML (Standard Generalized Markup Language), 145  
SHA-1 algorithm, 86, 547–549  
Shape interface, 769, 787  
shape/ShapeTest.java, 777  
ShapeMaker class  
    getPointCount method, 776  
    makeShape method, 776–777  
ShapePanel class, 777  
Shapes, 769–786  
    antialiasing, 817  
    clipping, 766, 805–807  
    combining, 786–788  
    control points of, 776–777  
    creating, 767  
    drawing, 766–769  
    filling, 766–767, 797  
    rendering, 768  
    transformations of, 766  
Shared libraries, 946, 985  
    initialization blocks in, 947  
shear method (*Graphics2D*), 800, 804  
Shear, 800  
short type  
    printing, 61  
    type code for, 87, 961  
    vs. C types, 947  
    writing in binary format, 69  
ShortBuffer class, 124  
ShortLookupTable class, 844, 851  
shouldSelectCell method (*CellEditor*), 631, 638  
show method (*JInternalFrame*), 759  
showInternalXxxDialog methods (*JOptionPane*), 750  
shutdownXxx methods (*Socket*), 249  
Side files, 475  
Signatures, 546–567  
    generating, 962  
    mangling, 961–963  
signed/FileReadApplet.java, 566  
Simple types, 172  
SimpleDateFormat class, *toPattern* method, 712  
SimpleDoc class, 874, 878  
SimpleFileVisitor class, 113  
    visitFile, visitFileFailed methods, 113, 115  
    xxxVisitDirectory methods, 115  
SimpleJavaFileObject class  
    extending, 445  
    getCharContent method, 445, 449  
    openOutputStream method, 445, 449  
SimpleLoginModule class, checkLogin, initialize  
    methods, 538  
SimpleScriptContext class, 433  
simpleType element (XML Schema), 172  
SimulatedActivity class, 720  
Single value annotations, 465

- Singletons  
of the splash screen, 923  
serializing, 94–95
- SISC Scheme programming language, 430, 438–439
- size method  
of *BasicFileAttributes*, 109  
of files, 108–109
- skip method  
of *InputStream*, 50  
of *Stream*, 10–11
- skipBytes method (*DataInput*), 71
- SMALLINT data type (SQL), 291, 348
- SMTP (Simple Mail Transport Protocol), 277–280
- Socket class  
connect method, 239  
constructor, 238–239  
getInputStream method, 237–238, 242  
getOutputStream method, 238, 242  
isClosed, isConnected methods, 239  
isXxxShutdown, shutdownXxx methods, 249  
setSoTimeout method, 238–239
- Socket permissions, 521
- socket/SocketTest.java, 236
- SocketChannel class, 250  
open method, 250, 256
- SocketPermission class, 517
- Sockets  
half-closing, 249  
interrupting, 250–257  
opening, 237, 509  
timeouts, 238–239
- SocketTimeoutException, 238, 266
- Software updates, 932
- Solaris operating system  
compiling invocation API, 984  
using Sun compiler, 943
- sort method (*Collections*), 394
- sorted method (*Stream*), 11–12
- Source interface, 226, 349
- Source files  
character encoding of, 407–408  
locating, 444  
reading from disk, 444
- Source-level annotations, 475–480
- Space. *See* Whitespace
- SPARC processor, big-endian order in, 70
- spinner/PermutationSpinnerModel.java, 708  
spinner/SpinnerFrame.java, 706
- SpinnerDateModel class, 711
- SpinnerListModel class, 704, 711
- SpinnerNumberModel class, 704, 711
- Spinners (Swing), 703–712  
increment value in, 704  
traversal order in, 704–705
- Splash screens, 921–926  
drawing on, 922–923  
replacing with follow-up window, 923
- SplashScreen class  
close method, 926  
createGraphics method, 926  
get/setImageURL methods, 926  
getBounds method, 926  
getSplashScreen method, 923, 926  
update method, 922, 926
- splashScreen/SplashScreenTest.java, 923
- split method  
of Pattern, 139–140  
of String, 64
- Split panes (Swing), 732–735
- splitAsStream method (Pattern), 6, 8
- splitPane/SplitPaneFrame.java, 733
- sprint function (C), 954
- sprintf function (C), 953
- SQL (Structured Query Language), 285–290  
changing data inside databases, 290  
commands in, 293  
data types in, 291, 347–349  
equality testing in, 289  
escapes in, 318–319  
exceptions in, 302–305  
executing statements in, 298–301  
keywords in, 288  
reading instructions from a file, 305  
strings in, 289  
vs. Java, 311  
warnings in, 303  
wildcards in, 289
- SQLException class, 302–305, 323  
and rolling back, 344  
and save points, 347  
getXxx, iterator methods, 302, 304
- SQLPermission class, 520
- SQLWarning class, 303, 323  
getNextWarning method, 304
- SQLXML data type (SQL), 348–349
- Square cap, 788–789
- Square root, computing, 17
- SQuirreL program, 334

- SRC, SRC\_Xxx composition rules, 809–810  
src.jar file, 981  
sRGB standard, 837  
SSL (Secure Sockets Layer), 579  
Standard extensions, 493  
StandardCharsets class, 69  
*StandardJavaFileManager* interface, 444–445  
  getJavaFileObjectsFromXxx methods, 448  
start method (*Matcher*), 135, 137, 140–141  
startDocument method (*ContentHandler*), 204  
startElement method (*ContentHandler*), 200–204  
*Statement* interface, 298–301  
  addBatch method, 345–347  
  close, closeOnCompletion methods, 300, 302  
  execute method, 300, 306, 319, 321  
  executeBatch method, 346–347  
  executeLargeBatch method, 347  
  executeQuery method, 298, 300, 323  
  executeUpdate method, 298, 300, 321, 344  
  getMoreResults method, 320  
  getResultSet method, 300  
  getUpdateCount method, 300, 320  
  getWarnings method, 304  
  isClosed method, 300  
  RETURN\_GENERATED\_KEYS field, 321  
  using for multiple queries, 301  
Statements (databases)  
  closing, 302  
  complex, 311  
  concurrently open, 301  
  executing, 298–301  
  grouping into transactions, 344–347  
  in batch updates, 345  
  multiple, 301  
  prepared, 309–316  
  truncations in, 303  
Static fields, in native code, 960–961  
Static initialization blocks, 944  
Static methods, calling from native code, 964–965  
StAX parser, 205–208, 214–222  
  namespace processing in, 206  
  no indented output in, 214  
stax/StAXTest.java, 206  
StAXSource class, 226  
stopCellEditing method (*CellEditor*), 631–632, 638  
Stored procedures, 318–319  
stream method  
  of Arrays, 8, 36  
  of Collection, 2–5
- Stream* interface  
  collect method, 19, 22, 35  
  concat method, 11  
  count method, 3–4, 12  
  distinct method, 11–12, 43  
  empty method, 5, 8  
  filter method, 3–4, 9–10, 13  
  findAny method, 13  
  findXxx methods, 13  
  flatMap method, 10  
  forEach method, 19, 22  
  generate method, 5, 8, 36  
  iterate method, 5, 8, 12, 36  
  limit method, 10–11, 43  
  map method, 9–10  
  max, min methods, 12  
  of method, 5, 8  
  peek method, 11–12  
  reduce method, 33–35  
  skip method, 10–11  
  sorted method, 11–12  
  toArray method, 19, 22  
  xxxMatch methods, 13  
Streaming parsers, 149, 199–208  
StreamPrintService class, 878  
StreamPrintServiceFactory class, 878  
  getPrintService method, 878–879  
  lookupStreamPrintServiceFactories method, 879  
StreamResult class, 213, 227  
Streams (Java SE 8), 1–46  
  combining, 11  
  converting:  
    between objects and primitive types of, 36  
    to arrays, 19  
  creating, 5–8  
  debugging, 11  
  empty, 5, 34  
  encrypted, 574–575  
  extracting substreams from, 10  
  filters for, 726  
  flattening, 10  
  for print services, 878–879  
  infinite, 3, 5, 10, 12  
  input, 150, 726–731  
  intermediate and terminal operations for, 3, 12  
  of primitive type values, 36–41  
  of random numbers, 37

operations on:  
noninterference of, 43  
threadsafe, 42  
ordered, 42  
parallel, 2, 13, 19, 41–46  
pipeline of, 11  
processed lazily, 3, 12, 43  
reductions of, 12  
sorting, 11  
transformations of, 9–12  
vs. collections, 3

`streams/CountLongWords.java`, 4  
`streams/CreatingStreams.java`, 6  
`streams/PrimitiveTypeStreams.java`, 37

`StreamSource` class, 226  
constructor, 231  
`transform` method, 227

`String` class, 54  
`charAt` method, 9  
`compareTo` method, 393  
`split` method, 64  
`toLowerCase` method, 9  
`trim` method, 153, 379

`STRING` class (`DocFlavor`), 875

`String` parameters, 949–956

`StringBuffer` class, 54, 124

`StringBuilder` class, 54, 73, 688, 727

`Strings`  
encoding, 407  
in Unicode, 371  
fixed-size, I/O of, 73–74  
in native code, 949–956  
in SQL, 289  
internationalizing, 408–410  
ordering, 393  
patterns for, 128–141  
permutations of, 705–706  
printing, 61  
sorting, 394  
splitting, 6  
transforming to lowercase, 9  
writing in binary format, 69

`StringSelection` class, 888, 894

`stringValue` method (`AbstractFormatter`), 693, 701

`Stroke` interface, 788

`stroke/StrokeTest.java`, 791

`StrokePanel` class, 791

`Strokes`, 766, 788–796  
antialiasing, 818  
dash patterns of, 790

end cap styles of, 788–790  
join styles of, 788–789  
setting, 767  
thickness of, 788

`StyledDocument` interface, 681

`StylePad` demo, 682

Stylesheets (XSLT), 222–232

`Subject` class  
`doAs`, `doAsPrivileged` methods, 532–533, 536  
`getPrincipals` method, 536

`Subjects` (logins), 532

`subSequence` method (`CharSequence`), 55

`subtract` method (`Area`), 787–788

Subtrees (Swing), 644, 661  
adding nodes to, 653  
collapsed and expanded, 645

Suetonius, Gaius Tranquillus, 498

sufficient keyword, 532

`sum` method (`XxxStream`), 37, 39–40

summarizing`Xxx` methods (`Collectors`), 20, 23

`summaryStatistics` method (`XxxSummaryStatistics`), 37, 39–40

summing`Xxx` methods (`Collectors`), 29, 33

Sun compiler, 943

SunJCE ciphers, 568

Superclasses, type use annotations in, 468

`Supplier` interface, `get` method, 8

`@SupportedAnnotationTypes` annotation, 476

`SupportedValuesAttribute` interface, 879

`supportsBatchUpdates` method (`DatabaseMetaData`), 345, 347

`supportsResultSetXxx` methods (`DatabaseMetaData`), 323, 328

`@SuppressWarnings` annotation, 470–471

SVG (Scalable Vector Graphics), 211–212

Swing, 581–764  
data transfer in, 904–921  
generating dynamic code for, 449  
loading time in, 921

Swing components  
attaching verifiers to, 690  
desktop panes, 741–760  
drag-and-drop behavior of, 909–912  
editor pane, 712–719  
internal frames, 741–760  
layers, 760–764  
layout managers for, 175–189  
lists, 582–588

- Swing components (*continued*)  
 organizing, 731–764  
 progress bars, 719–722  
 progress monitors, 722–726  
 spinners, 703–712  
 split panes, 732–735  
 tabbed panes, 735–741  
 tables, 599–638  
 text, 681–719  
 trees, 639–680
- Swing Connection newsletter, 694
- SwingWorker class, 720
- Symmetric ciphers, 567–569  
 performance of, 576
- SyncProviderException* interface, 331–332
- System class  
 console method, 406  
 loadLibrary method, 944, 946  
 setSecurityManager method, 516
- SYSTEM identifier (DTD), 164, 210
- System class loader, 493
- System properties, in policy files, 521
- System tray, 932–937
- System.err class, 61
- System.in class, 61  
 and character encoding, 406
- System.out class, 61  
 and character encoding, 406  
 println method, 405–406
- SystemTray class, 932–937  
 add method, 932, 936  
 getSystemTray method, 932, 936  
 getTrayIconSize method, 936  
 isSupported method, 932, 936  
 remove method, 936
- systemTray/SystemTrayTest.java, 934
- T**
- t literal (SQL), 318
- \t, in regular expressions, 130
- Tabbed panes (Swing), 735–741  
 adding/removing tabs in, 736  
 labels of, 737  
 scrolling, 736
- tabbedPane/TabbedPaneFrame.java, 738
- Table cell renderers, 609, 626
- Table index values, 614
- Table models (Swing), 600, 604–608  
 updating after cells were edited, 632
- table/TableTest.java, 603
- TableCellEditor interface, 629, 631  
 getTableCellEditorComponent method, 629, 631, 638
- TableCellRender/ColorTableCellEditor.java, 635
- TableCellRender/ColorTableCellRenderer.java, 635
- TableCellRender/PlanetTableModel.java, 633
- TableCellRender/TableCellRenderFrame.java, 632
- TableCellRenderer interface  
 getTableCellRendererComponent method, 626, 637  
 implementing, 626
- TableColumn class, 611–612, 617  
 constructor, 623
- setCellEditor method, 629, 637
- setCellRenderer method, 637
- setHeaderXxx methods, 627, 637
- setResizable, setWidth, setXxxWidth methods, 611, 623
- TableColumnModel interface, 610  
 getColumn method, 623
- TableModel interface, 614  
 get/setValueAt methods, 605–606, 608, 632  
 getColumnClass method, 609, 621  
 getColumnName method, 606, 608  
 getXxxCount methods, 605–606, 608  
 implementing, 605  
 isCellEditable method, 608, 628
- tableModel/InvestmentTable.java, 606
- tableRowColumn/PlanetTableFrame.java, 617
- TableRowSorter class, 614  
 setStringConverter method, 624
- Tables (databases), 285  
 changing data in, 290  
 creating, 290  
 duplication of data in, 287  
 inspecting, 287  
 metadata for, 333  
 multiple, selecting data from, 289  
 removing, 295
- Tables (Swing), 599–638  
 cells in:  
 editing, 628–629  
 rendering, 626  
 selecting, 612–614, 626
- columns in:  
 accessing, 610  
 adding, 617  
 hiding, 617–625  
 naming, 606  
 rearranging, 601  
 rendering, 609–610

resizing, 602, 611–612  
selecting, 612  
constructing, 601, 605  
headers in, 601  
    rendering, 627  
printing, 602  
relationship between classes of, 610  
rows in:  
    filtering, 615–617  
    hiding, 616  
    margins of, 612  
    resizing, 612  
    selecting, 602, 612  
    sorting, 602, 614–615  
    scrolling, 601  
*TableStringConverter* class, *toString* method, 615, 624  
@Target annotation, 456, 470, 472–473  
TCP (Transmission Control Protocol), 238  
telnet, 233–236  
    activating/connecting, 234  
    several windows communicating simultaneously, 246–247  
template element (XSLT), 224  
Temporal interface, with method, 359  
*TemporalAdjuster* interface, 359  
TemporalAdjusters class, 358–359  
*test/TestDB.java*, 296  
@Test annotation, 455  
@TestCase, @TestCases annotations, 475  
Text, 60–69  
    copying, 887  
    drag and drop of, 904–905  
    encoding of, 67–69  
    formatting, 887  
    generating from XML files, 225–227  
    output, 60–62  
    printing, 852–862, 874  
    reading, 62–63  
    saving objects in, 63–67  
    transferring via clipboard, 888–892  
    transmitting through sockets, 241–249  
    vs. binary data, 60  
Text components (Swing), 681–719  
    hierarchy of, 681  
    monitoring input in, 682–685  
    styled, 681, 712  
Text fields  
    committing, 687  
    editing, 629  
    formatted, 685–703  
    input in:  
        integers, 686  
        turning to upper/lower case, 690  
        validating, 524–530, 683, 685–703  
    losing focus, 687–688  
    reverting, 687  
Text files, encoding of, 405–406  
Text nodes (XML)  
    constructing, 209  
    retrieving from XML, 153  
TextCallbackHandler class, 538  
*textChange/ColorFrame.java*, 683  
*textFile/TextFileTest.java*, 65  
*textFormat/FormatTestFrame.java*, 695  
*textFormat/IntFilter.java*, 698  
*textFormat/IPAddressFormatter.java*, 699  
TextLayout class, *getXxx* methods, 807  
TextStyle enumeration, 387  
TextSyntax class, 882  
TexturePaint class, 797–799  
this keyword, 956  
    annotating, 468–469  
Thread class, *get/setContextClassLoader* methods, 495, 503  
*threaded/ThreadedEchoServer.java*, 247  
ThreadedEchoHandler class, 245–248  
Threads  
    blocking, 49, 250–257, 723  
    executing scripts in, 432  
    for new pages in JEditorPane, 715  
    Internet connections with, 245–248  
    race conditions in, 42  
    referencing class loaders in, 495–496  
Three-tier model, 284–285  
Throw, ThrowNew functions (C), 974–975, 979  
Throwable class, 974  
Thumbnails, 826  
Tiling windows, 744, 746–747  
Time  
    current, 352  
    editing, 691  
    formatting, 365–368, 385–393  
    literals for, 318  
    local, 360–361  
    measuring, 352–353  
    parsing, 367  
    zoned, 361–365  
Time class, 369  
Time of day service, 234

- Time pickers, 705  
Time zones, 385  
TIME, TIMESTAMP data types (SQL), 291, 318, 348  
`timeline/Timeline.java`, 354  
Timeouts, 238–239  
Timer class, 723  
Timestamp class, 369  
Timestamps, 365  
  using instants as, 352  
TimeZone class, and legacy classes, 369  
`toAbsolutePath` method (*Path*), 102–103  
`toArray` method  
  of *AttributeSet*, 886  
  of streams, 19, 22, 36, 38–39–40  
`toCollection` method (Collectors), 20, 23  
`toConcurrentMap` method (Collectors), 25, 27  
`toDays` method (Duration), 353  
`toFile` method (*Path*), 102–103  
`toFormat` method (DateTimeFormatter), 367  
`toHours` method (Duration), 353  
`toInstant` method  
  of Date, 369  
  of ZonedDateTime, 361, 363  
`toLanguageTag` method (Locale), 375, 377  
`toList` method (Collectors), 23  
`toLocaleXxx` methods (ZonedDateTime), 363  
`toLowerCase` method (String), 9  
`toMap` method (Collectors), 24–27  
`toMillis`, `toMinutes`, `toNanos` methods (Duration), 353  
`toNanoOfDay`, `toSecondOfDay` methods (LocalTime), 360  
`Tool` interface, `run` method, 447  
Toolkit class, `getSystemClipboard` method, 888, 891  
ToolProvider class, `getSystemJavaCompiler` method,  
  443  
tools.jar file, 443  
Tooltips, for tray icons, 932  
`toPath` method (*File*), 102–103  
`toPattern` method (SimpleDateFormat), 712  
Top-level windows, opening, 509  
`toSeconds` method (Duration), 353  
`toSet` method (Collectors), 20, 23, 29  
`toString` method  
  implementing with annotations, 477–480  
  of *Annotation*, 464  
  of *CharSequence*, 55  
  of *Currency*, 385  
  of *DefaultFormatter*, 691  
  of *Locale*, 377  
  of *TableStringConverter*, 615, 624  
  of *Variable*, 674  
  
toZonedDateTime method (*GregorianCalendar*),  
  369  
Transactions, 344–347  
  committing/rolling back, 344  
  error handling in, 346  
Transfer handlers, 906–907  
  adding, 910  
Transfer wrapper, 899  
*Transferable* interface, 892–894  
  `getTransferData` method, 892  
  `getTransferDataFlavors` method, 894  
  implementing, 888, 894, 902  
  `isDataFlavorSupported` method, 892  
TransferHandler class, 909–912  
  `canImport` method, 912, 918  
  constructor, 909  
  `createTransferable` method, 909, 912  
  `exportAsDrag` method, 910, 912  
  `exportDone` method, 911–912  
  `getSourceActions` method, 909, 912  
  `importData` method, 912–914, 918  
TransferHandler.DropLocation class, 914  
  `getDropPoint` method, 920  
TransferSupport class  
  `get/setDropAction` methods, 919  
  `getComponent`, `getDataFlavors` methods, 919  
  `getDropLocation` method, 914, 919  
  `getSourceDropActions` method, 919  
  `getTransferable` method, 913  
  `getUserDropAction` method, 919  
  `isDrop` method, 914, 919  
`transferText/TextTransferFrame.java`, 890  
`transform` method  
  of *Graphics2D*, 767, 802, 804  
  of *StreamSource*, 227  
  of *Transformer*, 213, 226  
`transform/makehtml.xsl`, 227  
`transform/makeprop.xsl`, 228  
`transform/TransformTest.java`, 228  
Transformations, 766, 799–804  
  affine, 802, 843  
  composing, 800–801  
  fundamental types of, 800  
  matrices for, 802  
  order of, 800  
  setting, 767  
  using for printing, 864  
Transformer class  
  `setOutputProperty` method, 213  
  `transform` method, 213, 226

- TransformerFactory class  
  newInstance method, 213  
  newTransformer method, 213, 231
- transient, 92
- Transitional events, 584
- translate, 800, 804, 864
- Translation, 800
- Transparency, 807–817
- Traversal order  
  in a spinner, 704–705  
  in a tree:  
    breadth-first vs. depth-first, 659  
    postorder, 659
- Tray icons, 932–933  
  pop-up menus for, 933  
  tooltips for, 932
- TrayIcon class, 933  
  add/removeActionListener methods, 937  
  constructor, 936  
  displayMessage method, 937  
  get/setImage methods, 936  
  get/setPopupMenu methods, 936  
  get/setTooltip methods, 936  
  is/setImageAutoSize methods, 937
- Tree events, 664–671
- Tree models  
  constructing, 641, 672  
  custom, 671–680  
  default, 641
- Tree parsers, 149
- Tree paths, 650–658  
  constructing, 653, 660
- Tree selection listeners, 664  
  adding to a tree, 664
- tree/SimpleTreeFrame.java, 643
- TreeCellRenderer interface, 661–664  
  getTreeCellRendererComponent method, 662–663  
  implementing, 661
- treeEdit/TreeEditFrame.java, 655
- TreeModel interface, 640, 650  
  add/removeTreeModelListener method, 672, 680  
  getChild method, 680  
  getChild, getRoot methods, 154, 672–674  
  getChildCount method, 672–674, 679  
  getIndexofChild method, 672, 680  
  getRoot method, 679  
  implementing, 154, 641  
  isLeaf method, 649, 672, 680  
  valueForPathChanged method, 673, 680
- treeModel/ObjectInspectorFrame.java, 675
- treeModel/ObjectTreeModel.java, 676
- treeModel/Variable.java, 678
- TreeModelEvent class, 680
- TreeModelListener interface, 672  
  treeNodesXxx, treeStructureChanged methods, 673, 680
- TreeNode interface, 641, 650  
  children, getChildXxx methods, 658  
  getAllowsChildren method, 649  
  getParent method, 658, 660  
  isLeaf method, 648–649
- TreePath class, 651  
  getLastPathComponent method, 651, 658
- treeRender/ClassNameTreeCellRenderer.java, 670
- treeRender/ClassTreeFrame.java, 666
- Trees (Swing), 639–680  
  adding listeners to, 664  
  background color for, 661  
  connecting lines in, 645–646  
  displaying, 640–650  
  editing, 650–658, 673  
  handles in, 644, 646, 661  
  hierarchy of classes for, 641  
  indexes in, 652  
  infinite, 675  
  leaves in, 639, 647–648, 661, 672  
  nodes in, 639, 648, 661, 672  
  paired with other components, 664  
  rendering, 661–664  
  scrolling to newly added nodes, 653–654  
  structure of, 639  
  subtrees in, 644  
    collapsing/expanding, 645  
  traversals for, 659–661  
  updating vs. reloading, 653  
  user objects for, 642  
    changing, 652  
  view of, 652–653  
  with horizontal lines, 646
- TreeSelectionEvent class  
  getPath method, 671  
  getPaths method, 665, 671
- TreeSelectionListener interface  
  implementing, 664–671  
    valueChanged method, 664, 666, 671
- TreeSelectionModel interface, 665
- trim method (String), 153, 379
- Troubleshooting. *See* Debugging
- True Odds: How Risks Affect Your Everyday Life (Walsh), 547

- tryLock method (*FileChannel*)method (*Lock*),  
    126–128  
try-with-resources statement, 53, 112  
    closing files with, 110  
    for database connections, 302  
    with locks, 127  
ts literal (SQL), 318  
Type bounds, type use annotations in, 468  
TYPE\_BYTE\_GRAY field (*BufferedImage*), 838  
Type codes, 87, 961  
Type definitions, 172  
    anonymous, 173  
    nesting, 173  
TYPE\_INT\_ARGB field (*BufferedImage*), 835–836, 840  
Type parameters, annotating, 466  
Type use annotations, 467  
TYPE\_XXX fields (*AffineTransformOp*), 843, 850  
*TypeElement* interface, 477  
Types. *See* Data types  
Typesafe enumerations, 94–95
- U**
- \u in masks, 692  
\u, in regular expressions, 130  
UDP (User Datagram Protocol), 238  
UIManager class, 626  
Unicode standard  
    and input/output streams, 48  
    and native code, 950  
    character order in, 393  
    converting to binary data, 60  
    in property files, 410  
    in regular expressions, 133  
    normalization forms in, 394  
    using for all strings, 371  
uninstallUI method (*LayerUI*), 764  
Units of measurement, 148  
UNIX operating system  
    authentications in, 531  
    file names starting with a period in,  
        516  
    Java deployment directory in, 564  
    line feed in, 61, 405  
    paths in, 101  
UnixLoginModule class, 532  
UnixNumericGroupPrincipal class, 532  
UnixPrincipal class, 531–532  
    getName method, 531  
UnknownHostException, 237  
unordered method (*BaseStream*), 43, 45
- Unparsed external entities, 168  
unread method (*PushbackInputStream*), 59  
UnsatisfiedLinkError, 941  
UnsupportedFlavorException, 892  
until method (*LocalDate*), 356–357  
update method  
    of Cipher, 568, 571, 573, 575  
    of MessageDigest, 549–550  
    of SplashScreen, 922, 926  
UPDATE statement (SQL), 290, 310, 324  
executing, 298, 300, 316  
in batch updates, 345  
truncations in, 303  
    vs. methods of *ResultSet*, 325  
updateObject method (*ResultSet*), 301  
updateXxx methods (*ResultSet*), 324–325, 328  
Upper case, turning characters to, 690  
URI class  
    getXxx methods, 258  
    no mailto support in, 927  
    no resource accessing with, 257  
URIs (Uniform Resource Identifiers), 257  
    absolute vs. relative, 258  
    base, 259  
    for e-mails, 927  
    hierarchical, 258  
    namespace, 196–199  
    opaque vs. nonopaque, 258  
    schemes for, 258  
URISyntax class, 882  
URL class (*DocFlavor*), 875  
URL class (*java.lang.Object*), 257–259  
    accepted schemes for, 257  
    openConnection method, 259, 265  
    openStream method, 257, 265  
URLClassLoader class, 493  
    constructor, 503  
    loadClass method, 494  
URLConnection class, 257, 259–267  
    connect method, 259, 262, 266  
    get/setAllowUserInteraction methods, 259, 261,  
        266  
    get/setConnectTimeout methods, 259, 266  
    get/setDoInput, get/setDoOutput methods, 259–260,  
        265  
    get/setIfModifiedSince methods, 259, 261, 266  
    get/setReadTimeout methods, 259, 266  
    get/setRequestProperty methods, 259, 261, 266  
    get/setUseCaches methods, 259, 261, 266  
    getContent method, 267

- getContentXxx, getDate, getExpiration,  
    getLastModified methods, 260, 263, 267
- getHeaderXxx methods, 260–262, 267
- getInputStream method, 260, 267, 271, 273
- getOutputStream method, 260, 267, 270
- getUseCaches method, 266
- setDoOutput method, 270
- URLConnection/URLConnectionTest.java, 263
- URLDecoder class, decode method, 277
- URLEncoder class, encode method, 277
- URLs (Uniform Resource Locators), 257
- attaching parameters to, 269
  - connections via, 257
  - constructing from strings, 691
  - encoding, 269–270
  - for databases, 292
  - for local files, 517
  - for namespace identifiers, 196
  - relative vs. absolute, 563
    - for DTDs, 164
    - resolving, 259
- URNs (Uniform Resource Names), 257
- US Letter paper, 855
- useLocale method (Scanner), 380
- User coordinates, 799
- User objects, 642
  - changing, 652
- Users
  - authentication of, 530–546
  - permissions for, 533
  - preferences of, 126
- UTC (Coordinated Universal Time), 362
- UTF-16 standard, 60, 67–68, 70
  - and native code, 949
  - in regular expressions, 130
- UTF-8 standard, 67–69, 70
  - for text files, 405–406
  - modified, 70–72, 407, 949–952
- V**
- V (void) type code, 961
- \v, \V, in regular expressions, 131
- Validation, 162–189
  - activating, 168
  - adding to classes, 92
  - of input, 683, 685–703
- valueChanged method
  - of *ListSelectionListener*, 584, 588
  - of *TreeSelectionListener*, 664, 666, 671
- valueForPathChanged method (*TreeModel*), 673, 680
- value-of element (XSLT), 225
- Values
  - grouping, 28–30
  - partitioning, 28
- valueToString method (*AbstractFormatter*), 693, 701
- VARCHAR data type (SQL), 291, 348
- Variable class, 673
  - toString method, 674
- VariableElement* interface, 477
- Variables
  - annotating, 456, 467
  - binding, 432
  - fields of, 674
  - initializing, 504
  - scope of, 433
- Variants, in locales, 373, 410
- Vendor name, of a reader, 825
- Verifier/VerifierTest.java, 507
- Verifiers, 504–508, 690
  - verify method (*InputVerifier*), 690
- VeriSign, Inc., 555, 557, 561
- Version number, of a reader, 825
- Versioning, 95–98
- VERTICAL\_SPLIT value (*JSplitPane*), 732
- VERTICAL, VERTICAL\_WRAP values (*JList*), 583
- Vetoable change listeners, 748
- VetoableChangeListener* interface
  - implementing, 749
  - vetoableChange method, 749, 759
- Vetoing, 748–750
- view/ViewDB.java, 335
- visitFile, visitFileFailed methods
  - of *FileVisitor*, 112
  - of *SimpleFileVisitor*, 113, 115
- Visual representation, 284
- W**
- \w, \W, in regular expressions, 131
- walk method (*Files*), 110
- walkFileTree method (*Files*), 112–114
- warning method (*ErrorHandler*), 170–171
- Warnings
  - in applets, 526
  - in SQL, 303
  - suppressing, 471
- WBMP format, 823
- WeakReference object, 675
- Web applications, connection management
  - in, 349–350
- Web containers, 472

- Web crawlers, 200  
with SAX parser, 201–202  
with StAX parser, 206
- Web pages  
dynamic, 449–454  
separating applet class loaders for, 496
- WebRowSet* interface, 329
- WHERE statement (SQL), 289
- Whitespace  
ignoring, while parsing, 153  
in e-mail URIs, 270, 927  
in regular expressions, 131, 133  
in XML Schema, 174
- Wildcards, type use annotations in, 468
- Wilde, Oscar, 372
- win32reg/Win32RegKey.c*, 993
- win32reg/Win32RegKey.java*, 991
- win32reg/Win32RegKeyTest.java*, 1000
- Win32RegKey* class, 987, 990  
get/setValue methods, 987–988  
names method, 987
- Win32RegKeyNameEnumeration* class, 989–990
- Windows operating system  
activating telnet in, 234  
authentication in, 533  
character encodings in, 404  
classpath in, 292  
compiling invocation API, 984  
cut and paste in, 887  
drag and drop in, 904  
dynamic linking in, 981  
frame dragging in, 751  
glob syntax in, 112  
Java deployment directory in, 564, 566  
line feed in, 61, 405  
multiple document interface of, 741–742  
paths in, 56, 101  
permissions in, 520  
registry, accessing from native code, 985–1001  
resources in, 408  
system tray in, 932  
using Microsoft compiler, 944
- Windows look-and-feel  
cascading/tiling in, 744  
trees in, 645
- with method (*Temporal*), 359
- withLocale method (*DateTimeFormatter*), 365, 392
- withXxx methods (Date and Time API), 356, 360, 363
- WordCheckPermission class, 524–530
- Words  
counting, in a book, 2  
in regular expressions, 131
- Working directory, 56
- wrap method (*ByteBuffer*), 123, 125
- WritableByteChannel* interface, 250
- WritableRaster* class, 835  
setDataElements method, 837, 841  
setPixel, setPixels methods, 835, 841
- write method  
of *CipherOutputStream*, 575  
of *Files*, 104  
of *ImageIO*, 824, 831  
of *ImageWriter*, 827, 834  
of *OutputStream*, 48, 50–51  
write/RectangleComponent.java, 218  
write/XMLWriteFrame.java, 215  
write/XMLWriteTest.java, 215  
writeExternal method (*Externalizable*), 93–94  
writeFixedString method (*DataIO*), 73–74  
writeInsert method (*ImageWriter*), 827, 834  
writeObject method  
of *Date*, 93  
of *ObjectOutputStream*, 80, 85, 92
- Writer* class, 48, 52–53  
write method, 51  
writeUTF method (*DataOutput*), 70, 72
- writeXxx methods (*DataOutput*), 69, 72, 81
- writeXxx methods (*XMLStreamWriter*), 214, 221–222
- X**
- X Window System  
cut and paste in, 887  
frame dragging in, 751
- X.509 format, 554
- \x, in regular expressions, 130
- XHTML (Extensible Hypertext Markup Language), 200
- XML (Extensible Markup Language), 143–232  
annotated version of the standard, 146  
case sensitivity of, 146  
end and empty tags in, 146  
hierarchical structures in, 144–145  
in databases, 349  
Java support of, 494  
namespaces in, 196–199  
using for layout description, 175–189  
vs. HTML, 146

- XML documents  
DTDs in, 147, 163–164  
format of, 145  
generating, 208–222  
  from non-XML legacy data, 227  
  HTML files from, 222–225  
  plain text from, 225–227  
locating information in, 190–196  
parsing, 149–162  
structure of, 146–149–150, 163  
validating, 162–189  
with/without namespaces, 209–210  
writing, 210–211  
  with StAX, 214–222
- XML Schema, 163, 172–174  
  attributes in, 174  
  parsing with, 174  
  referencing in XML documents, 172  
  repeated elements in, 174  
  type definitions in, 172  
    anonymous, 173  
    nesting, 173  
  whitespace in, 174
- XMLInputFactory class  
  createXMLStreamReader method, 207  
  newInstance method, 207  
  setProperty method, 206–207  
  xmlns attribute (XSLT), 197
- XMLOutputFactory class  
  createXMLStreamWriter method, 214, 221  
  newInstance method, 214, 221
- XMLReader interface  
  implementing, 226  
  parse method, 232  
  setContentHandler method, 232
- XMLStreamReader interface  
  getAttributeXxx methods, 206, 208  
  getName, getLocalName, getText methods, 208  
  hasNext, next methods, 208  
  isXxx methods, 208
- XMLStreamWriter interface, 214  
  close method, 222  
  setDefaultNamespace, setPrefix methods,  
    221  
  writeXxx methods, 214, 221–222
- XOR composition rule, 810
- XPath, 190–196  
  elements/attributes in, 190  
  expressions in, 191
- XPath interface, evaluate method, 191, 196  
  xpath/XPathTester.java, 192  
  XPathFactory class  
    newInstance method, 191, 195  
    newPath method, 195  
  xsd:, xsd: prefixes (XSL Schema), 172  
  xsd:attribute element (XML Schema), 174  
  xsd:choice element (XML Schema), 173  
  xsd:complexType element (XML Schema), 173  
  xsd:element element (XML Schema), 172  
  xsd:enumeration element (XML Schema), 172  
  xsd:schema element (XML Schema), 174  
  xsd:sequence element (XML Schema), 173  
  xsd:simpleType element (XML Schema), 172  
  xsl:apply-templates element (XSLT), 224  
  xsl:output element (XSLT), 224  
  xsl:template element (XSLT), 224  
  xsl:value-of element (XSLT), 225  
  XSLT (Extensible Stylesheet Language  
    Transformations), 210, 222–232  
    copying attribute values in, 225  
    templates in, 224  
  XSLT processor, 222
- Y
- Year, YearMonth classes, 357  
Years, leap, 356
- Z
- z (boolean), type code, 87, 961  
\z, \Z, in regular expressions, 132  
ZIP archives, 77–80  
  reading, 77  
    numbers from, 58  
  writing, 77
- ZipEntry class, methods of, 79
- ZipFile class, methods of, 80
- ZipInputStream class, 51, 77  
  closeEntry, getNextEntry methods, 77–78  
  constructor, 78  
  read method, 77
- ZipOutputStream class, 51, 77  
  closeEntry, putNextEntry methods, 77–78  
  constructor, 78  
  setLevel, setMethod methods, 78
- ZonedDateTime class  
  and legacy classes, 369  
  methods of, 361–365, 369, 386, 393  
  zonedtimes/ZonedDateTime.java, 364